Part I

Finite Element Methods

Chapter 1

Approximation of Functions

1.1 Approximation of Functions

Many successful numerical methods for differential equations aim at approximating the unknown function by a sum

$$u(x) = \sum_{i=0}^{N} c_i \varphi_i(x), \tag{1.1}$$

where $\varphi_i(x)$ are prescribed functions and c_i, $i = 0, \ldots, N$, are unknown coefficients to be determined. Solution methods for differential equations utilizing (1.1) must have a *principle* for constructing $N+1$ equations to determine $c_0, \ldots, c_N$. Then there is a *machinery* regarding the actual constructions of the equations for $c_0, \ldots, c_N$ in a particular problem. Finally, there is a *solve* phase for computing solution $c_0, \ldots, c_N$ of the $N+1$ equations.

Especially in the finite element method, the machinery for constructing the equations is quite comprehensive, with many mathematical and implementational details entering the scene at the same time. From a pedagogical point of view it can therefore be wise to introduce the computational machinery for a trivial equation, namely $u = f$. Solving this equation with f given and u on the form (1.1) means that we seek an approximation u to f. This approximation problem has the advantage of introducing most of the finite element toolbox, but with postponing variational forms, integration by parts, boundary conditions, and coordinate mappings. It is therefore from a pedagogical point of view advantageous to become familiar with finite element *approximation* before addressing finite element methods for differential equations.

First, we refresh some linear algebra concepts about approximating vectors in vector spaces. Second, we extend these concepts to approximating functions in function spaces, using the same principles and the same notation. We present examples on approximating functions by global basis functions with support throughout the entire domain. Third, we introduce the finite element type of local basis functions and explain the computational algorithms for working with such functions. Three types of approximation principles are covered: 1) the least squares method, 2) the Galerkin method, and 3) interpolation or collocation.

1.2 Approximation of Vectors

1.2.1 Approximation of Planar Vectors

Suppose we have given a vector $\boldsymbol{f} = (3, 5)$ in the x-y plane and that we want to approximate this vector by a vector aligned in the direction of the vector (a, b). We introduce the vector space V spanned by the vector $\boldsymbol{\varphi}_0 = (a, b)$:

$$V = \text{span}\,\{\boldsymbol{\varphi}_0\}\,. \tag{1.2}$$

We say that $\boldsymbol{\varphi}_0$ is a basis vector in the space V. Our aim is to find the vector $\boldsymbol{u} = c_0\boldsymbol{\varphi}_0 \in V$ which best approximates the given vector $\boldsymbol{f} = (3, 5)$. A reasonable criterion for a best approximation could be to minimize the length of the difference between the approximate $\boldsymbol{u}$ and the given $\boldsymbol{f}$. The difference, or error, $\boldsymbol{e} = \boldsymbol{f} - \boldsymbol{u}$ has its length given by the *norm*

$$||\boldsymbol{e}|| = (\boldsymbol{e}, \boldsymbol{e})^{\frac{1}{2}},$$

where $(\boldsymbol{e}, \boldsymbol{e})$ is the *inner product* of $\boldsymbol{e}$ and itself. The inner product, also called *scalar product* or *dot product*, of two vectors $\boldsymbol{u} = (u_0, u_1)$ and $\boldsymbol{v} = (v_0, v_1)$ is defined as

$$(\boldsymbol{u}, \boldsymbol{v}) = u_0v_0 + u_1v_1\,. \tag{1.3}$$

Here we should point out that we use the notation $(\cdot, \cdot)$ for two different things: (a, b) for scalar quantities a and b means the vector starting in the origin and ending in the point (a, b), while $(\boldsymbol{u}, \boldsymbol{v})$ with vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ means the inner product of these vectors. Since vectors are here written in boldface font there should be no confusion. Note that the norm associated with this inner product is the usual Eucledian length of a vector.

We now want to find c_0 such that it minimizes $||\boldsymbol{e}||$. The algebra is simplified if we minimize the square of the norm, $||\boldsymbol{e}||^2 = (\boldsymbol{e}, \boldsymbol{e})$. Define

$$E(c_0) = (\boldsymbol{e}, \boldsymbol{e}) = (\boldsymbol{f} - c_0\boldsymbol{\varphi}_0, \boldsymbol{f} - c_0\boldsymbol{\varphi}_0)\,. \tag{1.4}$$

We can rewrite the expressions of the right-hand side to a more convenient form for further work:

$$E(c_0) = (\boldsymbol{f}, \boldsymbol{f}) - 2c_0(\boldsymbol{f}, \boldsymbol{\varphi}_0) + c_0^2(\boldsymbol{\varphi}_0, \boldsymbol{\varphi}_0)\,. \tag{1.5}$$

The rewrite results from using the following fundamental rules for inner product spaces[1]:

$$(\alpha\boldsymbol{u}, \boldsymbol{v}) = \alpha(\boldsymbol{u}, \boldsymbol{v}), \quad \alpha \in \mathbb{R}, \tag{1.6}$$

$$(\boldsymbol{u} + \boldsymbol{v}, \boldsymbol{w}) = (\boldsymbol{u}, \boldsymbol{w}) + (\boldsymbol{v}, \boldsymbol{w}), \tag{1.7}$$

[1] It might be wise to refresh some basic linear algebra by consulting a textbook. Exercises 1.1 and 1.2 suggest specific tasks to regain familiarity with fundamental operations on inner product vector spaces.

$$(\boldsymbol{u},\boldsymbol{v}) = (\boldsymbol{v},\boldsymbol{u})\,. \tag{1.8}$$

Minimizing $E(c_0)$ implies finding c_0 such that

$$\frac{\partial E}{\partial c_0} = 0\,.$$

Differentiating (1.5) with respect to c_0 gives

$$\frac{\partial E}{\partial c_0} = -2(\boldsymbol{f},\boldsymbol{\varphi}_0) + 2c_0(\boldsymbol{\varphi}_0,\boldsymbol{\varphi}_0)\,.$$

Setting the above expression equal to zero and solving for c_0 gives

$$c_0 = \frac{(\boldsymbol{f},\boldsymbol{\varphi}_0)}{(\boldsymbol{\varphi}_0,\boldsymbol{\varphi}_0)}, \tag{1.9}$$

which in the present case with $\boldsymbol{\varphi}_0 = (a,b)$ results in

$$c_0 = \frac{3a+5b}{a^2+b^2}\,. \tag{1.10}$$

Minimizing $||\boldsymbol{e}||^2$ implies that $\boldsymbol{e}$ is orthogonal to the approximation $c_0\boldsymbol{\varphi}_0$. Straight calculation shows this (recall that two vectors are orthogonal when their inner product vanishes):

$$(\boldsymbol{e}, c_0\boldsymbol{\varphi}_0) = (\boldsymbol{f} - c_0 v_0, v_0) = (\boldsymbol{f},\boldsymbol{\varphi}_0) - \frac{(\boldsymbol{f},\boldsymbol{\varphi}_0)}{(\boldsymbol{\varphi}_0,\boldsymbol{\varphi}_0)}(\boldsymbol{\varphi}_0,\boldsymbol{\varphi}_0) = 0\,.$$

Therefore, instead of minimizing the square of the norm, we could demand that $\boldsymbol{e}$ is orthogonal to any vector in V. That is,

$$(\boldsymbol{e},\boldsymbol{v}) = 0, \quad \forall \boldsymbol{v} \in V\,. \tag{1.11}$$

Since an arbitrary $\boldsymbol{v} \in V$ can be expressed in terms of the basis of V, $\boldsymbol{v} = c_0\boldsymbol{\varphi}_0$, with an arbitrary $c = 0 \in \mathbb{R}$, (1.11) implies

$$(\boldsymbol{e}, c_0\boldsymbol{\varphi}_0) = c_0(\boldsymbol{e},\boldsymbol{\varphi}_0) = 0,$$

which means that

$$(\boldsymbol{e},\boldsymbol{\varphi}_0) = 0 \quad \Leftrightarrow \quad (\boldsymbol{f} - c_0\boldsymbol{\varphi}_0,\boldsymbol{\varphi}_0) = 0\,.$$

The latter equation gives (1.9) for c_0.

1.2.2 Approximation of General Vectors

Let us generalize the vector approximation from the previous section to vectors in spaces with arbitrary dimension. Given some vector $\boldsymbol{f}$, we want to find the best approximation to this vector in the space

$$V = \text{span}\,\{\boldsymbol{\varphi}_0,\ldots,\boldsymbol{\varphi}_N\}\,.$$

We assume that the *basis vectors* $\boldsymbol{\varphi}_0,\ldots,\boldsymbol{\varphi}_N$ are linearly independent so that none of them are redundant and the space has dimension $N+1$. Any vector $\boldsymbol{u}\in V$ can be written as a linear combination of the basis vectors,

$$\boldsymbol{u}=\sum_{j=0}^{N} c_j\boldsymbol{\varphi}_j,$$

where $c_j\in\mathbb{R}$ are scalar coefficients to be determined.

Now we want to find $c_0,\ldots,c_N$ such that $\boldsymbol{u}$ is the best approximation to $\boldsymbol{f}$ in the sense that the distance, or error, $\boldsymbol{e}=\boldsymbol{f}-\boldsymbol{u}$ is minimized. Again, we define the squared distance as a function of the free parameters $c_0,\ldots,c_N$,

$$\begin{aligned} E(c_0,\ldots,c_N) &= (\boldsymbol{e},\boldsymbol{e}) = (\boldsymbol{f}-\sum_j c_j\boldsymbol{\varphi}_j, \boldsymbol{f}-\sum_j c_j\boldsymbol{\varphi}_j) \\ &= (\boldsymbol{f},\boldsymbol{f}) - 2\sum_{j=0}^{N} c_j(\boldsymbol{f},\boldsymbol{\varphi}_j) + \sum_{p=0}^{N}\sum_{q=0}^{N} c_pc_q(\boldsymbol{\varphi}_p,\boldsymbol{\varphi}_q)\,. \end{aligned} \tag{1.12}$$

Minimizing this E with respect to the independent variables $c_0,\ldots,c_N$ is obtained by setting

$$\frac{\partial E}{\partial c_i}=0,\quad i=0,\ldots,N\,.$$

The second term in (1.12) is differentiated as follows:

$$\frac{\partial}{\partial c_i}\sum_{j=0}^{N} c_j(\boldsymbol{f},\boldsymbol{\varphi}_j) = c_i(\boldsymbol{f},\boldsymbol{\varphi}_i), \tag{1.13}$$

since the expression to be differentiated is a sum and only one term contains c_i (write out specifically for, e.g, $N=3$ and $i=1$). The last term in (1.12) is more tedious to differentiate. We start with

$$\frac{\partial}{\partial c_i}c_pc_q = \begin{cases} 0, & \text{if } p\neq i \text{ and } q\neq i,\\ c_q, & \text{if } p=i \text{ and } q\neq i,\\ c_p, & \text{if } p\neq i \text{ and } q=i,\\ 2c_i, & \text{if } p=q=i, \end{cases} \tag{1.14}$$

Then

$$\frac{\partial}{\partial c_i}\sum_{p=0}^{N}\sum_{q=0}^{N} c_pc_q(\boldsymbol{\varphi}_p,\boldsymbol{\varphi}_q) = \sum_{p=0,p\neq i}^{N} c_p(\boldsymbol{\varphi}_p,\boldsymbol{\varphi}_i) + \sum_{q=0,q\neq i}^{N} c_q(\boldsymbol{\varphi}_q,\boldsymbol{\varphi}_i) + 2c_i(\boldsymbol{\varphi}_i,\boldsymbol{\varphi}_i)\,.$$

The last term can be included in the other two sums, resulting in

$$\frac{\partial}{\partial c_i}\sum_{p=0}^{N}\sum_{q=0}^{N} c_pc_q(\boldsymbol{\varphi}_p,\boldsymbol{\varphi}_q) = 2\sum_{j=0}^{N} c_i(\boldsymbol{\varphi}_j,\boldsymbol{\varphi}_i)\,. \tag{1.15}$$

It then follows that setting

$$\frac{\partial E}{\partial c_i} = 0, \quad i = 0, \ldots, N,$$

leads to a linear system for $c_0, \ldots, c_N$:

$$\sum_{j=0}^{N} A_{i,j} c_j = b_i, \quad i = 0, \ldots, N, \tag{1.16}$$

where

$$A_{i,j} = (\boldsymbol{\varphi}_i, \boldsymbol{\varphi}_j), \tag{1.17}$$

$$b_i = (\boldsymbol{\varphi}_i, \boldsymbol{f})\,. \tag{1.18}$$

(Note that we can change the order of the two vectors in the inner product as desired.)

In analogy with the "one-dimensional" example in Chapter 1.2.1, it holds also here in the general case that minimizing the distance (error) $\boldsymbol{e}$ is equivalent to demanding that $\boldsymbol{e}$ is orthogonal to all $\boldsymbol{v} \in V$:

$$(\boldsymbol{e}, \boldsymbol{v}) = 0, \quad \forall \boldsymbol{v} \in V\,. \tag{1.19}$$

Since any $\boldsymbol{v} \in V$ can be written as $\boldsymbol{v} = \sum_{i=0}^{N} c_i \boldsymbol{\varphi}_i$, the statement (1.19) is equivalent to saying that

$$(\boldsymbol{e}, \sum_{i=0}^{N} c_i \boldsymbol{\varphi}_i) = 0,$$

for any choice of coefficients $c_0, \ldots, c_N \in \mathbb{R}$. The latter equation can be rewritten as

$$\sum_{i=0}^{N} c_i (\boldsymbol{e}, \boldsymbol{\varphi}_i) = 0\,.$$

If this is to hold for arbitrary values of $c_0, \ldots, c_N$, we must require that each term in the sum vanishes,

$$(\boldsymbol{e}, \boldsymbol{\varphi}_i) = 0, \quad i = 0, \ldots, N\,. \tag{1.20}$$

These $N + 1$ equations result in the same linear system as (1.16). Instead of differentiating the $E(c_0, \ldots, c_N)$ function, we could simply use (1.19) as the principle for determining $c_0, \ldots, c_N$, resulting in the $N + 1$ equations (1.20).

One often refers to the procedure of minimizing $||\boldsymbol{e}||^2$ as a *least squares method* or *least squares approximation*. The rationale for this name is that $||\boldsymbol{e}||^2$ is a sum of squared differences between the components in $\boldsymbol{f}$ and $\boldsymbol{u}$. We find $\boldsymbol{u}$ such that this sum of squares is minimized.

The principle (1.19), or the equivalent form (1.20), corresponds what is known as a *Galerkin method* when we later use the same reasoning to approximate functions in function spaces.

1.3 Global Basis Functions

Let V be a function space spanned by a set of *basis functions* $\varphi_0, \ldots, \varphi_N$,

$$V = \text{span}\,\{\varphi_0, \ldots, \varphi_N\},$$

such that any function $u \in V$ can be written as a linear combination of the basis functions:

$$u = \sum_{j=0}^{N} c_j \varphi_j \,. \tag{1.21}$$

For now, in this introduction, we shall look at functions of a single variable x: $u = u(x)$, $\varphi_i = \varphi_i(x)$, $i = 0, \ldots, N$. Later, we will extend the scope to functions of two- or three-dimensional space. The approximation (1.21) is typically used to discretize a problem in space. Other methods, most notably finite differences, are common for time discretization (although the form (1.21) can be used in time too).

1.3.1 The Least-Squares Method

Given a function $f(x)$, how can we determine its best approximation $u(x) \in V$? A natural starting point is to apply the same reasoning as we did for vectors in Chapter 1.2.2. That is, we minimize the distance between u and f. However, this requires a norm for measuring distances, and a norm is most conveniently defined through an inner product. Viewing a function as a vector of infinitely many point values, one for each value of x, the inner product could intuitively be defined as the usual summation of pairwise components, with summation replaced by integration:

$$(f, g) = \int f(x) g(x)\, dx \,.$$

To fix the integration domain, we let $f(x)$ and $\varphi_i(x)$ be defined for a domain $\Omega \subset \mathbb{R}$. The inner product of two functions $f(x)$ and $g(x)$ is then

$$(f, g) = \int_{\Omega} f(x) g(x)\, dx \,. \tag{1.22}$$

The distance between f and any function $u \in V$ is simply $f - u$, and the squared norm of this distance is

$$E = (f(x) - \sum_{j=0}^{N} c_j \varphi_j(x), f(x) - \sum_{j=0}^{N} c_j \varphi_j(x)) \,. \tag{1.23}$$

Note the analogy with (1.12): the given function f plays the role of the given vector $\boldsymbol{f}$, and the basis function φ_i plays the role of the basis vector $\boldsymbol{\varphi}_i$. We get can rewrite (1.23), through similar stepss as used for the result (1.12), leading to

$$E(c_0,\ldots,c_N) = (f,f) - 2\sum_{j=0}^{N} c_j(f,\varphi_i) + \sum_{p=0}^{N}\sum_{q=0}^{N} c_p c_q(\varphi_p,\varphi_q)\,. \tag{1.24}$$

Minimizing this function of $N+1$ scalar variables $c_0,\ldots,c_N$ requires differentiation with respect to c_i, for $i=0,\ldots,N$. This action gives a linear system of the form (1.16), with

$$A_{i,j} = (\varphi_i,\varphi_j) \tag{1.25}$$

$$b_i = (f,\varphi_i)\,. \tag{1.26}$$

As in Chapter 1.2.2, the minimization of (e,e) is equivalent to

$$(e,v) = 0, \quad \forall v \in V\,. \tag{1.27}$$

This is known as the *Galerkin method*. Using the same reasoning as in (1.19)–(1.20), it follows that (1.27) is equivalent to

$$(e,\varphi_i) = 0, \quad i = 0,\ldots,N\,. \tag{1.28}$$

Since (1.27) and (1.28) are equivalent to minimizing (e,e), the coefficient matrix and right-hand side implied by (1.28) are given by (1.25) and (1.26).

1.3.2 Example: Linear Approximation

Let us apply the theory in the previous section to a simple problem: given a parabola $f(x) = x^2 + x + 1$ for $x \in \Omega = [1,2]$, find the best approximation $u(x)$ in the space of all linear functions:

$$V = \text{span}\,\{1, x\}\,.$$

That is, $\varphi_0(x) = 1$, $\varphi_1(x) = x$, and $N = 1$. We seek

$$u = c_0\varphi_0(x) + c_1\varphi_1(x) = c_0 + c_1 x,$$

where c_0 and c_1 are found by solving a 2×2 the linear system. The coefficient matrix has elements

$$A_{0,0} = (\varphi_0,\varphi_0) = \int_1^2 1\cdot 1\,dx = 1, \tag{1.29}$$

$$A_{0,1} = (\varphi_0,\varphi_1) = \int_1^2 1\cdot x\,dx = 3/2, \tag{1.30}$$

$$A_{1,0} = A_{0,1} = 3/2, \tag{1.31}$$

$$A_{1,1} = (\varphi_1,\varphi_1) = \int_1^2 x\cdot x\,dx = 7/3\,. \tag{1.32}$$

The corresponding right-hand side is

$$b_1 = (f, \varphi_0) = \int_1^2 (10(x-1)^2 - 1) \cdot 1 \, dx = 7/3, \tag{1.33}$$

$$b_2 = (f, \varphi_1) = \int_1^2 (10(x-1)^2 - 1) \cdot x \, dx = 13/3\,. \tag{1.34}$$

Solving the linear system results in

$$c_0 = -38/3, \quad c_1 = 10, \tag{1.35}$$

and consequently

$$u(x) = 10x - \frac{38}{3}\,. \tag{1.36}$$

Figure 1.1 displays the parabola and its best approximation in the space of all linear functions.

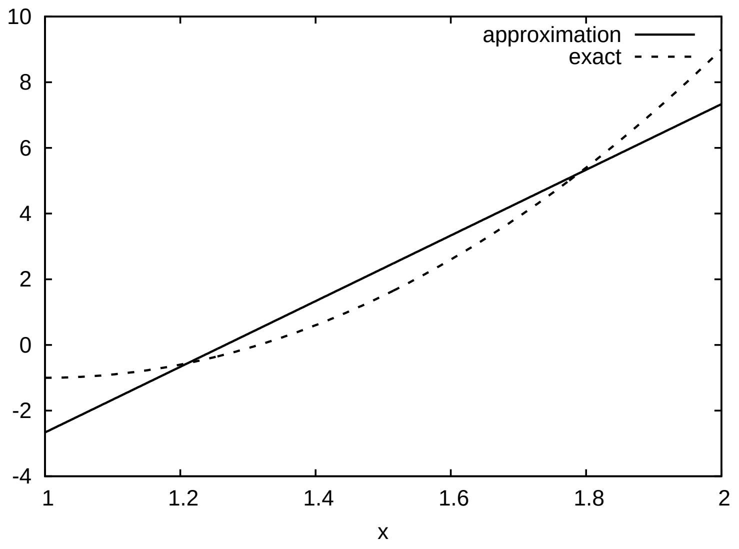


Fig. 1.1. Best approximation of a parabola by a straight line.

1.3.3 Implementation of the Least-Squares Method

The linear system can be computed either symbolically or numerically (a numerical integration rule is needed in the latter case). Here is a function for symbolic computation of the linear system, where $f(x)$ is given as a `sympy` expression `f` (involving the symbol `x`), `phi` is a list of $\varphi_0, \ldots, \varphi_N$, and `Omega` is a 2-tuple/list holding the domain Ω:

```
import sympy as sm

def least_squares(f, phi, Omega):
    N = len(phi) - 1
    A = sm.zeros((N+1, N+1))
    b = sm.zeros((N+1, 1))
    x = sm.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            A[i,j] = sm.integrate(phi[i]*phi[j],
                                  (x, Omega[0], Omega[1]))
            A[j,i] = A[i,j]
        b[i,0] = sm.integrate(phi[i]*f, (x, Omega[0], Omega[1]))
    c = A.LUsolve(b)
    u = 0
    for i in range(len(phi)):
        u += c[i,0]*phi[i]
    return u
```

Observe that we exploit the symmetry of the coefficient matrix: only the upper triangular part is computed. Symbolic integration in `sympy` is often time consuming, and (roughly) halving the work has noticable effect on the waiting time for the function to finish execution.

Comparing the given $f(x)$ and the approximate $u(x)$ visually is done by the following function, which with the aid of `sympy`'s `lambdify` tool converts a `sympy` functional expression to a Python function for numerical computations:

```
def comparison_plot(f, u, Omega, filename='tmp.eps'):
    x = sm.Symbol('x')
    f = sm.lambdify([x], f, modules="numpy")
    u = sm.lambdify([x], u, modules="numpy")
    resolution = 401  # no of points in plot
    xcoor  = linspace(Omega[0], Omega[1], resolution)
    exact  = f(xcoor)
    approx = u(xcoor)
    plot(xcoor, approx)
    hold('on')
    plot(xcoor, exact)
    legend(['approximation', 'exact'])
    savefig(filename)
```

The `modules='numpy'` argument to `lambdify` is important if there are mathematical functions, such as `sin` or `exp` in the symbolic expressions in `f` or `u`, and these mathematical functions are to be used with vector arguments, like `xcoor` above.

Both the `least_squares` and `comparison_plot` are found and coded in the file `approx1D.py`. The forthcoming examples on their use appear in `ex_approx1D.py`.

1.3.4 Perfect Approximation

Let us use the code above to recompute the problem from Chapter 1.3.2 where we want to approximate a parabola. What happens if we add an element x^2

to the basis and test what the best approximation is if V is the space of all parabolic functions? The answer is quickly found by running

```
u = least_squares(f=10*(x-1)**2-1, phi=[1, x, x**2], Omega=[1, 2])
print u
print sm.expand(f)
```

From the output we realize that u becomes identical to f in this case. We may also add many more basis functions, e.g., $\phi_i(x) = x^i$ for $i = 0, \ldots, N = 40$. The output from `least_squares` is $c_i = 0$ for $i > 2$.

The following is in fact a general result: if $f \in V$, the best approximation is $u = f$. The proof is straightforward: if $f \in V$, f can be expanded in terms of the basis functions, $f = \sum_{j=0}^{N} d_j\varphi_j$, for some coefficients $d_0, \ldots, d_N$, and the right-hand side then has entries

$$b_i = (f, \varphi_i) = \sum_{j=0}^{N} d_j(\varphi_j, \varphi_i) = \sum_{j=0}^{N} d_j A_{i,j}\,.$$

The linear system $\sum_j A_{i,j}c_j = b_i$, $i = 0, \ldots, N$, is then

$$\sum_{j=0}^{N} c_j A_{i,j} = \sum_{j=0}^{N} d_j A_{i,j}, \quad i = 0, \ldots, N,$$

which implies that $c_i = d_i$ for $i = 0, \ldots, N$.

1.3.5 Ill-Conditioning

The computational example in Chapter 1.3.4 applies the `least_squares` function which invokes symbolic methods to calculate and solve the linear system. The correct solution $c_0 = 9, c_1 = -20, c_2 = 10, c_i = 0$ for $i \geq 3$ is perfectly recovered.

Suppose we convert the matrix and right-hand side to floating-point arrays and then solve the system using finite-precision arithmetics, which is what one will (almost) always do in real life. This time we get astonishing results! Up to about $N = 7$ we get a solution that is reasonably close to the exact one. Increasing N shows that seriously wrong coefficients are computed. Table 1.1 shows results for $N = 10$ obtained by three different methods:

- Column 2: The matrix and vector are converted to the `sympy.mpmath.fp.matrix` data structure and the `sympy.mpmath.fp.lu_solve` function is used to solve the system.
- Column 3: The matrix and vector are converted to `numpy` arrays with data type `numpy.float32` (single precision floating-point number) and solved by the `numpy.linalg.solve` function.

– Column 4: As column 3, but the data type is `numpy.float64` (double precision floating-point number).

We see from the numbers in the table that double precision performs much better than single precision. Nevertheless, when plotting all these solutions the curves cannot be visually distinguished (!). This means that the approximations look perfect, despite the partially wrong values of the coefficients.

Increasing N to 12 makes the numerical solver in `sympy` report abort with the message: "matrix is numerically singular". A matrix has to be non-singular to be invertible, which is a requirement when solving a linear system. Already when the matrix is close to singular, it is *ill-conditioned*, which here implies that the numerical solution algorithms are sensitive to round-off errors and may produce (very) inaccurate results.

The reason why the coefficient matrix is nearly singular and ill-conditioned is that our basis functions $\varphi_i(x) = x^i$ are nearly linearly dependent for large i. That is, x^i and x^{i+1} are very close when i is about 10 and higher, which is evident by plotting two such functions. Almost linearly dependent basis functions give rise to an ill-conditioned and almost singular matrix. This fact can be illustrated by computing the determinant, which is indeed very close to zero (recall that a zero determinant implies a singular and non-invertible matrix): 10^{-65} for $N = 10$ and 10^{-92} for $N = 12$. Already for $N = 28$ the numerical determinant computation returns a plain zero.

On the other hand, the double precision `numpy` solver do run for $N = 100$, resulting in answers that are not significantly worse than those in Table 1.1, and large powers are associated with small coefficients (e.g., $c_j < 10^{-2}$ for $10 \leq j \leq 20$ and $c < 10^{-5}$ for $j > 20$). Even for $N = 100$ the approximation lies on top of the exact curve in a plot (!).

The conclusion is that visual inspection of the quality of the approximation may not uncover fundamental numerical problems with the computations. However, numerical analysts have studied approximations and ill-conditioning for decades, and it is well known that the basis $\{1, x, x^2, x^3, \ldots, \}$ is a bad basis. The best basis from a matrix conditioning point of view is to have orthogonal functions such that $(\phi_i, \phi_j = 0$ for $i \neq j$. There are many known sets of orthogonal polyomials. The functions used in the finite element methods are almost orthogonal, and this property helps to avoid problems with solving matrix systems[2].

1.3.6 Fourier Series

A set of sine functions is widely used for approximating functions. Let us take

$$V = \text{span}\left\{\sin \pi x, \sin 2\pi x, \ldots, \sin(N+1)\pi x\right\}.$$

[2] Almost orthogonal is helpful, but not enough when it comes to partial differential equations, and ill-conditioning of the coefficient matrix is a theme when solving large-scale finite element systems.

Table 1.1. Solution of the linear system arising from approximating a parabola by functions on the form $u(x) = \sum_{j=0}^{N} c_j x^j$, $N = 10$.

exact	`sympy`	`numpy32`	`numpy64`
9	9.62	5.57	8.98
-20	-23.39	-7.65	-19.93
10	17.74	-4.50	9.96
0	-9.19	4.13	-0.26
0	5.25	2.99	0.72
0	0.18	-1.21	-0.93
0	-2.48	-0.41	0.73
0	1.81	-0.013	-0.36
0	-0.66	0.08	0.11
0	0.12	0.04	-0.02
0	-0.001	-0.02	0.002

That is,

$$\varphi_i(x) = \sin((i+1)\pi x), \quad i = 0, \ldots, N\,.$$

An approximation to the $f(x)$ function from Chapter 1.3.2 can then be computed by the `least_squares` function from Chapter 1.3.3:

```
N = 3
from sympy import sin, pi
phi = [sin(pi*(i+1)*x) for i in range(N+1)]
f = 10*(x-1)**2 - 1
Omega = [0, 1]
u = least_squares(f, phi, Omega)
comparison_plot(f, u, Omega)
```

Figure 1.2a shows the oscillatory approximation. Changing N to 11 improves the approximation considerably, see Figure 1.2b.

The choice of sine functions $\varphi_i(x) = \sin((i+1)\pi x)$ has a great computational advantage: on $\Omega = [0,1]$ these basis functions are *orthogonal*, implying that $A_{i,j} = 0$ if $i \neq j$. This result is realized by trying

```
integrate(sin(j*pi*x)*sin(k*pi*x), x, 0, 1)
```

in *http://wolframalpha.com* (avoid `i` in the integrand as this symbol means the imaginary unit $\sqrt{-1}$). Also by asking *http://wolframalpha.com* about $\int_0^1 \sin^2(j\pi x)dx$, we find it to equal 1/2. With a diagonal matrix we can easily solve for the coefficients by hand:

$$c_i = 2\int_0^1 f(x)\sin((i+1)\pi x)dx, \quad i = 0, \ldots, N, \tag{1.37}$$

which is nothing but the classical formula for the coefficients of the Fourier sine series of $f(x)$ on $[0,1]$. In fact, when V contains the basic functions used in

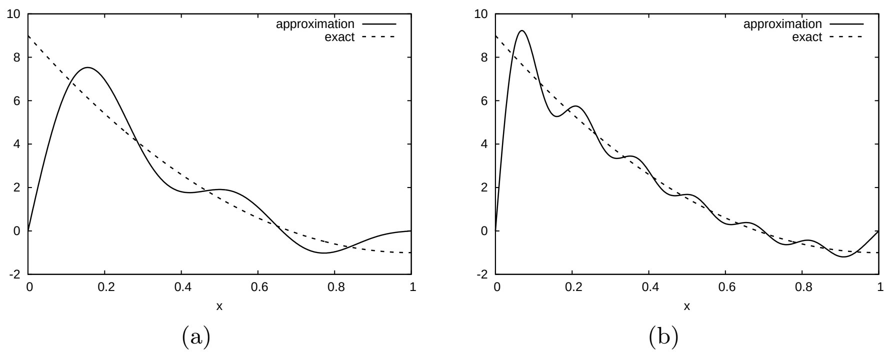


Fig. 1.2. Best approximation of a parabola by a sum of sines: $\sum_{j=0}^{N} c_j \sin((j+1)\pi x)$; (a) $N = 3$; (b) $N = 11$.

a Fourier series expansion, the approximation method derived in Chapter 1.3 results in the classical Fourier series for $f(x)$ (see Exercise 1.7 for details).

For orthogonal basis functions we can make the `least_squares` function (much) more efficient:

```
def least_squares_orth(f, phi, Omega):
    N = len(phi) - 1
    A = [0]*(N+1)
    b = [0]*(N+1)
    x = sm.Symbol('x')
    for i in range(N+1):
        A[i] = sm.integrate(phi[i]**2, (x, Omega[0], Omega[1]))
        b[i] = sm.integrate(phi[i]*f,  (x, Omega[0], Omega[1]))
    c = [b[i]/A[i] for i in range(len(b))]
    u = 0
    for i in range(len(phi)):
        u += c[i]*phi[i]
    return u
```

This function is found in the file `approx1D.py`.

1.3.7 The Collocation Method

The principle of minimizing the distance between u and f is an intuitive way of computing a best approximation $u \in V$ to f. However, there are other attractive approaches as well. One is to demand that $u(x_i) = f(x_i)$ at some selected points x_i, $i = 0, \ldots, N$:

$$u(x_i) = \sum_{j=0}^{N} c_j \varphi_j(x_i) = f(x_i), \quad i = 0, \ldots, N\,. \tag{1.38}$$

This criterion also gives a linear system with $N+1$ unknown coefficients $c_0, \ldots, c_N$:

$$\sum_{j=0}^{N} A_{i,j} c_j = b_i, \quad i = 0, \ldots, N, \tag{1.39}$$

with

$$A_{i,j} = \varphi_j(x_i), \tag{1.40}$$
$$b_i = f(x_i)\,. \tag{1.41}$$

This time the coefficient matrix is not symmetric because $\varphi_j(x_i) \neq \varphi_i(x_j)$ in general. The method is often referred to as a *collocation method* and the x_i points are known as *collocation points*. Others view the approach as an *interpolation method* since some point values of f are given $(f(x_i))$ and we fit a continuous function u that goes through the $f(x_i)$ points. In that case the x_i points are called *interpolation points*.

Given f as a `sympy` symbolic expression `f`, $\varphi_0, \ldots, \varphi_N$ as a list `phi`, and a set of points $x_0, \ldots, x_N$ as a list or array `points`, the following Python function sets up and solves the matrix system for the coefficients $c_0, \ldots, c_N$:

```python
def interpolation(f, phi, points):
    N = len(phi) - 1
    A = sm.zeros((N+1, N+1))
    b = sm.zeros((N+1, 1))
    x = sm.Symbol('x')
    # Turn phi and f into Python functions
    phi = [sm.lambdify([x], phi[i]) for i in range(N+1)]
    f = sm.lambdify([x], f)
    for i in range(N+1):
        for j in range(N+1):
            A[i,j] = phi[j](points[i])
        b[i,0] = f(points[i])
    c = A.LUsolve(b)
    u = 0
    for i in range(len(phi)):
        u += c[i,0]*phi[i](x)
    return u
```

Note that it is convenient to turn the expressions `f` and `phi` into Python functions which can be called with elements of `points` as arguments when building the matrix and the right-hand side. The `interpolation` function is a part of the `approx1D` module.

A nice feature of the interpolation or collocation method method is that it avoids computing integrals. However, one has to decide on the location of the x_i points. A simple, yet common choice, is to distribute them uniformly throughout Ω.

Let us illustrate the interpolation or collocation method by approximating our parabola $f(x) = 10(x-1)^2 - 1$ by a linear function on $\Omega = [1, 2]$, using two collocation points $x_0 = 1 + 1/3$ and $x_1 = 1 + 2/3$:

```
f = 10*(x-1)**2 - 1
phi = [1, x]
Omega = [1, 2]
points = [1 + sm.Rational(1,3), 1 + sm.Rational(2,3)]
u = interpolation(f, phi, points)
comparison_plot(f, u, Omega)
```

The resulting linear system becomes

$$\begin{pmatrix} 1 & 4/3 \\ 1 & 5/3 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 1/9 \\ 31/9 \end{pmatrix}$$

with solution $c_0 = -119/9$ and $c_1 = 10$. Figure 1.3a shows the resulting approximation $u = -119/9 + 10x$. We can easily test other interpolation points, say $x_0 = 1$ and $x_1 = 2$. This changes the line quite significantly, see Figure 1.3b.

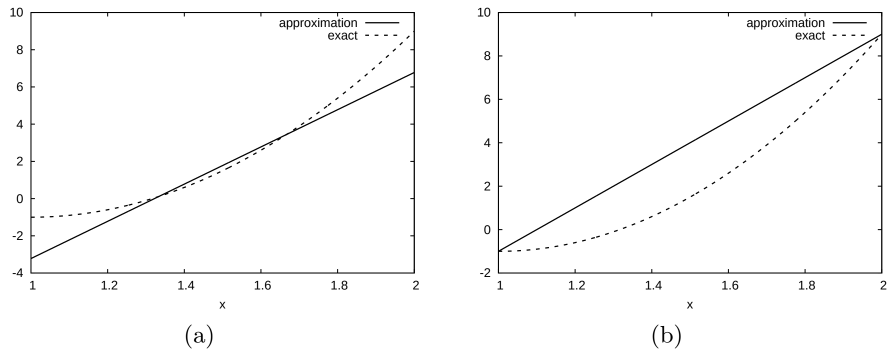


Fig. 1.3. Approximation of a parabola on $[1, 2]$ by linear functions computed by interpolation: (a) $x_0 = 1 + 1/3$, $x_1 = 1 + 2/3$; (b) $x_0 = 1$, $x_1 = 2$.

1.3.8 Lagrange Polynomials

In Chapter 1.3.6 we explain the advantage with having a diagonal matrix: formulas for the coefficients $c_0, \ldots, c_N$ can then be derived by hand. For a interpolation or collocation method a diagonal matrix implies that $\varphi_j(x_i) = 0$ if $i \neq j$. One set of basis functions $\varphi_i(x)$ with this property is the *Lagrange* *interpolating polynomials*, or just *Lagrange polynomials*[3]:

$$\varphi_i(x) = \prod_{j=0,j\neq i}^{N} \frac{x-x_j}{x_i-x_j} = \frac{x-x_0}{x_i-x_0}\cdots\frac{x-x_{i-1}}{x_i-x_{i-1}}\frac{x-x_{i+1}}{x_i-x_{i+1}}\cdots\frac{x-x_N}{x_i-x_N}, \tag{1.42}$$

for $i=0,\ldots,N$. We see from (1.42) that all the φ_i functions are polynomials of degree N which have the property

$$\varphi_i(x_s) = \begin{cases} 1, \, i=s, \\ 0, \, i\neq s, \end{cases} \tag{1.43}$$

when x_s is an interpolation (collocation) point. This property implies that $A_{i,j}=0$ for $i\neq j$ and $A_{i,j}=1$ when $i=j$. The solution of the linear system is them simply

$$c_i = f(x_i), \quad i=0,\ldots,N, \tag{1.44}$$

and

$$u(x) = \sum_{j=0}^{N} f(x_i)\varphi_i(x)\,. \tag{1.45}$$

The following function computes the Lagrange interpolating polynomial $\varphi_i(x)$, given the interpolation points $x_0,\ldots,x_N$ in the list or array `points`:

```
def Lagrange_polynomial(x, i, points):
    p = 1
    for k in range(len(points)):
        if k != i:
            p *= (x - points[k])/(points[i] - points[k])
    return p
```

The next function computes a complete basis using equidistant points throughout Ω:

```
def Lagrange_polynomials_01(x, N):
    if isinstance(x, sm.Symbol):
        h = sm.Rational(1, N-1)
    else:
        h = 1.0/(N-1)
    points = [i*h for i in range(N)]
    phi = [Lagrange_polynomial(x, i, points) for i in range(N)]
    return phi, points
```

When `x` is an `sm.Symbol` object, we let the spacing between the interpolation points, `h`, be a `sympy` rational number for nice end results in the formulas for φ_i. The other case, when `x` is a plain Python `float`, signifies numerical computing, and then we let `h` be a floating-point number. Observe that

[3] Although the functions are named after Lagrange, they were first discovered by Waring in 1779, rediscovered by Euler in 1783, and published by Lagrange in 1795.

the `Lagrange_polynomial` function works equally well in the symbolic and numerical case (think of `x` being an `sm.Symbol` object or a Python `float`). A little interactive session illustrates the difference between symbolic and numerical computing of the basis functions and points:

```
>>> import sympy as sm
>>> x = sm.Symbol('x')
>>> phi, points = Lagrange_polynomials_01(x, N=3)
>>> points
[0, 1/2, 1]
>>> phi
[(1 - x)*(1 - 2*x), 2*x*(2 - 2*x), -x*(1 - 2*x)]

>>> x = 0.5  # numerical computing
>>> phi, points = Lagrange_polynomials_01(x, N=3, symbolic=True)
>>> points
[0.0, 0.5, 1.0]
>>> phi
[-0.0, 1.0, 0.0]
```

The Lagrange polynomials are very much used in finite element methods because of their property (1.43).

Trying out the Lagrange polynomial basis for approximating $f(x) = \sin 2\pi x$ on $\Omega = [0, 1]$ with the least squares and the interpolation techniques can be done by

```
x = sm.Symbol('x')
f = sm.sin(2*sm.pi*x)
phi, points = Lagrange_polynomials_01(x, N)
Omega=[0, 1]
u = least_squares(f, phi, Omega)
comparison_plot(f, u, Omega)
u = interpolation(f, phi, points)
comparison_plot(f, u, Omega)
```

Figure 1.3 shows the results. There is little difference between the least squares and the interpolation technique. Increasing N gives visually better approximations.

The next example concerns interpolating $f(x) = |1 - 2x|$ on $\Omega = [0, 1]$ using Lagrange polynomials. Figure 1.5 shows a peculiar effect: the approximation starts to oscillate more and more as N grows. This numerical artifact is not surprising when looking at the individual Lagrange polynomials: Figure 1.6 shows two such polynomials of degree 11, and it is clear that the basis functions oscillate significantly. The reason is simple, since we force the functions to be 1 at one point and 0 at many other points. A polynomial of high degree is then forced to oscillate between these points. The oscillations are particularly severe at the boundary. The phenomenon is named *Runge's phenomenon* and you can read a more detailed explanation on Wikipedia.

The oscillations can be reduced by a more clever choice of interpolation points, called the *Chebyshev nodes*:

$$x_i = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos\left(\frac{2i+1}{2(N+1)}pi\right), \quad i = 0\ldots, N, \tag{1.46}$$

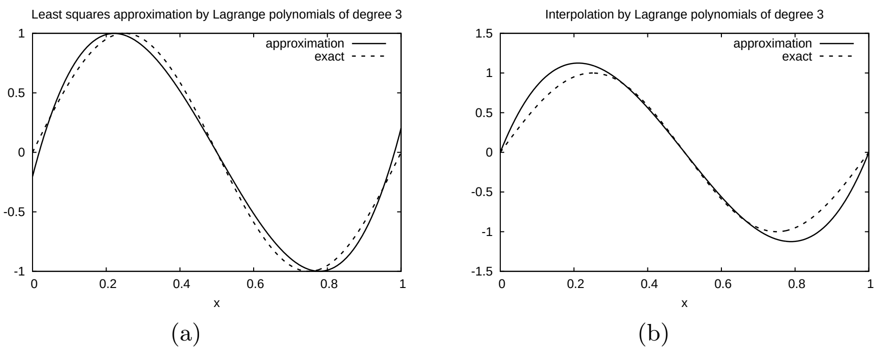


Fig. 1.4. Approximation of a sine function by Lagrange interpolating polynomials of degree 4. (a) least squares; (b) interpolation (collocation).

on the interval $\Omega = [a, b]$. Here is a flexible version of the `Lagrange_polynomials_01` function above, valid for any interval $\Omega = [a, b]$ and with the possibility to generate both uniformly distributed points and Chebyshev nodes:

```
def Lagrange_polynomials(x, N, Omega, point_distribution='uniform'):
    if point_distribution == 'uniform':
        if isinstance(x, sm.Symbol):
            h = sm.Rational(Omega[1] - Omega[0], N)
        else:
            h = (Omega[1] - Omega[0])/float(N)
        points = [Omega[0] + i*h for i in range(N+1)]
    elif point_distribution == 'Chebyshev':
        points = Chebyshev_nodes(Omega[0], Omega[1], N)
    phi = [Lagrange_polynomial(x, i, points) for i in range(N+1)]
    return phi, points

def Chebyshev_nodes(a, b, N):
    from math import cos, pi
    return [0.5*(a+b) + 0.5*(b-a)*cos(float(2*i+1)/(2*(N+1))*pi) \
            for i in range(N+1)]
```

All the functions computing Lagrange polynomials listed above are found in the module file `Lagrange.py`. Figure 1.7 shows the improvement of using Chebyshev nodes (compared with Figure 1.5).

Another cure for undesired oscillation of higher-degree interpolating polynomials is to use lower-degree Lagrange polynomials on many small patches of the domain, which is the idea persued in the finite elemenet method. For instance, linear Lagrange polynomials on $[0, 1/2]$ and $[1/2, 1]$ would yield a perfect approximation to $f(x) = |1 - 2x|$ on $\Omega = [0, 1]$ since f is piecewise linear.

Unfortunately, `sympy` has problems integrating the $f(x) = |1 - 2x|$ function times a polynomial. Other choices of $f(x)$ can also make the symbolic integration fail. Therefore, we should extend the `least_squares` function

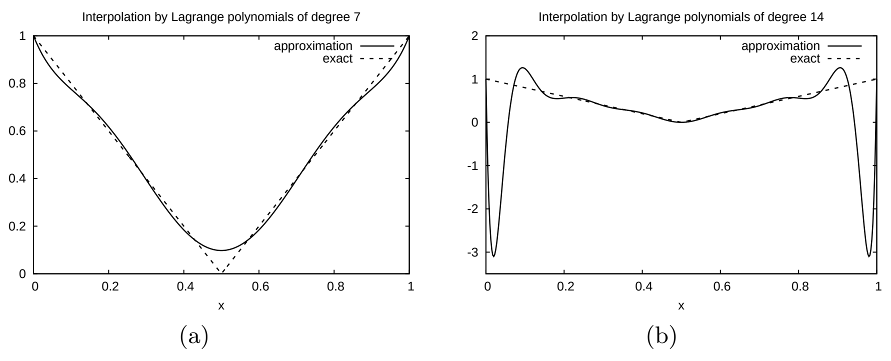


Fig. 1.5. Interpolation of an absolute value function by Lagrange polynomials and uniformly distributed interpolation points: (a) degree 7; (b) degree 14.

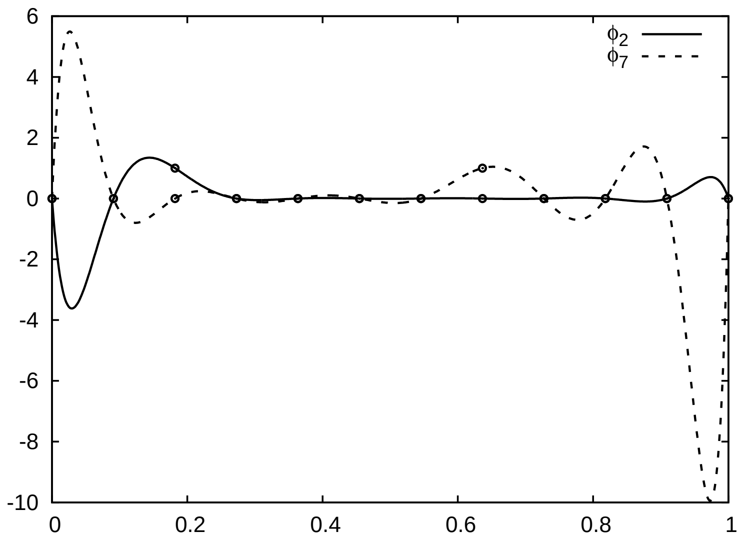
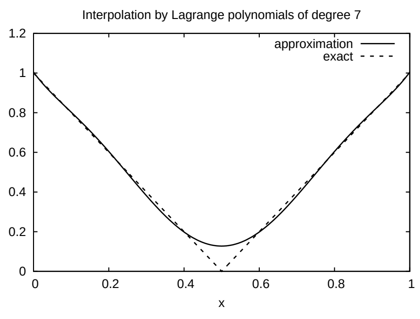
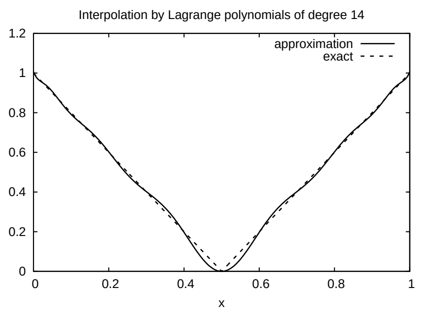


Fig. 1.6. Illustration of the oscillatory behavior of two Lagrange polynomials for $N = 12$ uniformly spaced points (marked by circles).



(a)



(b)

Fig. 1.7. Interpolation of an absolute value function by Lagrange polynomials and Chebyshev nodes as interpolation points (a) degree 7; (b) degree 14.

such that it falls back on numerical integration if the symbolic integration is uncessful. In the latter case, the returned value from `sympy`'s `integrate` function is an object of type `Integral`. We can test on this type and utilize the `mpmath` module in `sympy` to perform numerical integration of high precision. Here is the code:

```
def least_squares(f, phi, Omega):
    N = len(phi) - 1
    A = sm.zeros((N+1, N+1))
    b = sm.zeros((N+1, 1))
    x = sm.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            integrand = phi[i]*phi[j]
            I = sm.integrate(integrand, (x, Omega[0], Omega[1]))
            if isinstance(I, sm.Integral):
                # Could not integrate symbolically, fallback
                # on numerical integration with mpmath.quad
                integrand = sm.lambdify([x], integrand)
                I = sm.mpmath.quad(integrand, [Omega[0], Omega[1]])
            A[i,j] = A[j,i] = I
        integrand = phi[i]*f
        I = sm.integrate(integrand, (x, Omega[0], Omega[1]))
        if isinstance(I, sm.Integral):
            integrand = sm.lambdify([x], integrand)
            I = sm.mpmath.quad(integrand, [Omega[0], Omega[1]])
        b[i,0] = I
    c = A.LUsolve(b)
    u = 0
    for i in range(len(phi)):
        u += c[i,0]*phi[i]
    return u
```

1.4 Finite Element Basis Functions

The basis functions in Chapter 1.3 are in general nonzero on the whole domain Ω. We shall now turn the attention to basis functions that have *compact support*, meaning that they are nonzero on only a small portion of Ω. Moreover, we shall restrict the functions to be *piecewise polynomials*. This means that the domain is split into subdomains and the function is a polynomial on one or more subdomains. At the boundaries between subdomains one normally forces only continuity of the function so that when connecting two polynomials from two subdomains, the derivative usually becomes discontinuous. These type of basis functions are fundamental in the *finite element method*.

1.4.1 Elements and Nodes

Let us divide the interval Ω on which f and u are defined into non-overlapping subintervals $\Omega^{(e)}$, $e = 0, \ldots, n_e$:

$$\Omega = \Omega^{(0)} \cup \cdots \cup \Omega^{(M)} . \tag{1.47}$$

We shall refer to $\Omega^{(e)}$ as an *element*, having number e, since *element* is the common term for such subintervals in the finite element method. On each element we introduce a set of points called *nodes*. For now we assume that the nodes are uniformly spaced throughout the element and that the boundary points of the elements are also nodes. The nodes are given numbers both at the element level and in the global domain, referred to as *local* and *global* node numerings, respectively.

Nodes and elements uniquely define a *finite element mesh*, which is our discrete representation of the domain in the computations. A common special case is that of a *uniformly partitioned mesh* where each element has the same length and the distance between nodes is constant.

For example, on $\Omega = [0, 1]$ we may introduce two elements, $\Omega^{(0)} = [0, 0.4]$ and $\Omega^{(1)} = [0.4, 1]$. Furthermore, let us introduce three nodes per element, equally spaced within each element. The three nodes in element number 1 are $x_0 = 0$, $x_1 = 0.2$, and $x_2 = 0.4$. The local and global node numbers are here equal. In element number 2, we have the local nodes $x_0 = 0.4$, $x_1 = 0.7$, and $x_2 = 1$ and the corresponding global nodes $x_2 = 0.4$, $x_3 = 0.7$, and $x_4 = 1$. Note that the global node $x_2 = 0.4$ is shared by the two elements.

For the purpose of implementation, we introduce two lists or arrays: `nodes` for storing the coordinates of the nodes, with the global node numbers as indices, and `elements` for holding the global node numbers in each element, with the local node numbers as indices. The `nodes` and `elements` lists for the sample mesh above take the form

```
nodes = [0, 0.2, 0.4, 0.7, 1]
elements = [[0, 1, 2], [2, 3, 4]]
```

Looking up the coordinate of local node number 2 in element 1 is here done by `nodes[elements[1][2]]`.

1.4.2 The Basis Functions

Standard finite element basis functions are now defined as follows. Let i be the global node number corresponding to local node r in element number e.

- If local node number r is not on the boundary of the element, take $\varphi_i(x)$ to be the Lagrange polynomial that is 1 at the local node number r and zero at all other nodes in the element. Let $\varphi_i = 0$ on all other elements.
- If local node number r is on the boundary of the element, let φ_i be made up of the Lagrange polynomial that is 1 at this node in element number e and its neighboring element. On all other elements, $\varphi_i = 0$.

Note that when we refer to a Lagrange polynomial on an element, this polynomial is thought to vanish on all other elements.

The construction of basis functions according to the principles above lead to two important properties of $\varphi_i(x)$. First,

$$\varphi_i(x_j) = \begin{cases} 1, \ i = j, \\ 0, \ i \neq j, \end{cases} \tag{1.48}$$

when x_j is a node in the mesh with global node number j. This property implies a convenient interpretation of c_j:

$$u(x_i) = \sum_{j=0}^{N} c_j\varphi_j(x_i) = c_i\varphi_i(x_i) = c_i\,. \tag{1.49}$$

That is, c_i is the value of u at node i (x_i).

Second, $\varphi_i(x)$ is mostly zero throughout the domain: $\varphi_i(x) \neq 0$ only on those elements that contain global node i. In particular, $\varphi_i(x)\varphi_j(x) \neq 0$ if and only if i and j are global node numbers in the same element. Since $A_{i,j}$ is the integral of $\varphi_i\varphi_j$ it means that most of the elements in the coefficient matrix will be zero. We will come back to these properties and use them actively later.

We let each element have $d + 1$ nodes, resulting in local Lagrange polynomials of degree d. It is not a requirement to have the same d value in each element, but for now we will assume so.

Figure 1.8 illustrates how piecewise quadratic basis functions can look like ($d = 2$). We work with the domain $\Omega = [0, 1]$ divided into four equal-sized elements, each having three nodes. The `nodes` and `elements` lists in this particular example become

```
nodes = [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.0]
elements = [[0, 1, 2], [2, 3, 4], [4, 5, 6], [6, 7, 8]]
```

Nodes are marked with circles on the x axis in the figure, and element boundaries are marked with small vertical lines.

Consider element number 1, $\Omega^{(1)} = [0.25, 0.5]$, with local nodes 0, 1, and 2 corresponding to global nodes 2, 3, and 4. The coordinates of these nodes are 0.25, 0.375, and 0.5, respectively. We define three Lagrange polynomials on this element. The one that is 1 at local node 1 ($x = 0.375$) becomes the global basis function $\varphi_3(x)$ over this element, with $\varphi_3(x) = 0$ outside the element. The other global functions associated with internal nodes, φ_1, φ_5, and φ_7, are all of the same shape as φ_3. The basis function $\varphi_2(x)$, corresponding to a node on the boundary of element 0 and 1, is made up of two pieces: (i) the Lagrange polynomial on element 1 that is 1 at local node 0 (global node 2) and zero at all other nodes in element 1, and (ii) the Lagrange polynomial on element 1 that is 1 at local node 2 (global node 2) and zero at all other nodes in element 0. Outside the elements that share global node 2, $\varphi_2(x) = 0$. The same reasoning is applied to the construction of $\varphi_4(x)$ and $\varphi_6(x)$.

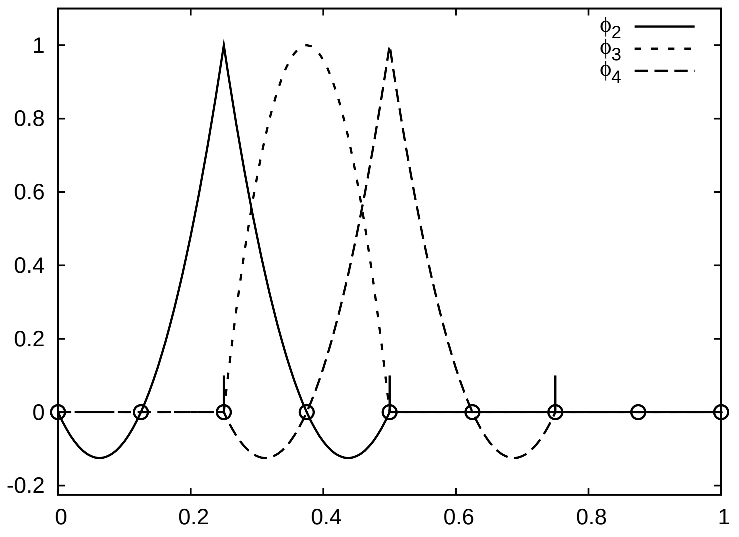


Fig. 1.8. Illustration of the piecewise quadratic basis functions associated with nodes in element $\Omega^{(1)}$.

Figure 1.9 shows the construction of piecewise linear basis functions ($d = 1$). Also here we have four elements on $\Omega = [0, 1]$. Consider the element $\Omega^{(1)} = [0.25, 0.5]$. Now there are no internal nodes in the elements so that all basis functions are associated with nodes at the element boundaries and hence made up of two Lagrange polynomials from neighboring elements. For example, $\varphi_1(x)$ results from (i) the Lagrange polynomial in element 0 that is 1 at local node 1 and 0 at local node 0, and (ii) the Lagrange polynomial in element 1 that is 1 at local node 0 and 0 at local node 1. The other basis functions are constructed similarly.

Explicit mathematical formulas are needed for $\varphi_i(x)$ in computations. In the piecewise linear case, one can show that

$$\varphi_i(x) = \begin{cases} 0, & x < x_{i-1}, \\ (x - x_{i-1})/(x_i - x_{i-1}), & x_{i-1} \le x < x_i, \\ 1 - (x - x_i)/(x_{i+1} - x_i), & x_i \le x < x_{i+1}, \\ 0, & x \ge x_{i+1}\,. \end{cases} \tag{1.50}$$

Here, x_j, $j = i-1, i, i+1$, denotes the coordinate of node j. For elements of equal length h the formulas can be simplified to

$$\varphi_i(x) = \begin{cases} 0, & x < x_{i-1}, \\ (x - x_{i-1})/h, & x_{i-1} \le x < x_i, \\ 1 - (x - x_i)/h, & x_i \le x < x_{i+1}, \\ 0, & x \ge x_{i+1} \end{cases} \tag{1.51}$$

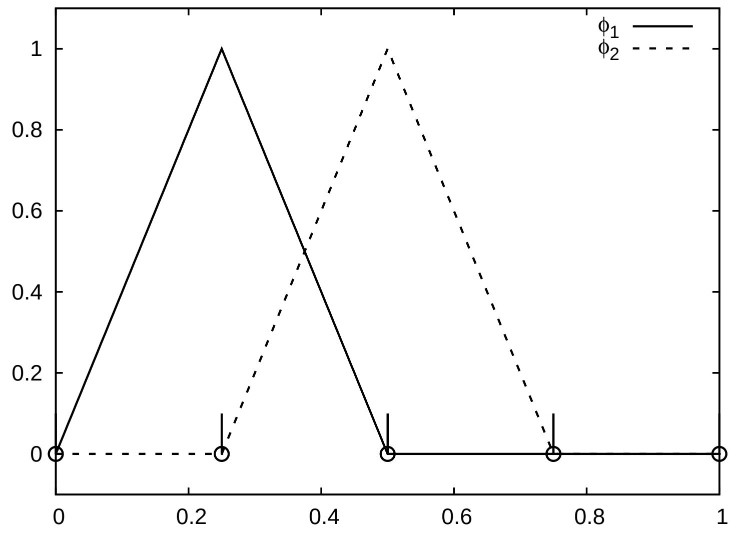


Fig. 1.9. Illustration of the piecewise linear basis functions associated with nodes in element $\Omega^{(1)}$.

Piecewise cubic basis functions can be defined by introducing four nodes per element. Figure 1.10 shows examples on $\varphi_i(x)$, $i = 3, 4, 5, 6$, associated with element number 1. Note that φ_4 and φ_5 are nonzero on element number 1, while φ_3 and φ_6 are made up of Lagrange polynomials on two neighboring elements.

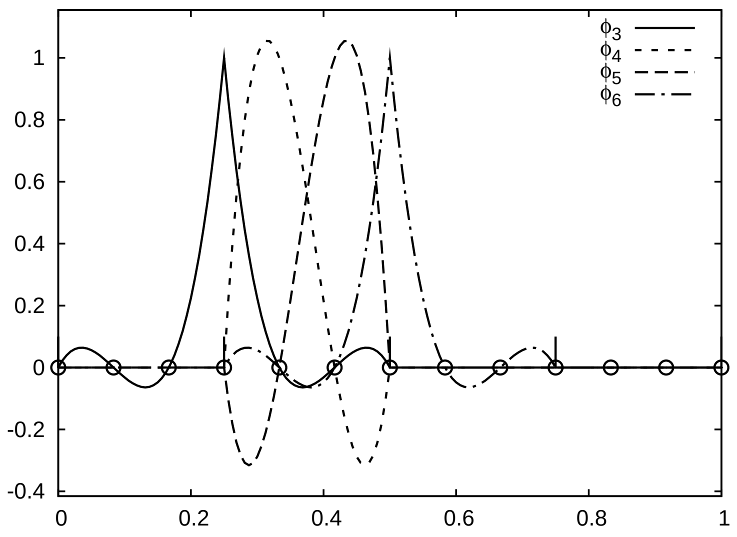


Fig. 1.10. Illustration of the piecewise cubic basis functions associated with nodes in element $\Omega^{(1)}$.

We see that all the piecewise linear basis functions have the same "hat" shape. They are naturally referred to as *hat functions* (also called *chapau functions*). The piecewise quadratic functions in Figure 1.8 are seen to be of two types. "Rounded hats" associated with internal nodes in the elements and some more "sombrero" shaped hats associated with element boundary nodes. Higher-order basis functions also have hat-like shapes, but the functions have pronounced oscillations in addition, as illustrated in Figure 1.10.

A common terminology is to speak about *linear elements* as elements with two local nodes and where the basis functions are piecewise linear. Similarly, *quadratic elements* and *cubic elements* refer to piecewise quadratic or cubic functions over elements with three or four local nodes, respectively. Alternative names, frequently used later, are P1 elements for linear elements, P2 for quadratic elements, and so forth (Pd signifies degree d of the polynomial basis functions).

1.4.3 Calculating the Linear System

The elements in the coefficient matrix and right-hand side, given by the formulas (1.25) and (1.26), will now be calculated for piecewise polynomial basis functions. Consider P1 (piecewise linear) elements. Nodes and elements numbered consequtively from left to right imply the elements

$$\Omega^{(i)} = [x_i, x_{i+1}] = [ih, (i+1)h], \quad i = 0, \ldots, N-1\,. \tag{1.52}$$

We have in this case N elements and $N+1$ nodes, and $\Omega = [x_0, x_N]$. The formula for $\varphi_i(x)$ is given by (1.51) and a graphical illustration is provided in Figure 1.9. First we see from Figure 1.9 that $\varphi_i(x)\varphi_j(x) \neq 0$ if and only if $j = i-1$, $j = i$, or $j = i+1$, or with other words, if and only if i and j are nodes in the same element. Otherwise, φ_i and φ_j are too distant to have an overlap and consequently a nonzero product.

The element $A_{i,i-1}$ in the coefficient matrix can be calculated as

$$\int_\Omega \varphi_i\varphi_{i-1} dx = \int_{x_{i-1}}^{x_i} \left(1 - \frac{x - x_{i-1}}{h}\right) \frac{x - x_i}{h} dx = \frac{h}{6}\,.$$

It turns out that $A_{i,i+1} = h/6$ as well and that $A_{i,i} = 2h/3$. The numbers are modified for $i = 0$ and $i = N$: $A_{0,0} = h/3$ and $A_{N,N} = h/3$. The general formula for the right-hand side becomes

$$b_i = \int_{x_{i-1}}^{x_i} \frac{x - x_{i-1}}{h} f(x)dx + \int_{x_i}^{x_{i+1}} \left(1 - \frac{x - x_i}{h}\right) f(x)dx\,. \tag{1.53}$$

With two equal-sized elements in $\Omega = [0,1]$ and $f(x) = x(1-x)$, one gets

$$A = \frac{h}{6}\begin{pmatrix} 2\;1\;0 \\ 1\;4\;1 \\ 0\;1\;2 \end{pmatrix}, \quad b = \frac{h^2}{12}\begin{pmatrix} 2 - 3h \\ 12 - 14h \\ 10 - 17h \end{pmatrix}.$$

The solution becomes

$$c_0 = \frac{h^2}{6}, \quad c_1 = h - \frac{5}{6}h^2, \quad c_2 = 2h - \frac{23}{6}h^2\,.$$

The resulting function

$$u(x) = c_0\varphi_0(x) + c_1\varphi_1(x) + c_2\varphi_2(x)$$

is displayed in Figure 1.11a. Doubling the number of elements to four results in the improved approximation in Figure 1.11b.

The integrals are naturally split into integrals over individual elements since the formulas change with the elements. This idea of splitting the integral is fundamental in all practical implementations of the finite element method.

1.4.4 Assembly of Elementwise Computations

Let us split the integral over Ω into a sum of contributions from each element:

$$A_{i,j} = \int_\Omega \varphi_i\varphi_j dx = \sum_e A_{i,j}^{(e)}, \quad A_{i,j}^{(e)} = \int_{\Omega^{(e)}} \varphi_i\varphi_j dx\,. \tag{1.54}$$

Now, $A_{i,j}^{(e)} \neq 0$ if and only if i and j are nodes in element e. Introduce $i = q(e,r)$ as the mapping of local node number r in element e to the global node

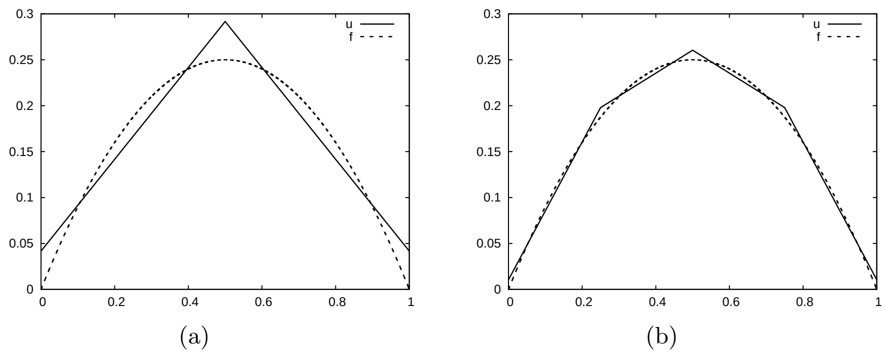


Fig. 1.11. Least-squares approximation of $f(x) = x(1-x)$ on $\Omega = [0, 1]$ by finite elements with piecewise linear basis functions: (a) two elements; (b) four elements.

number i. This is just a short notation for the expression `i=elements[e][r]` in a program. Let r and s be the local node numbers corresponding to the global node numbers $i = q(e, r)$ and $j = q(e, s)$. With d nodes per element, all the nonzero elements in $A^{(e)}_{i,j}$ arise from the integrals involving basis functions with indices corresponding to the global node numbers in element number e:

$$\int_{\Omega^{(e)}} \varphi_{q(e,r)}\varphi_{q(e,s)}dx, \quad r, s = 0, \ldots, d\,.$$

These contributions can be collected in a $(d + 1) \times (d + 1)$ matrix known as the *element matrix*. We introduce the notation

$$\tilde{A}^{(e)} = \{\tilde{A}^{(e)}_{r,s}\}, \quad r, s = 0, \ldots, d,$$

for the element matrix. Given the numbers $\tilde{A}^{(e)}_{r,s}$, we should acoording to (1.54) add the contributions to the global coefficient matrix by

$$A_{q(e,r),q(e,s)} := A_{q(e,r),q(e,s)} + \tilde{A}^{(e)}_{r,s}, \quad r, s = 0, \ldots, d\,. \tag{1.55}$$

This process of adding in elementwise contributions to the global matrix is called *finite element assembly* or simply *assembly*. Figure 1.12 gives a picture how element matrices for elements with two nodes are added into the global matrix.

The right-hand side of the linear system is also computed elementwise:

$$b_i = \int_\Omega \varphi_i\varphi_j dx = \sum_e b^{(e)}_i, \quad b^{(e)}_i = \int_{\Omega^{(e)}} f(x)\varphi_i(x)dx\,. \tag{1.56}$$

We observe that $b^{(e)}_i \neq 0$ if and only if global node i is a node in element e. With d nodes per element we can collect the $d+1$ nonzero contributions $b^{(e)}_i$,

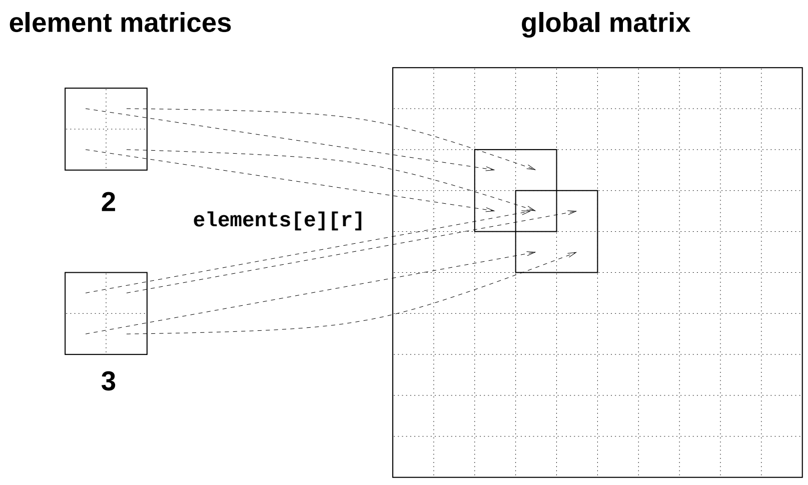


Fig. 1.12. Illustration of matrix assembly: element matrices, corresponding to element 2 and 3, are added to the global matrix. The global node numbers and elements are here numbered from left to right in the domain Ω.

for $i = q(e, r)$, $r = 0, \ldots, d$, in an *element vector*

$$\tilde{b}_r^{(e)} = \{\tilde{b}_r^{(e)}\}, \quad r = 0, \ldots, d\,.$$

These contributions are added to the global right-hand side by an assembly process similar to that for the element matrices:

$$b_{q(e,r)} := b_{q(e,r)} + \tilde{b}_r^{(e)}, \quad r, s = 0, \ldots, d\,. \tag{1.57}$$

1.4.5 Mapping to a Reference Element

Instead of computing the integrals

$$\tilde{A}_{r,s}^{(e)} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}(x)\varphi_{q(e,s)}(x)dx$$

over some element[4] $\Omega^{(e)} = [x_L, x_R]$, it is convenient to map the element domain $[x_L, x_R]$ to a standardized reference element domain $[-1, 1]$. Let X be the coordinate in the reference element. A linear or *affine mapping* from X to x reads

$$x = \frac{1}{2}(x_L + x_R) + \frac{1}{2}(x_R - x_L)X\,. \tag{1.58}$$

[4] We now introduce x_L and x_R as the left and right boundary points of an element. With a natural numbering of noes and elements from left to right through the domain, $x_L = x_e$ and $x_R = x_{e+1}$.

Integrating on the reference element is a matter of just changing the integration variable from x to X. Let

$$\tilde{\varphi}_r(X) = \varphi_{q(e,r)}(x(X)) \tag{1.59}$$

be the basis function associated with local node number r in the reference element. The integral transformation reads

$$\tilde{A}^{(e)}_{r,s} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}(x)\varphi_{q(e,s)}(x)dx = \int_{-1}^{1} \tilde{\varphi}_r(X)\tilde{\varphi}_s(X)\frac{dx}{dX}dX\,. \tag{1.60}$$

The stretch factor dx/dX between the x and X coordinates becomes the determinant of the Jacobian matrix of the mapping between the coordinate systems in 2D and 3D. To obtain a uniform notation for 1D, 2D, and 3D problems we therefore replace dx/dX by $\det J$ already now. In 1D, $\det J = dx/dX = h/2$, h being the length of the element ($h = x_R - x_L$ when $\Omega_e = [x_L, x_R]$). The integration over the reference element is then written as

$$\tilde{A}^{(e)}_{r,s} = \int_{-1}^{1} \tilde{\varphi}_r(X)\tilde{\varphi}_s(X)\det J\,dX\,. \tag{1.61}$$

The corresponding formula for the element vector entries becomes

$$\tilde{b}^{(e)}_{r} = \int_{\Omega^{(e)}} f(x)\varphi_{q(e,r)}(x)dx = \int_{-1}^{1} f(x(X))\tilde{\varphi}_r(X)\det J\,dX\,. \tag{1.62}$$

Since we from now on will work in the reference element, we need explicit mathematical formulas for the basis functions $\varphi_i(x)$ in the reference element only, i.e., we only need to specify formulas for $\tilde{\varphi}_r(X)$. These functions are simply the Lagrange polynomials defined through the local nodes in the reference element. For $d = 1$ and two nodes per element, we have the linear Lagrange polynomials

$$\tilde{\varphi}_0(X) = \frac{1}{2}(1 - X) \tag{1.63}$$

$$\tilde{\varphi}_1(X) = \frac{1}{2}(1 + X) \tag{1.64}$$

Quadratic polynomials, $d = 2$, have the formulas

$$\tilde{\varphi}_0(X) = \frac{1}{2}(X - 1)X \tag{1.65}$$

$$\tilde{\varphi}_1(X) = 1 - X^2 \tag{1.66}$$

$$\tilde{\varphi}_2(X) = \frac{1}{2}(X + 1)X \tag{1.67}$$

In general,

$$\tilde{\varphi}_r(x) = \prod_{s=0}^{d} \frac{X - X_{(s)}}{X_{(r)} - X_{(s)}}, \tag{1.68}$$

where $X_{(0)}, \ldots, X_{(d)}$ are the coordinates of the local nodes in the reference element. These are normally uniformly spaced: $X_{(r)} = -1 + 2r/d$, $r = 0, \ldots, d$.

1.4.6 Implementation of Reference Element Integration

To illustrate the concepts from the previous section, we now consider calculation of the element matrix and vector for a specific choice of d and $f(x)$. A simple choice is $d = 1$ and $f(x) = x(1 - x)$ on $\Omega = [0, 1]$. We have

$$\tilde{A}^{(e)}_{r,s} = \int_{-1}^{1} \tilde{\varphi}_r(X)\tilde{\varphi}_s(X)\det J\,dX, \tag{1.69}$$

$$\tilde{b}^{(e)}_{r} = \int_{-1}^{1} f(x(X))\tilde{\varphi}_r(X)\det J\,dX\,. \tag{1.70}$$

1.4.7 Implementation of Symbolic Integration

Although it may be instructive to compute $\tilde{A}^{(e)}_{r,s}$ and $\tilde{b}^{(e)}_{r}$ by hand, it is also natural to make use of `sympy` and automate the integrations. Appropriate functions for this purpose are found in the module `fe_approx1D.py`. First we need a Python function for defining $\tilde{\varphi}_r(X)$ in terms of a Lagrange polynomial of degree `d`:

```
import sympy as sm
import numpy as np

def phi_r(r, X, d):
    if isinstance(X, sm.Symbol):
        h = sm.Rational(1, d)
        nodes = [2*i*h - 1 for i in range(d+1)]
    else:
        # assume X is numeric: use floats for nodes
        nodes = np.linspace(-1, 1, d+1)
    return Lagrange_polynomial(X, r, nodes)

def Lagrange_polynomial(x, i, points):
    p = 1
    for k in range(len(points)):
        if k != i:
            p *= (x - points[k])/(points[i] - points[k])
    return p
```

Observe how we construct the `phi_r` function to be a symbolic expression for $\tilde{\varphi}_r(X)$ if `X` is a `sympy Symbol` object. Otherwise, we assume that `X` is a `float` object and compute the corresponding floating-point value of $\tilde{\varphi}_r(X)$. The `Lagrange_polynomial` function, copied here from Chapter 1.3.6, works with both symbolic and numeric `x` and `points` variables.

The complete basis $\tilde{\varphi}_0(X), \ldots, \tilde{\varphi}_d(X)$ on the reference element is constructed by

```
def basis(d=1):
    X = sm.Symbol('X')
    phi = [phi_r(r, X, d) for r in range(d+1)]
    return phi
```

Now we are in a position to write the function for computing the element matrix:

```
def element_matrix(phi, Omega_e, symbolic=True):
    n = len(phi)
    A_e = sm.zeros((n, n))
    X = sm.Symbol('X')
    if symbolic:
        h = sm.Symbol('h')
    else:
        h = Omega_e[1] - Omega_e[0]
    detJ = h/2  # dx/dX
    for r in range(n):
        for s in range(r, n):
            A_e[r,s] = sm.integrate(phi[r]*phi[s]*detJ, (X, -1, 1))
            A_e[s,r] = A_e[r,s]
    return A_e
```

In the symbolic case (`symbolic` is `True`), we introduce the element length as a symbol `h` in the computations. Otherwise, the real numerical value of the element interval `Omega_e` is used and the final matrix elements are numbers, not symbols. This functionality can be demonstrated:

```
>>> phi = basis(d=1)
>>> phi
[1/2 - X/2, 1/2 + X/2]
>>> element_matrix(phi, Omega_e=[0.1, 0.2], symbolic=True)
[h/3, h/6]
[h/6, h/3]
>>> element_matrix(phi, Omega_e=[0.1, 0.2], symbolic=False)
[0.0333333333333333, 0.0166666666666667]
[0.0166666666666667, 0.0333333333333333]
```

The computation of the element vector is done by a similar procedure:

```
def element_vector(f, phi, Omega_e, symbolic=True):
    n = len(phi)
    b_e = sm.zeros((n, 1))
    # Make f a function of X
    X = sm.Symbol('X')
    if symbolic:
        h = sm.Symbol('h')
    else:
        h = Omega_e[1] - Omega_e[0]
    x = (Omega_e[0] + Omega_e[1])/2 + h/2*X  # mapping
    f = f.subs('x', x)
    detJ = h/2  # dx/dX
    for r in range(n):
        b_e[r] = sm.integrate(f*phi[r]*detJ, (X, -1, 1))
    return b_e
```

Here we need to replace `x` in the expression for `f` by `X`, using the mapping formula. Realize that the previous code segments also document the step by step procedyre that is needed to perform calculations by hand.

The integration in the element matrix function involves only products of polynomials, which `sympy` can easily deal with, but for the right-hand side `sympy` may face difficulties with certain types of expressions `f`. It may therefore be wise to introduce a fallback on numerical integration (see page 22):

```
        I = sm.integrate(f*phi[r]*detJ, (X, -1, 1))
        if isinstance(I, sm.Integral):
            h = Omega_e[1] - Omega_e[0]  # Ensure h is numerical
            detJ = h/2
            integrand = sm.lambdify([X], f*phi[r]*detJ)
            I = sm.mpmath.quad(integrand, [-1, 1])
        b_e[r] = I
```

Successful numerical integration requires that the symbolic integrand is converted to a plain Python function (`integrand`) and that the element length `h` is indeed a real number.

1.4.8 Implementation of Linear System Assembly and Solution

The complete algorithm for computing and assembling the elementwise contributions takes the following form

```
def assemble(nodes, elements, phi, f, symbolic=True):
    n_n, n_e = len(nodes), len(elements)
    A = sm.zeros((n_n, n_n))
    b = sm.zeros((n_n, 1))
    for e in range(n_e):
        Omega_e = [nodes[elements[e][0]], nodes[elements[e][-1]]]

        A_e = element_matrix(phi, Omega_e, symbolic)
        b_e = element_vector(f, phi, Omega_e, symbolic)

        for r in range(len(elements[e])):
            for s in range(len(elements[e])):
                A[elements[e][r],elements[e][s]] += A_e[r,s]
            b[elements[e][r]] += b_e[r]
    return A, b
```

The `nodes` and `elements` variables represent the finite element mesh as explained earlier.

Given the coefficient matrix `A` and the right-hand side `b`, we can compute the cofficients $c_0, \ldots, c_N$ in the expansion $u(x) = \sum_j c_j \varphi_j$ as the solution vector `c` of the linear system:

```
c = A.LUsolve(b)
```

Note that `A` and `b` are `sympy` matrices and that the solution procedure implied by `A.LUsolve` is symbolic. This means that the functions above are suited only for small problems (small N values). Normally, the symbolic integration will be more time consuming than the symbolic solution of the linear system.

Example on Computing Approximations. We can exemplify the use of `assemble` on the computational case from Chapter 1.4.3 with two elements with linear basis functions on the domain $\Omega = [0, 1]$. Let us first work with a symbolic element length:

```
>>> h, x = sm.symbols('h x')
>>> nodes = [0, h, 2*h]
>>> elements = [[0, 1], [1, 2]]
>>> phi = basis(d=1)
>>> f = x*(1-x)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=True)
>>> A
[h/3,   h/6,   0]
[h/6, 2*h/3, h/6]
[  0,   h/6, h/3]
>>> b
[     h**2/6 - h**3/12]
[      h**2 - 7*h**3/6]
[5*h**2/6 - 17*h**3/12]
>>> c = A.LUsolve(b)
>>> c
[                           h**2/6]
[12*(7*h**2/12 - 35*h**3/72)/(7*h)]
[  7*(4*h**2/7 - 23*h**3/21)/(2*h)]
```

We may, for comparison, compute the `c` vector that corresponds to just interpolating `f` at the node points:

```
>>> fn = sm.lambdify([x], f)
>>> [fn(xc) for xc in nodes]
[0, h*(1 - h), 2*h*(1 - 2*h)]
```

The corresponding numerical computations, as done by `sympy` and still based on symbolic integration, goes as follows:

```
>>> nodes = [0, 0.5, 1]
>>> elements = [[0, 1], [1, 2]]
>>> phi = basis(d=1)
>>> x = sm.Symbol('x')
>>> f = x*(1-x)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=False)
>>> A
[ 0.166666666666667, 0.0833333333333333,                  0]
[0.0833333333333333,  0.333333333333333, 0.0833333333333333]
[                 0, 0.0833333333333333,  0.166666666666667]
>>> b
[          0.03125]
[0.104166666666667]
[          0.03125]
>>> c = A.LUsolve(b)
>>> c
[0.0416666666666666]
[ 0.291666666666667]
[0.0416666666666666]
```

The `fe_approx1D` module contains functions for generating the `nodes` and `elements` lists for equal-sized elements with any number of nodes per element. The coordinates in `nodes` can be expressed either through the element length symbol `h` or by real numbers. There is also a function

```
def approximate(f, symbolic=False, d=1, n_e=4, filename='tmp.eps'):
```

which computes a mesh with `n_e` elements, basis functions of degree `d`, and approximates a given symbolic expression `f` by a finite element expansion $u(x) = \sum_j c_j\varphi_j(x)$. When `symbolic` is `False`, $u(x)$ can be computed at a (large) number of points and plotted together with $f(x)$. The construction of u points from the solution vector `c` is done elementwise by evaluting $\sum_r c_r\tilde{\varphi}_r(X)$ at a (large) number of points in each element, and the discrete (x,u) values on each elements are stored in arrays that are finally concatenated to form global arrays with the x and u coordinates for plotting. The details are found in the `u_glob` function in `fe_approx1D.py`.

1.4.9 The Structure of the Coefficient Matrix

Let us first see how the global matrix looks like if we assemble symbolic element matrices, expressed in terms of `h`, from several elements:

```
>>> d=1; n_e=8; Omega=[0,1]  # 8 linear elements on [0,1]
>>> phi = basis(d)
>>> f = x*(1-x)
>>> nodes, elements = mesh_symbolic(n_e, d, Omega)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=True)
>>> A
[h/3,   h/6,     0,     0,     0,     0,     0,     0,   0]
[h/6, 2*h/3,   h/6,     0,     0,     0,     0,     0,   0]
[  0,   h/6, 2*h/3,   h/6,     0,     0,     0,     0,   0]
[  0,     0,   h/6, 2*h/3,   h/6,     0,     0,     0,   0]
[  0,     0,     0,   h/6, 2*h/3,   h/6,     0,     0,   0]
[  0,     0,     0,     0,   h/6, 2*h/3,   h/6,     0,   0]
[  0,     0,     0,     0,     0,   h/6, 2*h/3,   h/6,   0]
[  0,     0,     0,     0,     0,     0,   h/6, 2*h/3, h/6]
[  0,     0,     0,     0,     0,     0,     0,   h/6, h/3]
```

The reader should assemble the element matrices by hand and verify this result. In general we have a coeffient matrix that is tridiagonal:

$$A = \frac{h}{6}\begin{pmatrix}
2 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0\\
1 & 4 & 1 & \ddots & & & & & \vdots\\
0 & 1 & 4 & 1 & \ddots & & & & \vdots\\
\vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots\\
\vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots\\
\vdots & & & 0 & 1 & 4 & 1 & \ddots & \vdots\\
\vdots & & & & \ddots & \ddots & \ddots & \ddots & 0\\
\vdots & & & & & \ddots & 1 & 4 & 1\\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 & 2
\end{pmatrix} \tag{1.71}$$

The structure of the right-hand side is more difficult to reveal since it involves an assembly of elementwise integrals of $f(x(X))\tilde{\varphi}_r(X)h/2$, which obviously depend on $f(x)$. It is easier to look at the integration in x coordinates, which gives the general formula (1.53). For equal-sized elements of length h, we can apply the Trapezoidal rule to arrive at a somewhat more specific expression that (1.53).

$$b_i = h\left(\frac{1}{2}\phi_i(x_0)f(x_0) + \frac{1}{2}\phi_i(x_N)f(x_N) + \sum_{j=1}^{N-1}\phi_i(x_i)f(x_i)\right) = \begin{cases} \frac{1}{2}hf(x_i), & i = 0 \text{ or } i = N, \\ hf(x_i), & 1 \leq i \leq N-1 \end{cases} \tag{1.72}$$

The reason for this simple formula is simply that ϕ_i is either 0 or 1 at the nodes and 0 at all but one of them.

Going to P2 elements (`d=2`) leads to the element matrix

$$A^{(e)} = \frac{h}{30}\begin{pmatrix} 4 & 2 & -1 \\ 2 & 16 & 2 \\ -1 & 2 & 4 \end{pmatrix} \tag{1.73}$$

and the following global assembled matrix from four elements:

$$A = \frac{h}{30}\begin{pmatrix} 4 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 16 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & 8 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 16 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & 8 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 16 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & 8 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 16 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 & 4 \end{pmatrix} \tag{1.74}$$

In general, for i odd we have the nonzeroes

$$A_{i,i-2} = -1, \quad A_{i-1,i} = 2, \quad A_{i,i} = 8, \quad A_{i+1,i} = 2, \quad A_{i+2,i} = -1,$$

multiplied by $h/30$, and for i even we have the nonzeros

$$A_{i-1,i} = 2, \quad A_{i,i} = 16, \quad A_{i+1,i} = 2,$$

multiplied by $h/30$. The rows with odd numbers correspond to nodes at the element boundaries and get contributions from two neighboring elements in the assembly process, while the even numbered rows correspond to internal nodes in the elements where the only one element contributes to the values in the global matrix.

1.4.10 Applications

With the aid of the `approximate` function in the `fe_approx1D` module we can easily investigate the quality of various finite element approximations to some given functions. Figure 1.13 shows how linear and quadratic elements approximates the polynomial $f(x) = x(1-x)^8$ on $\Omega = [0,1]$. The results arise from the program

```
import sympy as sm
from fe_approx1D import approximate
x = sm.Symbol('x')

approximate(x*(1-x)**8, symbolic=False, d=1, n_e=4)
approximate(x*(1-x)**8, symbolic=False, d=2, n_e=2)
approximate(x*(1-x)**8, symbolic=False, d=1, n_e=8)
approximate(x*(1-x)**8, symbolic=False, d=2, n_e=4)
```

The quadratic functions are seen to be better than the linear ones for the same value of N, as we increase N. This observation has some generality: higher degree is not necessarily better on a coarse mesh, but it is as we refined the mesh.

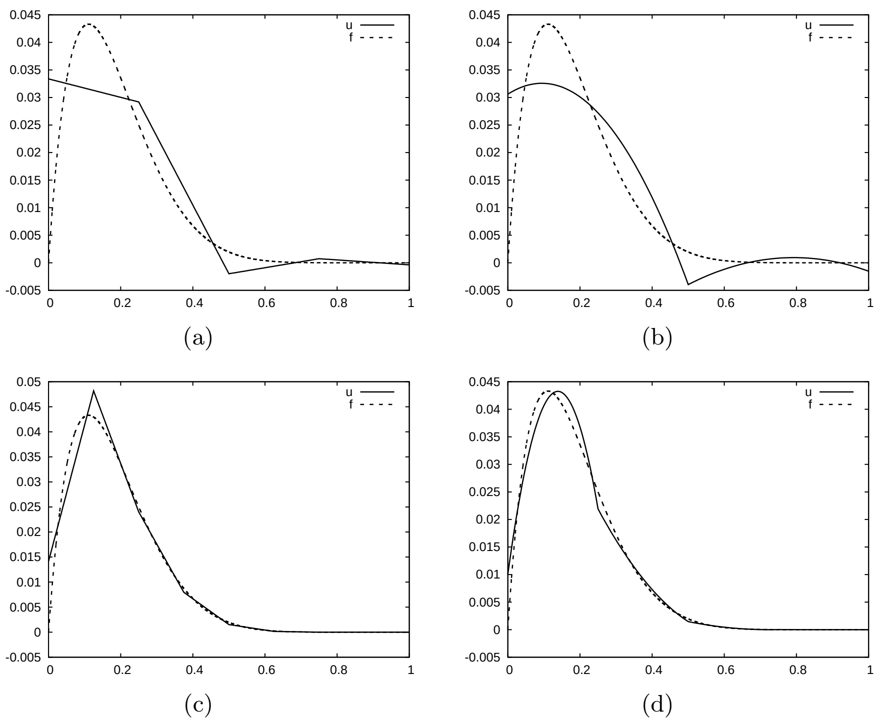


Fig. 1.13. Comparison of the finite element approximation $u(x)$ to $f(x) = x(1-x)^8$ on $\Omega = [0,1]$. The approximation applies n_e equal-sized elements with polynomials of degree d, resulting in N unknowns: (a) $d = 1$, $n_e = 4$, $N = 5$; (b) $d = 2$, $n_e = 2$, $N = 5$; (c) $d = 1$, $n_e = 8$, $N = 9$; (d) $d = 2$, $n_e = 4$, $N = 9$.

Some of the examples in the present section took several minutes to compute, even on small meshes consisting of up to eight elements. The main explanation for slow computations is unsuccessful symbolic integration: `sympy` may use a lot of energy on integrals like $\int f(x(X))\tilde{\varphi}_r(X)h/2dx$ before giving up, and the program resorts to numerical integration. Codes that can deal with a large number of basis functions and accept flexible choices of $f(x)$ should compute all integrals numerically and replace the matrix objects from `sympy` by the far more efficent array objects from `numpy`. In particular, one should turn to sparse storage formats for the coefficient matrix since most of the terms are known to be zero, and also utilize efficient algorithms for solving linear systems with sparse coefficient matrices. This is the topic of the next two sections.

1.4.11 Numerical Integration

Finite element codes usually apply numerical approximations to integrals. Since the integrands in the coefficient matrix often are (lower-order) polynomials, integration rules that can integrate polynomials exactly are popular.

The numerical integration rules can be expressed on a common form,

$$\int_{-1}^{1} g(X)dX \approx \sum_{j=0}^{M} w_j\bar{X}_j, \tag{1.75}$$

where $\bar{X}_j$ are *integration points* and w_j are *integration weights*, $j = 0, \ldots, M$. Different rules correspond to different choices of points and weights.

Three well-known rules are the *Midpoint* rule,

$$\int_{-1}^{1} g(X)dX \approx 2g(0), \quad \bar{X}_0 = 0,\ w_0 = 2, \tag{1.76}$$

the *Trapezoidal rule*,

$$\int_{-1}^{1} g(X)dX \approx g(-1) + g(1), \quad \bar{X}_0 = -1,\ \bar{X}_1 = 1,\ w_0 = w_1 = 1, \tag{1.77}$$

and *Simpson's rule*,

$$\int_{-1}^{1} g(X)dX \approx \frac{1}{3}\left(g(-1) + 4g(0) + g(1)\right), \tag{1.78}$$

where

$$\bar{X}_0 = -1,\ \bar{X}_1 = 0,\ \bar{X}_2 = 1,\ w_0 = w_2 = \frac{1}{3},\ w_1 = \frac{4}{3}\,. \tag{1.79}$$

All these rules apply equally spaced points. More accurate rules, for a given M, arise if the location of the points are optimized for polynomial integrands.

The *Gauss-Legendre rules* (also known as *Gauss-Legendre quadrature*) constitute one such class of integration methods. Two widely Gauss-Legendre rules in this family have the choice

$$M = 1: \quad \bar{X}_0 = -\frac{1}{\sqrt{3}},\ \bar{X}_1 = \frac{1}{\sqrt{3}},\ w_0 = w_1 = 1 \tag{1.80}$$

$$M = 2: \quad \bar{X}_0 = -\sqrt{\frac{3}{\sqrt{5}}},\ \bar{X}_0 = 0,\ \bar{X}_2 = \sqrt{\frac{3}{\sqrt{5}}},\ w_0 = w_2 = \frac{5}{9},\ w_1 = \frac{8}{9}\,. \tag{1.81}$$

These rules integrate 3rd and 5th degree polynomials exactly. In general, an M-point Gauss-Legendre rule integrates a polynomial of degree $2M+1$ exactly.

1.4.12 Summary of a Finite Element

The concept of a finite element contains four key components, which we now formulate in a more abstract sense suitable for later use. A finite element is defined by

1. a geometric *domain* in a local *reference coordinate system*;
2. a set of $d+1$ *basis functions* $\tilde{\varphi}_i$ defined on the element;
3. a set of $d+1$ *degrees of freedom* that uniquely determine the basis functions such that $\tilde{\varphi}_i = 1$ for degree of freedom number i and $\tilde{\varphi}_i = 0$ for all other degrees of freedom;
4. a *mapping* of the element from the reference coordinate system to the physical coordinate system.

Property 3 ensures that a finite element function u can be written as

$$u(x) = \sum_{j=0}^{N} c_j \tilde{\varphi}_j(x),$$

where c_j is the value of degree of freedom number j of u. The most common example of a degree of freedom is the function value at a point. With a mapping between local degrees of freedom and global degrees of freedom, one can relate the expansion of u on an element to its expansion in the global physical domain.

The elements we have seen so far in these notes have been one-dimensional and the geometric domain is therefore an interval and in particular the interval $[-1, 1]$ in the reference coordinate system X. As basis functions we have chosen Lagrange polynomials. The degrees of freedom are then the function values at $d+1$ nodes. We have used equally spaced nodes. The mapping from the reference system to the physical system is linear.

We shall see that the above characteristics of an element generalize to higher dimensions and to much more complicated elements. The concept of degrees of freedom is important: we may choose other parameters than the function values at points as the interpretation of the coefficients c_j. Here is one example. Suppose we want an approximation to f based on piecewise *constant* functions. Then we can construct an element with one local basis function, $\tilde{\varphi}_0(X) = 1$. The associated degree of freedom can be taken as the function value at a node in the middle of the element. In this case the element will have only one node and no nodes on the boundary. Alternatively, we can omit the concept of nodes and say the degree of freedom is the *mean value* of a function rather than a point value. That is, c_0 is the mean value of u over the element. To get a mean value (degree of freedom value) of 1 for $\tilde{\varphi}_0(x)$ over $[-1,1]$, we must have $\tilde{\varphi}_0(X) = 1/2$. A global basis functions is associated with one element, typically φ_i equals $1/h_i$, where h_i is the length of element i. Then $\int_{\Omega^{(i)}} \varphi_i dx = 1$. The mapping from local degrees of freedom to global degrees of freedom is simple: local degree of freedom 0 in element e maps to global degree of freedom e.

1.4.13 Sparse Matrix Storage and Solution

1.4.14 Accuracy of Piecewise Polynomial Approximations

Experimental.

1.5 Exercises

Exercise 1.1. Linear algebra refresher I.

Look up the topic of *vector space* in your favorite linear algebra book or search for the term at Wikipedia. Prove that vectors in the plane (a,b) form a vector space by showing that all the axioms of a vector space are satisfied. Similarly, prove that all linear functions of the form $ax+b$ constitute a vector space. ◇

Exercise 1.2. Linear algebra refresher II.

As an extension of Exercise 1.2, check out the topic of *inner vector spaces.* Show that both examples of spaces in Exercise 1.2 can be equipped with an inner product and show that the choice of inner product satisfied the general requirements of an inner product in a vector space. ◇

Exercise 1.3. Approximate a three-dimensional vector in a plane.

Given $\boldsymbol{f} = (1,1,1)$ in $\mathbb{R}^3$, find the best approximation vector $\boldsymbol{u}$ in the plane spanned by the unit vectors $(1,0)$ and $(0,1)$. Name of program file: `vec111_approx1.py`. ◇

Exercise 1.4. Solve Exer. 1.3 using a different basis.

We address the same approximation problem as in Exercise 1.3, but choose $\boldsymbol{u} \in V$ where

$$V = \text{span}\{(2,1),(1,2)\}\,.$$

Name of program file: `vec111_approx2.py`. ◇

Exercise 1.5. Approximate the exponential function by power functions.

Let V be a function space with basis functions x^k, $k = 0, 1, \ldots, N$. Find the best approximation to $f(x) = e^x$ among all functions in V, using $N = 8$ and the `least_squares` function from Chapter 1.3. Name of program file: `exp_by_powers.py`. ◇

Exercise 1.6. Approximate a high frequency sine function by lower frequency sines.

Find the best approximation of $f(x) = \sin(20x)$ on $[0, 2\pi]$ in the space V with basis

$$\{\sin x,\ \sin 2x, \sin 3x\},$$

using the `least_squares_orth` function from Chapter 1.3.6. Plot $f(x)$ and its approximation. Name of program file: `hilow_sine_approx.py`. ◇

Exercise 1.7. Fourier series as a least-squares approximation.

Given a function $f(x)$ on an interval $[0, L]$, find the formula for the coefficients of the Fourier series of f:

$$f(x) = a_0 + \sum_{j=1}^{\infty} a_j \cos\left(j\frac{\pi x}{L}\right) + \sum_{j=1}^{\infty} b_j \sin\left(j\frac{\pi x}{L}\right)\,.$$

Let an infinite-dimensional vector space V have the basis functions $\cos j\frac{\pi x}{L}$ for $j = 0, 1, \ldots, \infty$ and $\sin j\frac{\pi x}{L}$ for $j = 1, \ldots, \infty$. Show that the least-squares approximation method from Chapter 1.3 leads to a linear system whose solution coincides with the standard formulas for the coefficients in a Fourier series of $f(x)$ (see also Chapter 1.3.6). You may choose

$$\varphi_{2i} = \cos\left(i\frac{\pi}{L}x\right), \quad \varphi_{2i+1} = \sin\left(i\frac{\pi}{L}x\right),$$

for $i = 0, 1, \ldots, N \to \infty$.

Choose a specific function $f(x)$, calculate the coefficients in the Fourier expansion by solving the linear system, arising from the least squares method, by hand. Plot some truncated versions of the series together with $f(x)$ to show how the series expansion converge. Name of program file: `Fourier_series.py`. ◇

Exercise 1.8. Approximate a tanh *function by Lagrange polynomials.*

Use interpolation (or collocation) with uniformly distributed points and Chebychev nodes to approximate

$$f(x) = \tanh(s(x - \frac{1}{2}))$$

by Lagrange polynomials for $s = 10, 100$ and $N = 3, 6, 9, 11$. Name of program file: `tanh_approx.py`. ◇

Exercise 1.9. Improve an approximation by sines.

Consider the approximations of a parabola by a sum of sine functions in Chapter 1.3.6. Since we always have that $u(0) = 0$ the approximation at $x = 0$ can never be good. Try a remedy:

$$u(x) = f(0) + \sum_{j=0}^{N} \sin((i+1)\pi x)\,.$$

Now $u(0) = f(0)$. Plot the approximations for $N = 4$ and $N = 12$ together with f. Is the approximation better than the ones in Figure 1.2? Name of program file: `parabola_by_sines.py`. ◇

Exercise 1.10. Define finite element meshes.

Consider a domain $\Omega = [0, 2]$ divided into the three elements $[0, 1]$, $[1, 1.2]$, and $[1.2, 2]$. Suggest three different element numberings and global node numberings for this mesh and set up the corresponding `nodes` and `elements` lists in each case. Then subdivide the element $[1.2, 2]$ into two new equal-sized elements and explain how you can extend the `nodes` and `elements` data structures to incorporate the new elements and nodes. `fe_numberings.py`. ◇

Exercise 1.11. Approximate a step function by finite elements.

Approximate the step function

$$f(x) = \begin{cases} 1 & x < 1/2, \\ 2 & x \geq 1/2 \end{cases}$$

This f can also be expressed in terms of the Heaviside function $H(x)$: $f(x) = H(x-1/2)$. Use 2, 4, and 8 P1 and P2 elements, and compare approximations visually.

Hint: f can be defined by `f = sm.Heaviside(x -  sm.Rational(1,2))`, making the `approximate` function in the `fe_approx1D.py` module an obvious candidate to solve the problem. However, `sympy` does not handle symbolic integration with the integrands and the `approximate` function faces a problem when converting `f` to a Python function (for plotting) since `Heaviside` is not an available function in `numpy`. Make special-purpose code for this case instead, or perform all caluclations by hand. Name of program file: `Heaviside_approx_P1P2.py`. ◇

Exercise 1.12. Perform symbolic finite element computations.

Find the coefficient matrix and right-hand side for approximating $f(x) = A \sin \omega x$ on $\Omega = [0, 2\pi/\omega]$ by P1 elements of size h. Perform the calculations in software. Solve the system in case of two elements. `Asinwt_approx_P1.py`. ◇

Exercise 1.13. ....

`...py`. ◇

Chapter 2

Stationary Diffusion

The finite element method is a very flexible approach for solving partial differential equations. Its two most attractive features are the ease of handling domains of complex shape in two and three dimensions and the ease of constructing higher-order discretization methods. The finite element method is usually applied for discretization in space, and therefore spatial problems will be our focus in this chapter. Extensions to time-dependent problems are covered in the next chapter.

2.1 Basic Principles

2.1.1 Differential Equation Models

Let us consider an abstract differential equation for a function $u(x)$ of one variable, written as

$$\mathcal{L}(u) = 0, \quad x \in \Omega\,. \tag{2.1}$$

Here are a few examples on possible choices of $\mathcal{L}(u)$, of increasing complexity:

$$\mathcal{L}(u) = \frac{d^2u}{dx^2} - f(x), \tag{2.2}$$

$$\mathcal{L}(u) = \frac{d}{dx}\left(\alpha(x)\frac{du}{dx}\right) + f(x), \tag{2.3}$$

$$\mathcal{L}(u) = \frac{d}{dx}\left(\alpha(u)\frac{du}{dx}\right) - \omega^2 u + f(x), \tag{2.4}$$

$$\mathcal{L}(u) = \frac{d}{dx}\left(\alpha(u)\frac{du}{dx}\right) + f(u,x)\,. \tag{2.5}$$

Both $\alpha(x)$ and $f(x)$ are considered as specified functions, while ω is a prescribed parameter. Differential equations corresponding to (2.2)–(2.3) arise in diffusion phenomena, such as steady transport of heat in solids and flow of viscous fluids between flat plates. The form (2.4) arises when transient diffusion or wave phenomenon are discretized in time by finite differences. The equation (2.5) appear in chemical models when diffusion of a substance is combined with chemical reactions. Also in biology, (2.5) plays an important role, both for spreading of species and in physiological models involving generation and propagation of electrical signals.

Let $\Omega = [0, L]$ be the domain in one space dimension. In addition to the differential equation, u must fulfill boundary conditions at the boundaries of the domain, $x = 0$ and $x = L$. When $\mathcal{L}$ contains up to second-order derivatives, as in the examples above, $m = 1$, we need one boundary conditions at each of the two boundary points, here abstractly specified as

$$\mathcal{B}_0(u) = 0,\ x = 0, \quad \mathcal{B}_1(u) = 0,\ x = L \tag{2.6}$$

There are three common choices of boundary conditions:

$$\begin{aligned}
\mathcal{B}_i(u) &= u - g, && \text{Dirichlet condition,} && (2.7)\\
\mathcal{B}_i(u) &= -\alpha\frac{du}{dx} - g, && \text{Neumann condition,} && (2.8)\\
\mathcal{B}_i(u) &= -\alpha\frac{du}{dx} - a(u - g), && \text{Robin condition}\,. && (2.9)\\
& && && (2.10)
\end{aligned}$$

Here, g and a are specified quantities.

From now on we shall use $u_{\mathrm{e}}(x)$ as symbol for the *exact* solution, fulfilling

$$\mathcal{L}(u_{\mathrm{e}}) = 0, \quad x \in \Omega, \tag{2.11}$$

while $u(x)$ denotes an *approximate* solution of the differential equation. We must immediately remark that in the literature about the finite element method, is common to use u as the exact solution and u_h as the approximate solution, where h is a discretization parameter. However, the vast part of the present text is about the approximate solutions, and having a subscript h attached all the time is cumbersome. Of equal importance is the close correspondence between implementation and mathematics that we strive to achieve in this book: when it is natural to use `u` and not `u_h` in code, we let the mathematical notation be dictated by the code's preferred notation. After all, it is the powerful computer implementations of the finite element method that justifies studying the mathematical formulation and aspects of the method.

A common model problem used much in this chapter is

$$-u''(x) = f(x), \quad x \in \Omega = [0, L], \quad u(0) = 0,\ u(L) = D\,. \tag{2.12}$$

The specific choice of $f(x) = 2$ gives the solution

$$u_{\mathrm{e}}(x) = x(U_1 + L - x)\,.$$

A closely related problem with a different boundary condition at $x = 0$ reads

$$-u''(x) = f(x), \quad x \in \Omega = [0, L], \quad u'(0) = E,\ u(L) = D\,. \tag{2.13}$$

A third variant has a variable coefficient,

$$-(\alpha(x)u'(x))' = f(x), \quad x \in \Omega = [0, L], \quad u'(0) = E,\ u(L) = D\,. \tag{2.14}$$

2.1.2 Residual-Minimizing Principles

The fundamental idea is to seek an approximate solution u in some space V with basis

$$\{\varphi_0(x), \ldots, \varphi_N(x)\},$$

which means that u can always be expressed as[1]

$$u(x) = \sum_{j=0}^{N} c_j \varphi_j(x),$$

for some unknown coefficients $c_0, \ldots, c_N$. As in Chapter 1.3, we need principles for deriving $N+1$ equations to determine the $N+1$ unknowns $c_0, \ldots, c_N$. A key idea of Chapter 1.3 was to minimize the approximation error $e = u - f$. That principle is not so useful here since the approximation error $e = u_{\mathrm{e}} - u$ is unknown to us when u_{e} is unknown. The only general indicator we have on the quality of the approximate solution is to what degree u fulfills the differential equation. Inserting $u = \sum_j c_j \varphi_j$ into $\mathcal{L}(u)$ reveals that the result is not zero, because u is only likely to equal u_{e}. The nonzero result,

$$R = \mathcal{L}(u) = \mathcal{L}(\sum_j c_j \varphi_j), \tag{2.15}$$

is called the *residual* and measures the error in fulfilling the governing equation. Various principles for determining $c_0, \ldots, c_N$ try to minimize R in some sense. Note that R varies with x and the $c_0, \ldots, c_N$ parameters. We may write this dependence explicitly as

$$R = R(x; c_0, \ldots, c_N)\,. \tag{2.16}$$

The Least-Squares Method. The least-squares method aims to find $c_0, \ldots, c_N$ so that the integrated square of the residual,

$$\int_\Omega R^2 dx \tag{2.17}$$

is minimized. By introducing an inner product of two fuctions f and g on Ω as

$$(f, g) = \int_\Omega f(x) g(x) dx, \tag{2.18}$$

the least-squares method can be defined as

$$\min_{c_0, \ldots, c_N} E = (R, R)\,. \tag{2.19}$$

[1] Later, in Chapter 2.1.6, we will see that if we specify boundary values of u different from zero, we must look for an approximate solution $u(x) = B(x) + \sum_{j=0}^{N} c_j \varphi_j(x)$, where $\sum_j c_j \varphi_j \in V$ and $B(x)$ is some function for incorporating the right boundary values. Because of $B(x)$, u will not necessarily lie in V. This modification does not imply any difficulties.

Differentiating with respect to the free parameters $c_0, \ldots, c_N$ gives the $N+1$ equations

$$\int_\Omega 2R\frac{\partial R}{\partial c_i}dx = 0 \quad \Leftrightarrow \quad (R, \frac{\partial R}{\partial c_i}) = 0, \quad i = 0, \ldots, N\,. \tag{2.20}$$

The Galerkin Method. From Chapter 1.3.1 we saw that the least-squares principle is equivalent to demanding the error to be orthogonal to the space V. Adopting this principle for the residual R, instead of the true error, implies that we seek $c_0, \ldots, c_N$ such that

$$(R, v) = 0, \quad \forall v \in V\,. \tag{2.21}$$

This is the Galerkin method for differential equations. As shown in (1.19)–(1.20), this statement is equivalent to R being orthogonal to the $N+1$ basis functions only,

$$(R, \varphi_i) = 0, \quad i = 0, \ldots, N, \tag{2.22}$$

yielding $N+1$ equations for determining $c_0, \ldots, c_N$.

The Method of Weighted Residuals. A generalization of the Galerkin method is to demand that R is orthogonal to some space W, not necessarily the same space as V where we seek the unknown function. This generalization is naturally called the *method of weighted residuals*:

$$(R, v) = 0, \quad \forall v \in W\,. \tag{2.23}$$

If $\{w_0, \ldots, w_N\}$ is a basis for W, we can equivalently express the method of weighted residuals as

$$(R, w_i) = 0, \quad i = 0, \ldots, N\,. \tag{2.24}$$

The result is $N+1$ equations for $c_0, \ldots, c_N$.

The least-squares method can also be viewed as a weighted residual method with $w_i = \partial R/\partial c_i$.

Variational Formulation. Formulations like (2.23) (or (2.24)) and (2.21) (or (2.21)) are known as *variational formulations*[2]. These equations are in this book foremost used for a numerical approximation $u \in V$, where V is a finite-dimensional space with dimension $N+1$. However, we may also let V

[2] It may be subject to debate whether it is only the form of (2.23) or (2.21) after integration by parts, as explained in Chapter 2.1.4, that qualifies for the term variational formulation. The result after integration by parts is what is obtained after taking the *first variation* of an optimization problem, see Chapter 2.2.1. However, here we use variational formulation as a common term for formulations which, in contrast to the differential equation $R = 0$, instead demand that an average of R is zero: $(R, v) = 0$ for all v in some space.

be an inifite-dimensional space containing the exact solution $u_{\mathrm{e}}(x)$ such that also u_{e} fulfills a variational formulation. The variational formulation is in that case a mathematical way of stating the problem, being an alternative to the usual formulation of a differential equation with initial and/or boundary conditions.

Test and Trial Functions. In the context of the Galerkin method and the method of weighted residuals it is common to use the name *trial function* for the approximate $u = \sum_j c_j\varphi_j$. The space containing the trial function is known as the *trial space*. The function v entering the orthogonality requirement in the Galerkin method and the method of weighted residuals is called *test function*, and so are the φ_i or w_i functions that are used as weights in the inner products with the residual. The space where the test functions comes from is naturally called the *test space*.

We see that in the method of weighted residuals the test and trial spaces are different and so are the test and trial functions. In the Galerkin method the test and trial spaces are the same (so far). Later in Chapter 2.1.6 we shall see that boundary conditions may lead to a difference between the test and trial spaces in the Galerkin method.

The Collocation Method. The idea of the collocation method is to demand that R vanishes at $N+1$ selected points $x_0,\ldots,x_N$ in Ω:

$$R(x_i; c_0,\ldots,c_N) = 0, \quad i = 0,\ldots,N\,. \tag{2.25}$$

The collocation method can also be viewed as a method of weighted residuals with Dirac delta functions as weighting functions. Let $\delta(x-x_i)$ be the Dirac delta function centered around $x = x_i$ with the properties that $\delta(x-x_i) = 0$ for $x \neq x_i$ and

$$\int_\Omega f(x)\delta(x-x_i)dx = f(x_i), \quad x_i \in \Omega\,. \tag{2.26}$$

Intuitively, we think of $\delta(x-x_i)$ as a very peak-shaped function around $x = x_i$ with integral 1. Because of (2.26), we can let $w_i = \delta(x-x_i)$ be weighting functions in the method of weighted residuals, and (2.24) becomes equivalent to (2.25).

The Subdomain Collocation Method. The idea of this approach is to demand the integral of R to vanish over $N+1$ subdomains Ω_i of Ω:

$$\int_{\Omega_i} R\,dx = 0, \quad i = 0,\ldots,N\,. \tag{2.27}$$

2.1.3 Examples on Using the Principles

Let us now apply global basis function to illustrate the principles for minimizing R. Our choice of basis functions φ_i for V is

$$\varphi_i(x) = \sin\left((i+1)\pi\frac{x}{L}\right), \quad i = 0,\ldots,N\,. \tag{2.28}$$

The following property of these functions becomes useful in the forthcoming calculations:

$$\int_0^L \sin\left((i+1)\pi\frac{x}{L}\right)\sin\left((j+1)\pi\frac{x}{L}\right)\,dx = \begin{cases}\frac{1}{2}L & i = j\\ 0, & i \neq j\end{cases} \tag{2.29}$$

provided i and j are integers.

We address the model problem (2.12). One immediate difficulty is that with the above choice of basis functions, $u(1) = \sum_j c_j \sin((i+1)\pi) = 0$ regardless of the coefficients $c_0,\ldots,c_N$. We therefore set $U_1 = 0$ so that $u(x)$ fulfills the boundary conditions $u(0) = u(1) = 0$. Later, in Chapter 2.1.6, we shall see how we can deal with the condition $u(1) = U_1 \neq 0$.

The Residual. We can readily calculate the following explicit expression for the residual:

$$\begin{aligned}
R(x; c_0,\ldots,c_N) &= u''(x) + f(x),\\
&= \frac{d^2}{dx^2}\left(\sum_{j=0}^{N} c_j\varphi_j(x)\right) + f(x),\\
&= -\sum_{j=0}^{N} c_j\varphi_j''(x) + f(x)\,.
\end{aligned} \tag{2.30}$$

The Least-Squares Method. The equations (2.20) in the least-squares method require an expression for $\partial R/\partial c_i$. We have

$$\frac{\partial R}{\partial c_i} = \frac{\partial}{\partial c_i}\left(\sum_{j=0}^{N} c_j\varphi_j''(x) + f(x)\right) = \varphi_i''(x)\,. \tag{2.31}$$

The governing equations for $c_0,\ldots,c_N$ are then

$$(\sum_j c_j\varphi_j'' + f, \varphi_i'') = 0, \quad i = 0,\ldots,N, \tag{2.32}$$

which can be rearranged as

$$\sum_{j=0}^{N} (\varphi_i'', \varphi_j'')c_j = -(f, \varphi_i''), \quad i = 0,\ldots,N\,. \tag{2.33}$$

This is nothing but a linear system

$$\sum_{j=0}^{N} A_{i,j} c_j = b_i, \quad i = 0, \dots, N,$$

with

$$\begin{aligned} A_{i,j} &= (\varphi_i'', \varphi_j'') \\ &= \pi^4 (i+1)^2 (j+1)^2 L^{-4} \int_0^L \sin\left((i+1)\pi \frac{x}{L}\right) \sin\left((j+1)\pi \frac{x}{L}\right) \, dx \\ &= \begin{cases} \frac{1}{2} L^{-3} \pi^4 (i+1)^4 & i = j \\ 0, & i \neq j \end{cases} \end{aligned} \tag{2.34}$$

$$b_i = -(f, \varphi_i'') = (i+1)^2 \pi^2 L^{-2} \int_0^L f(x) \sin\left((i+1)\pi \frac{x}{L}\right) \, dx \tag{2.35}$$

Since the coefficient matrix is diagonal we can easily solve for

$$c_i = \frac{2L^2}{\pi^2 (i+1)^2} \int_0^L f(x) \sin\left((i+1)\pi \frac{x}{L}\right) \, dx \, . \tag{2.36}$$

With the special choice of $f(x) = 2$ the integral becomes

$$\frac{L \cos(\pi i) + L}{\pi (i+1)},$$

according to *http://wolframalpha.com*(use`j` and not `i` ($= \sqrt{-1}$) when asking). Hence,

$$c_i = \frac{4L^3 (1 + (-1)^i)}{\pi^3 (i+1)^3} \, .$$

Now, $1 + (-1)^i = 0$ for i odd, so only the coefficients with even index are nonzero. Introducing $i = 2k$ for $k = 0, \dots, N/2$ to count the relevant indices, we get the solution

$$u(x) = \sum_{k=0}^{N/2} \frac{8L^3}{\pi^3 (2k+1)^3} \sin\left((2k+1)\pi \frac{x}{L}\right) \, . \tag{2.37}$$

The coefficients decay very fast: $c_2 = c_0/27$, $c_4 = c_0/125$. The solution will therefore be dominated by the first term,

$$u(x) \approx \frac{8L^3}{\pi^3} \sin\left(\pi \frac{x}{L}\right) \, .$$

The Galerkin Method. The Galerkin principle (2.21) applied to (2.12) consists of inserting our special residual (2.30) in (2.21)

$$(u'' + f, v) = 0, \quad \forall v \in V,$$

or

$$(u'', v) = -(f, v), \quad \forall v \in V\,. \tag{2.38}$$

This is the variational formulation, based on the Galerkin principle, of our differential equation. By inserting the approximation for u and letting $v = \varphi_i$, as in (2.21) we get

$$(\sum_{j=0}^{N} c_j \varphi_j'', \varphi_i) = -(f, \varphi), \quad i = 0, \ldots, N\,. \tag{2.39}$$

This equation can be rearranged to a form that explicitly shows that we get a linear system for the unknowns $c_0, = \ldots, c_N$:

$$\sum_{j=0}^{N} (\varphi_i, \varphi_j'') c_j = (f, \varphi_i), \quad i = 0, \ldots, N\,. \tag{2.40}$$

For the particular choice of the basis functions (2.28) we get in fact the same linear system as in the least-squares method (because $\varphi'' = \text{const}\,\varphi$).

The Collocation Method. For the collocation method (2.25) we need to decide upon a set of $N+1$ collocation points in Ω. A simple choice is to use uniformly spaced points: $x_i = i\Delta x$, where $\Delta x = L/N$ in our case ($N \geq 1$). However, these points lead to at least two rows in the matrix consisting of zeros (since $\varphi_i(x_0) = 0$ and $\varphi_i(x_N) = 0$), thereby making the matrix singular and non-invertible. This forces us to choose some other collocation points, e.g., random points:

```
points = np.random.uniform(0, L, size=N+1)
```

Demanding the residual to vanish at these points leads, in our model problem (2.12), to the equations

$$-\sum_{j=0}^{N} c_j \varphi_j''(x_i) = f(x_i), \quad i = 0, \ldots, N\,.. \tag{2.41}$$

This is seen to be a linear system with entries

$$A_{i,j} = -\varphi_j''(x_i) = (j+1)^2 \pi^2 L^{-2} \sin\left((j+1)\pi i \frac{L}{NL}\right),$$

in the coefficient matrix and entries $b_i = 2$ for the right-hand side (when $f(x) = 2$). The special case of $N = 0$ can sometimes be of interest. A natural choice is then the midpoint $x_0 = L/2$ of the domain, which here results in $A_{0,0} = \pi^2$ and $c_0 = 2L^2/\pi^2$.

Different dimension from Galerkin/LS

Comparison.

2.1.4 Integration by Parts

A problem arises if we want to use the finite element functions from Chapter 1.4 to solve our model problem (2.12) by the least-squares, Galerkin, or collocation methods: the piecewise polynomials $\varphi_i(x)$ have discontinuous derivatives at the element boundaries which makes it problematic to compute $\varphi_i''(x)$. This fact actually makes the least-squares and collocation methods less suitable for finite element approximation of the unknown function[3]. The Galerkin method and the method of weighted residuals can, however, be applied using integration by parts as a means for transforming a second-order derivative to a first-order one.

Consider the model problem (2.12) and its Galerkin formulation

$$-(u'', v) = (f, v) \quad \forall v \in V\,.$$

Using integration by parts in the Galerkin method, we can move a derivative on u to v:

$$\begin{aligned}\int_0^L u''(x)v(x)dx &= -\int_0^L u'(x)v'(x)dx + [vu']_0^L\\ &= -\int_0^L u'(x)v'(x)dx + u'(L)v(L) - u'(0)v(0)\,. \end{aligned} \tag{2.42}$$

Usually, one integrates the problem at the stage where the u and v functions enter the formulation. Alternatively, we can integrate by parts in the expressions for the matrix entries:

$$\begin{aligned}\int_0^L \varphi_i(x)\varphi_j''(x)dx &= -\int_0^L \varphi_i'(x)\varphi_j'(x)dx + [\varphi_i\varphi_j']_0^L\\ &= -\int_0^L \varphi_i'(x)\varphi_j'(x)dx + \varphi_i(L)\varphi_j'(L) - \varphi_i(0)\varphi_j'(0)\,. \end{aligned} \tag{2.43}$$

Integration by parts serves to reduce the order of the derivatives and to make the coefficient matrix symmetric since $(\varphi_i', \varphi_j') = (\varphi_i', \varphi_j')$. The symmetry property depends on the type of terms that enter the differential equation. As will be seen later in Chapter 2.1.7, integration by parts also provides a method for implementing boundary conditions involving u'.

[3] By rewriting the equation $-u'' = f$ as a system of two first-order equations, $u' = v$ and $-v' = f$, the least-squares method can be applied. Differentiating discontinuous functions can, however, be handled by distribution theory in mathematics, but this is a topc beyond the classical calculus demanded in this book.

With the choice (2.28) of basis functions we see that the "boundary terms" $\varphi_i(L)\varphi_j'(L)$ and $\varphi_i(0)\varphi_j'(0)$ vanish since $\varphi_i(0) = \varphi_i(L) = 0$. This will happen in many other examples too.

Since the variational formulation after integration by parts make weaker demands on the differentiability of u and the basis functions φ_i, the resulting integral formulation is referred to as a *weak form* of the differential equation problem. The original variational formulation with second-order derivatives, or the differential equation problem with second-order derivative, is then the *strong form*, with stronger requirements on the differentiability of the functions.

For differential equations with second-order derivatives, expressed as variational formulations and solved by finite element methods, we will always perform integration by parts to arrive at expressions involving only first-order derivatives.

2.1.5 Computing with Finite Elements

The purpose of this section is to demonstrate how the model problem (2.12) with $D = 0$ and $f(x) = 2$ can be solved using finite element basis functions. The appropriate variational formulation is given by (2.42). Since u is known to be zero at the end points of the interval, we can utilize a sum over the basis functions associated with internal nodes only:

$$u(x) = \sum_{j=1}^{N-1} c_j \varphi_j(x)\,.$$

Observe that $u(0)$ and $u(L)$ are zero since φ_0 and φ_N are left out of the sum. This means that only $c_1, \ldots, c_{N-1}$ are unknowns and the variational statement in (2.42) holds only for $i = 1, \ldots, N-1$. For simplicity, we introduce uniformly spaced nodes:

$$x_i = ih, \quad h = L/N, \quad i = 0, \ldots, N\,.$$

The simplest choice of elements are P1 elements with piecewise linear functions. The elements are then $\Omega^{(e)} = [x_e, x_{e+1}]$.

Global Computation. We shall first perform a computation in the x coordinate system because the integrals can be easily computed here by some geometric considerations. This is called a global approach since we work in the x coordinate system and compute integrals on the global domain $[0, L]$.

The $\varphi_i(x)$ function is specified in (1.51) on page 26. The entries in the coefficent matrix and right-hand side are

$$A_{i,j} = \int_0^L \varphi_i'(x)\varphi_j'(x)dx, \quad b_i = \int_0^L 2\varphi_i(x)dx\,.$$

We need the derivative of $\varphi_i(x)$ to compute the coefficient matrix. These are

$$\varphi_i(x) = \begin{cases} 0, & x < x_{i-1}, \\ h^{-1}, & x_{i-1} \le x < x_i, \\ -h^{-1}, & x_i \le x < x_{i+1}, \\ 0, & x \ge x_{i+1} \end{cases} \tag{2.44}$$

We realize that φ'_i and φ'_j has no overlap, and hence their product vanishes, unless i and j are nodes belonging to the same element. The only nonzero contributions to the coefficient matrix are therefore

$$A_{i-1,i} = \int_0^L \varphi'_{i-1}(x)\varphi'_i(x)dx, \quad A_{i,i} = \int_0^L \varphi_i(x)^2 dx, \quad A_{i,i+1} = \int_0^L \varphi'_i(x)\varphi'_{i+1}(x)dx\,.$$

We see that $\varphi'_{i-1}(x)$ and $\varphi'_i(x)$ have overlap of one element $\Omega^{(i-1)} = [x_{i-1}, x_i]$ and that their product then is $-h^{-2}$. Then $A_{i-1,i} = -h^{-2}h = -h^{-1}$. A similar reasoning can be applied to $A_{i+1,i}$, which also becomes $-h^{-1}$. The integral of $\varphi'_i(x)^2$ gets contributions from two elements and becomes $h^{-2}2h = 2h^{-1}$. The right-hand side involves an integral of $\varphi_i(x)$, which is just the area under a "hat" function of height 1 and width $2h$, i.e., equal to h. Hence, $b_i = 2h$.

The equation system to be solved now reads

$$\frac{1}{h}\begin{pmatrix} 2 & -1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} c_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_{N-1} \end{pmatrix} = \begin{pmatrix} 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \end{pmatrix} \tag{2.45}$$

Since we know that c_j equals $u(x_j)$, we can introduce the notation u_j for the value of u at node j. The i-th equation in this system is then

$$-\frac{1}{h}u_{i-1} + \frac{2}{h}u_i - \frac{1}{h}u_{i+1} = 2h\,. \tag{2.46}$$

A finite difference discretization of $-u''(x) = 2$ by a centered, second-order finite difference yields

$$-\frac{u_{i-1} + 2u_i - u_{i+1}}{h^2} = 2, \tag{2.47}$$

which is equivalent to (2.46) if (2.46) is divided by h. Therefore, the finite difference and the finite element method are equivalent in this simple test problem. Sometimes a finite element method generates the finite difference equations on a uniform mesh, and sometimes the finite element method generates equations that are different. The differences are modest, but may influence the numerical quality of the solution significantly. There will be many examples illustrating this point.

Elementwise Computations. We now employ the element by element computational procedure as explained in Chapters 1.4.4–1.4.6. All integrals need to be mapped to the local reference coordinate system (X) according to Chapter 1.4.5. In the present case, the matrix entries contain derivatives with respect to x,

$$A_{i,j} = \int_0^L \varphi_i'(x)\varphi_j'(x)dx,$$

but our basis functions specified in the reference element $[-1,1]$ are expressed in terms of the reference coordinate X. We can easily compute $\tilde{\varphi}_i/dX$:

$$\tilde{\varphi}_0(X) = \frac{1}{2}(1-X), \quad \tilde{\varphi}_1(X) = \frac{1}{2}(1+X),$$

$$\frac{d\tilde{\varphi}_0}{dX} = -\frac{1}{2}, \quad \frac{d\tilde{\varphi}_1}{dX} = \frac{1}{2}\,.$$

From the chain rule,

$$\frac{d\tilde{\varphi}_r}{dx} = \frac{d\tilde{\varphi}_r}{dX}\frac{dX}{dx} = \frac{2}{h}\frac{d\tilde{\varphi}_r}{dX}\,. \tag{2.48}$$

The integral is then transformed as follows:

$$A_{i,j} = \int_0^L \varphi_i'(x)\varphi_j'(x)dx = \int_{-1}^{1} \frac{2}{h}\frac{d\tilde{\varphi}_r}{dX}\frac{2}{h}\frac{d\tilde{\varphi}_s}{dX}\frac{h}{2}dX\,.$$

Here, r and s are local node numbers corresponding to the global i and j, respectively.

The right-hand side is transformed as

$$b_i = \int_0^L 2\varphi_i(x)dx = \int_{-1}^{1} 2\tilde{\varphi}_r(X)\frac{h}{2}dX\,.$$

Specifically for P1 elements we get

$$\begin{aligned}
A_{0,0} &= \int_{-1}^{1} \frac{2}{h}\left(-\frac{1}{2}\right)\frac{2}{h}\left(-\frac{1}{2}\right)\frac{2}{h}dX = \frac{1}{h}\\
A_{0,1} &= \int_{-1}^{1} \frac{2}{h}\left(-\frac{1}{2}\right)\frac{2}{h}\left(\frac{1}{2}\right)\frac{2}{h}dX = -\frac{1}{h}\\
A_{1,0} &= \int_{-1}^{1} \frac{2}{h}\left(\frac{1}{2}\right)\frac{2}{h}\left(-\frac{1}{2}\right)\frac{2}{h}dX = -\frac{1}{h}\\
A_{1,1} &= \int_{-1}^{1} \frac{2}{h}\left(\frac{1}{2}\right)\frac{2}{h}\left(\frac{1}{2}\right)\frac{2}{h}dX = \frac{1}{h}\\
b_0 &= \int_{-1}^{1} 2\frac{1}{2}(1-X)\frac{h}{2}dX = h\\
b_1 &= \int_{-1}^{1} 2\frac{1}{2}(1+X)\frac{h}{2}dX = h
\end{aligned}$$

Writing up the element contributions on matrix and vector form, we have

$$\tilde{A}^{(e)} = A = \frac{1}{h}\begin{pmatrix} 1 & -1\\ -1 & 1\end{pmatrix}, \quad \tilde{b}^{(e)} = h\begin{pmatrix}1\\1\end{pmatrix}. \tag{2.49}$$

The next step is to assemble the contributions from the various elements. Since only the unknowns $c_1, \ldots, c_{N-1}$ enter the linear system, we assembly only $A_{1,1}^{(0)}$ and $A_{0,0}^{(N-1)}$ from the first and last element, respectively. The result becomes identical to (2.45) (which is not surprising since the mathematical procedures are equivalent).

2.1.6 Boundary Conditions: Specified Value

When the unknown function is specified at the boundary, we have to take special actions to incorporate the condition into the computational procedures. This type of boundary condition is therefore called an *essential condition*. The present section outlines alternative (yet mathematically equivalent) methods. Later sections will futher explain and demonstrate how to handle essential boundary conditions.

Boundary Function. A boundary condition of the form $u(L) = D$ can be implemented by demanding that $\varphi_i(L) = 0$, $i = 0, \ldots, N$, and adding a function $B(x)$ with the right boundary value: $B(L) = 0$. We then see that

$$u(x) = B(x) + \sum_{j=0}^{N} c_j\varphi_j(x)$$

gets the right value at $x = L$:

$$u(L) = B(L) + \sum_{j=0}^{N} c_j\varphi_j(L) = B(L) = D\,.$$

For any boundary where u is known we demand φ_i to vansh and construct a function $B(x)$ to attain the boundary value of u.

For example, with $u(0) = 0$ and $u(L) = D$ we can choose $B(x) = xD/L$, since $B(0) = 0$ and $B(L) = D$. The unknown function is then sought on the form

$$u(x) = \frac{x}{L}D + \sum_{j=0}^{N} c_j\varphi_j(x), \tag{2.50}$$

with $\varphi_i(0) = \varphi_i(L) = 0$. Exercise 2.4 encourages doing the computations to see how (2.50) modifies the results in the examples from Chapter 2.1.3.

Note that $B(x)$ can be chosen in many ways as long as its boundary values are correct. As an example, consider a domain $\Omega = [a, b]$ where the the boundary conditions are $u(a) = U_a$ and $u(b) = U_b$. A class of possible $B(x)$ functions is

$$B(x) = U_a + \frac{U_b - U_a}{(b-a)^p}(x-a)^p, \quad p \in \mathbb{R}\,.$$

In general we can formulate the procedure as follows. Let $\partial\Omega_E$ be the part(s) of the boundary $\partial\Omega$ of the domain Ω where u is specified. Set $\varphi_i = 0$ at the points in $\partial\Omega_E$. Seek u on the form

$$u(x) = B(x) + \sum_{j=0}^{N} c_j\varphi_j(x), \tag{2.51}$$

where $B(x)$ equals the boundary conditions on u at $\partial\Omega_E$. With the $B(x)$ term, u does not in general lie in $V = \text{span}\,\{\varphi_0, \ldots, \varphi_N\}$ anymore. Moreover, when a prescribed value of u at the boundary, say $u(a) = U_a$ is different from zero, it does not make sense to say that u lies in a vector space, because this space does not obey the requirements of addition and scalar multiplication. For example, $2u$ does not lie in the space since its boundary value is $2U_a$, which is incorrect. It only makes sense to split u in two parts, as done above, and have the unknown part $\sum_j c_j\varphi_j$ in a proper function space.

Construction of $B(x)$. An important special case arises for basis functions with the property

$$\varphi_i(x_j) = \begin{cases} 1, & i = j, \\ 0, & i \neq j, \end{cases}$$

when x_j is a boundary point. Examples on such functions are the Lagrange interpolating polynomials and finite element functions. With $\Omega = [x_0, x_N]$ and $u(x_0) = U_0$ and $u(x_N) = U_N$ we can then let

$$B(x) = U_0\varphi_0(x) + U_N\varphi_N(x)\,. \tag{2.52}$$

It is easily realized that $B(x_0) = U_0$ and $B(x_N) = U_N$, which is what we want. The unknown $u(x)$ applies the usual expansion, but with known boundary values,

$$u(x) = U_0\varphi_0(x) + U_N\varphi_N(x) + \sum_{j=1}^{N-1} c_j\varphi_j(x)\,. \tag{2.53}$$

If u is specified at only one boundary point, $B(x)$ contains just the term corresponding to that point and the sum $\sum_j c_j\varphi_j$ runs over the rest of the points. This construction of B can easily be generalized to two- and three-dimensional problems for which the construction is particularly powerful.

Modification of the Linear System.

Modification of the Element Matrix and Vector.

2.1.7 Boundary Conditions: Specified Derivative

Consider now the boundary conditions $u'(0) = E$ and $u(L) = D$ in the differential equation $-u''(x) = f(x)$. The latter boundary condition can be implemented by introducing $B(x) = xD/L$ and ensuring that $\varphi_i(L) = 0$, $i = 0, \ldots, N$. The former boundary condition involvng u' is implemented by the boundary terms that arise from integrating by parts. This will now be shown.

Starting with the Galerkin method,

$$\int_0^L (u''(x) + f(x))\varphi_i(x)dx = 0, \quad i = 0, \ldots, N,$$

integrating $u''\varphi_i$ by parts results in

$$\int_0^L u'(x)'\varphi_i'(x)dx - (u'(L)\varphi_i(L) - u'(0)\varphi_i(0)) = \int_0^L f(x)\varphi_i(x)dx, \quad i = 0, \ldots, N\,.$$

The first boundary term vanishes since $\varphi_i(L) = 0$ when $u(L)$ is known. The second term can be used to implement $u'(0) = E$, provided $\varphi_i(0) \neq 0$ for some i. The variational formulation then becomes

$$\int_0^L u'(x)\varphi_i'(x)dx + E\varphi_i(0) = \int_0^L f(x)\varphi_i(x)dx, \quad i = 0, \ldots, N\,.$$

Inserting

$$u(x) = B(x) + \sum_{j=0}^{N} c_j\varphi_j(x),$$

leads to the linear system

$$\sum_{j=0}^{N} \left(\int_0^L \varphi_i'(x)\varphi_j'(x)dx \right) c_j = \int_0^L (f(x)\varphi_i(x) - B'(x)\varphi_i'(x))\,dx - E\varphi_i(0), \tag{2.54}$$

for $\quad i = 0, \ldots, N$.

Example. Let us solve

$$-u''(x) = f(x), \quad x \in \Omega = [0,1], \quad u'(0) = E,\ u(1) = D,$$

using a polynomial basis. The exact solution is easily obtained by integrating twice and applying the boundary conditions:

$$u_{\mathrm{e}}(x) = 1 - x^2 + E(x-1) + D\,.$$

The requirements on φ_i is that $\varphi_i(1) = 0$, because u is specified at $x = 1$, so a proper set of polynomial basis functions are $\varphi_i(x) = (1-x)^{i+1}$, $i = 0, \ldots, N$. A suitable $B(x)$ function to handle the boundary condition $u(1) = D$ is $B(x) = xD$. The variational formulation is given by (2.54). With $N = 1$ we can calculate the global matrix system to be

$$\begin{pmatrix} 1 & 1 \\ 1 & 4/3 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 1 + D - E \\ 2/3 + D - E \end{pmatrix}$$

The solution becomes $c_0 = 2 + D - E$ and $c_1 = -1$, resulting in

$$u(x) = xD + (2 + D - E)(1 - x) - (1 - x^2), \tag{2.55}$$

which is actually equivalent to the exact solution of the problem.

2.1.8 Implementation

It is tempting to automate the computations in the previous example. A function similar to `least_squares` from Chapter 1.3.3 can easily be made. However, in the approximation problem the formulas for the entries in the linear system are fixed, while when we solve a differential equation the formulas are only known by the user of the function. We therefore require that the user prepares a function `integrand_lhs(phi, i, j)` for returning the integrand of the integral that contributes to matrix element (i, j). The `phi` variable is a Python dictionary holding the basis functions and their derivatives in symbolic form. That is, `phi[q]` is a list of

$$\{\frac{d^q\varphi_0}{dx^q}, \ldots, \frac{d^q\varphi_N}{dx^q}\}\,.$$

Similarly, `integrand_rhs(phi, i)` returns the integrand for element i in the right-hand side vector. Since we also have contributions to this vector (and potentially also the matrix) from boundary terms without any integral, we introduce two additional functions, `boundary_lhs(phi, i, j)` and `boundary_rhs(phi, i)` for returning terms in the variational formulation that are not to be integrated over the domain Ω.

The linear system can now be computed and solved symbolically by the following function:

```
def solve(integrand_lhs, integrand_rhs, phi, Omega,
          boundary_lhs=None, boundary_rhs=None):
    N = len(phi[0]) - 1
    A = sm.zeros((N+1, N+1))
    b = sm.zeros((N+1, 1))
    x = sm.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            integrand = integrand_lhs(phi, i, j)
            I = sm.integrate(integrand, (x, Omega[0], Omega[1]))
            if boundary_lhs is not None:
                I += boundary_lhs(phi, i, j)
            A[i,j] = A[j,i] = I
        integrand = integrand_rhs(phi, i)
        I = sm.integrate(integrand, (x, Omega[0], Omega[1]))
        if boundary_rhs is not None:
            I += boundary_rhs(phi, i)
        b[i,0] = I
    c = A.LUsolve(b)
    u = 0
    for i in range(len(phi[0])):
        print i
        u += c[i,0]*phi[0][i]
    return u
```

It turns out that symbolic solution of differential equations, discretized by a Galerkin method with global basis functions, is of limited interest beyond the simplest problems. Symbolic integration might be very time consuming or impossible, not only in `sympy` but also in *http://wolframalpha.com* (which applies the perhaps most powerful symbolic integration software available today: Mathematica). Numerical integration as an option is therefore desirable. The extended `solve` function below tries to combine the symbolic and numerical integration. The latter can be enforced by the user, or it can be invoked after a non-successful symbolic integration (being detected by an `Integral` object as the result of the integration, see also Chapter 1.3.8). Note that for a numerical integration, symbolic expressions must be converted to Python function (using `lambdify`), and the expressions cannot contain other symbols than `x`. The real `solve` routine in the `varform1D` module has error checking and meaningful error messages in such cases. The `solve` code below is a condensed version of the real one, with the purpose of showing how to automate the Galerkin method for solving differential equations in 1D with global basis functions:

```
def solve(integrand_lhs, integrand_rhs, phi, Omega,
          boundary_lhs=None, boundary_rhs=None, numint=False):
    N = len(phi[0]) - 1
    A = sm.zeros((N+1, N+1))
    b = sm.zeros((N+1, 1))
    x = sm.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            integrand = integrand_lhs(phi, i, j)
            if not numint:
                I = sm.integrate(integrand, (x, Omega[0], Omega[1]))
                if isinstance(I, sm.Integral):
                    numint = True  # force num.int. hereafter
            if numint:
                integrand = sm.lambdify([x], integrand)
                I = sm.mpmath.quad(integrand, [Omega[0], Omega[1]])
            if boundary_lhs is not None:
                I += boundary_lhs(phi, i, j)
            A[i,j] = A[j,i] = I
        integrand = integrand_rhs(phi, i)
        if not numint:
            I = sm.integrate(integrand, (x, Omega[0], Omega[1]))
            if isinstance(I, sm.Integral):
                numint = True
        if numint:
            integrand = sm.lambdify([x], integrand)
            I = sm.mpmath.quad(integrand, [Omega[0], Omega[1]])
        if boundary_rhs is not None:
            I += boundary_rhs(phi, i)
        b[i,0] = I
    print
    c = A.LUsolve(b)
    u = 0
    for i in range(len(phi[0])):
        u += c[i,0]*phi[0][i]
    return u
```

Is type(I) == type(sm.Integral) better than too much isinstance? Might be easier to read

Example: Constant Right-Hand Side. To demonstrate the code above, we address

$$-u''(x) = b, \quad x \in \Omega = [0,1], \quad u(1) = 1,\ u(0) = 0,$$

with b as a (symbolic) constant. A possible choice of space V, where the basis functions satisfy the requirements $\varphi_i(0) = \varphi_i(1) = 0$, is $V = \text{span}\,\{\varphi_i(x) = x^{i+1}(1-x)\}_{i=0}^{N}$. We also need a $B(x)$ function to take care of the known boundary values of u. Any function $B(x) = 1 - x^p$, $p \in \mathbb{R}$, is a candidate. One choice is $B(x) = 1 - x^3$. The unknown function is then written as a sum of a known and an unknown function:

$$u(x) = B(x) + \bar{u}(x), \quad \bar{u}(x) = \sum_{j=0}^{N} c_j \varphi_j(x)\,.$$

The appropriate variational formulation arises by multiplying the differential equation by v and integrate by parts, yielding

$$\int_0^1 u'v'dx = \int_0^1 fvdx \quad \forall v \in V,$$

and with $u = B + \bar{u}$,

$$\int_0^1 \bar{u}'v'dx = \int_0^1 (f - B')vdx \quad \forall v \in V\,. \tag{2.56}$$

Inserting $\bar{u} = \sum_j c_j\varphi_j$, we get the linear system

$$\sum_{j=0}^{N}\left(\int_0^1 \varphi_i'\varphi_j' dx\right) c_j = \int_0^1 (f - B')\varphi_i dx, \quad i = 0, \dots, N\,. \tag{2.57}$$

```
x, b = sm.symbols('x b')
f = b
B = 1 - x**3
dBdx = sm.diff(B, x)

N = 3
# Compute basis functions and their derivatives
phi = {0: [x**(i+1)*(1-x) for i in range(N+1)]}
for d in range(1, highest_derivative+1):
    phi[d] = [sm.diff(phi[0][i], x, d) for i in range(len(phi[0]))]

def integrand_lhs(phi, i, j):
    return phi[1][i]*phi[1][j]

def integrand_rhs(phi, i):
    return f*phi[0][i] - dBdx*phi[1][i]

Omega = [0, 1]

u_bar = solve(integrand_lhs, integrand_rhs, phi, Omega,
              boundary_lhs, boundary_rhs, verbose=verbose,
              numint=numint)
u = B + u_bar
print u
```

The exact solution, $u_e(x)$, can be obtained by integrating $f(x)$ twice and determine the two integration constants from the two boundary conditions. The following `sympy` code does the task:

```
f1 = sm.integrate(f, x)
f2 = sm.integrate(f1, x)
C1, C2 = sm.symbols('C1 C2')
u = -f2 + C1*x + C2
s = sm.solve([u.subs(x,0) - 1, u.subs(x,1) - 0], [C1, C2])
u_exact = -f2 + s[C1]*x + s[C2]
```

In this example, $u_e(x)$ is a parabola and our approximate u recovers the exact solution already for $N = 0$.

Example: Variable Right-Hand Side.

Example: Variable Coefficient $a(x)$.

2.1.9 Reference Element Computing with Finite Elements

We are now in a position to redo the finite element problem in Chapter 2.1.5 with more general boundary conditions and with an element by element computational algorithm. Let us address the model problem

$$-u''(x) = 2, \quad x \in \Omega = [0, L], \quad u(0) = b_0,\ u'(L) = b_L\,. \tag{2.58}$$

The exact solution reads $u_{\mathrm{e}}(x) = -x^2 + (b_L + 2L)x + b_0$. The test and trial space V must consists of finite element functions vanishing for $x = 0$ since $u(0)$ is specified as boundary condition. Numbering nodes from left to right through the domain, we let $B(x) = b_0\varphi_0$ and hence

$$u(x) = b_0\varphi_0(x) + \sum_{j=1}^{N} c_j\varphi_j(x)\,.$$

To derive the appropriate variational formulation, based on Galerkin's method, we first multiply the differential equation by a test function $v \in V$ and integrate over the domain:

$$-\int_0^L u''vdx = \int_0^L 2vdx \quad \forall v \in V\,.$$

Then we integrate by parts, use that $v(0) = 0$ in the boundary term, and insert the boundary condition $u'(L) = b_L$ in the other boundary term:

$$\int_0^L u'v'dx = \int_0^L 2dx + b_L v(L) \quad \forall v \in V\,. \tag{2.59}$$

For detailed calculations by hand we now insert the expansion for u and after some algebraic manipulations arrive at the linear system

$$\sum_{j=1}^{N}\left(\int_0^1 \varphi_i'\varphi_j'dx\right)dx = \int_0^1 (f - b_0\varphi_0)dx + b_L\varphi_i(L), \quad i = 0,\ldots,N\,. \tag{2.60}$$

The element by element computational procedure consists in splitting the integral to integrals over each element, and transforming each element integral to an integral over a reference element on $[-1,1]$ having X as coordinate. Chapters 1.4.4 and 1.4.5 explains the details.

However, there is one complicating factor arising here: we need to compute the derivatives $\varphi_i'(x)$ in the reference element. What we have on the reference element, is the expression $\tilde{\varphi}_r(X)$, $r = 0,\ldots,d$. We can easily compute $d\tilde{\varphi}_r/dX$, but what we need is $d\tilde{\varphi}_r(X)/dx$, which equals the desired factor $\varphi_{q(e,r)}(x)$ in the expressions for the matrix entries. By the chain rule we have that

$$\frac{d}{dx}\tilde{\varphi}_r(X) = \frac{d}{dX}\tilde{\varphi}_r(X)\frac{dX}{dx}\,.$$

From the mapping (1.58) it follows that

$$\frac{dX}{dx} = \frac{2}{h},$$

where h is the length of the current element. We hence have

$$\frac{d\tilde{\varphi}_r}{dx} = \frac{2}{h}\frac{d\tilde{\varphi}_r}{dX}\,. \tag{2.61}$$

The expressions for the entries in the element matrix and vector become

$$\tilde{A}^{(e)}_{r,s} = \int_{-1}^{1} \frac{d\tilde{\varphi}_r}{dx}\frac{d\tilde{\varphi}_s}{dx} \det J \, dX = \int_{-1}^{1} \frac{2}{h}\frac{d\tilde{\varphi}_r}{dX}\frac{2}{h}\frac{d\tilde{\varphi}_s}{dX}\frac{h}{2}\, dX, \tag{2.62}$$

$$\tilde{b}^{(e)}_{r} = \int_{-1}^{1} f(x(X))\tilde{\varphi}_r(X) \det J \, dX = \int_{-1}^{1} f(x(X))\tilde{\varphi}_r(X)\frac{2}{2}\, dX\,. \tag{2.63}$$

The specific derivatives of the basis functions with respect to X are easily computed. For P1 (linear, $d = 1$) elements we get

$$\tilde{\varphi}_0(X) = \frac{1}{2}(1 - X), \tag{2.64}$$

$$\frac{d\tilde{\varphi}_0}{dX} = -\frac{1}{2}, \tag{2.65}$$

$$\tilde{\varphi}_1(X) = \frac{1}{2}(1 + X), \tag{2.66}$$

$$\frac{d\tilde{\varphi}_1}{dX} = \frac{1}{2}\,. \tag{2.67}$$

The results for P2 (quadratic, $d = 2$) elements become

$$\tilde{\varphi}_0(X) = \frac{1}{2}(X - 1)X \tag{2.68}$$

$$\frac{d\tilde{\varphi}_0}{dX} = \frac{1}{2}(2X - 1)\,.\tilde{\varphi}_1(X) \qquad = 1 - X^2 \tag{2.69}$$

$$\frac{d\tilde{\varphi}_1}{dX} = -2X\,.\tilde{\varphi}_2(X) \qquad = \frac{1}{2}(X + 1)X\frac{d\tilde{\varphi}_2}{dX} = \frac{1}{2}(2X + 1)\,. \tag{2.70}$$

Let us calculate the element matrix from the formula (2.62) in case of P1 elements:

$$\tilde{A}^{(e)}_{0,0} = \int_{-1}^{1} \frac{2}{h}\frac{d\tilde{\varphi}_0}{dX}\frac{2}{h}\frac{d\tilde{\varphi}_0}{dX}\frac{h}{2}\, dX = \frac{2}{h}\int_{-1}^{1}\left(\frac{d\tilde{\varphi}_0}{dX}\right)^2 dX = \frac{2}{h}\int_{-1}^{1}\left(-\frac{1}{2}\right)^2 dX = \frac{1}{h}, \tilde{A}^{(e)}_{0,1} \quad = \frac{2}{h}\int_{-1}^{1}\frac{d\tilde{\varphi}_0}{dX}\frac{d\tilde{\varphi}_1}{dX}\, dX$$

$$\tilde{A}^{(e)}_{1,0} = \frac{2}{h}\int_{-1}^{1}\frac{d\tilde{\varphi}_1}{dX}\frac{d\tilde{\varphi}_0}{dX}\, dX = \frac{2}{h}\int_{-1}^{1}\frac{1}{2}(-\frac{1}{2})\, dX = -\frac{1}{h},$$

$$\tilde{A}^{(e)}_{1,1} = \frac{2}{h}\int_{-1}^{1}\frac{d\tilde{\varphi}_1}{dX}\frac{d\tilde{\varphi}_1}{dX}\, dX = \frac{2}{h}\int_{-1}^{1}\frac{1}{2}\frac{1}{2})\, dX = \frac{1}{h}\,.$$

These are valid expressions for any element but the first. For the first element, local 0 is on the boundary where u is known, and the corresponding coefficient c_0 is left out of the global system. We therefore only have one active node in the first element and consequently we form only the $A^{(0)}_{1,1}$ entry in the element matrix, which is treated as a 1×1 matrix.

For the right-hand side we achieve these results:

$$b_0^{(e)} = \int_{-1}^{1} 2\tilde{\varphi}_0(X)\det J\, dX = 2\frac{h}{2}\int_{-1}^{1}\frac{1}{2}(1-X)dX = h,$$
$$b_1^{(e)} = \int_{-1}^{1} 2\tilde{\varphi}_1(X)\det J\, dX = 2\frac{h}{2}\int_{-1}^{1}\frac{1}{2}(1+X)dX = h\,.$$

Also in the first element we compute only $b_1^{(0)}$ since the unknown associated with local 0 does not enter the global linear system. For the last element, $e = N-1$, we get an additional contribution from the boundary term $b_L\tilde{\varphi}_r(1)$ (which corresponds to $b_L\varphi_i(L)$, $i = q(e,r)$, $r = 0,1$):

$$b_0^{(e)} = h + b_L\tilde{\varphi}_0(1) = h, \quad b_1^{(e)} = h + b_L\tilde{\varphi}_1(1) = h + b_L \tag{2.71}$$

The various entries can be collected in a 2×2 element matrix and and 2–vector for all the internal elements (not having boundary nodes),

$$A^{(e)} = \frac{1}{h}\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}, \quad b^{(e)} = h\begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{2.72}$$

For the first element we have

$$A^{(e)} = \frac{1}{h}(1), \quad b^{(e)} = h(1), \tag{2.73}$$

while for the last element,

$$A^{(e)} = \frac{1}{h}\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}, \quad b^{(e)} = \begin{pmatrix} h \\ h + b_L \end{pmatrix}. \tag{2.74}$$

Assuming that all elements have the same length, we can now assemble these element matrices and vectors into the global matrix and vector in the linear system. We introduce a standard 1D mesh with nodes and elements numbered from left to right. The nodes are then $x_i = ih$, $i = 0,\ldots,N$, $h = L/N$, and elements are $\Omega^{(e)} = [x_e, x_{e+1}]$, $e = 0,\ldots,N$. Here is the result of applying the assembly algorithm from Chapter 1.4.4 to form the global coefficient matrix

$$A = \frac{1}{h}\begin{pmatrix}
2 & -1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
-1 & 2 & -1 & \ddots & & & & & \vdots \\
0 & -1 & 2 & -1 & \ddots & & & & \vdots \\
\vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\
\vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\
\vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\
\vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\
\vdots & & & & & \ddots & -1 & 2 & -1 \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & -1 & 1
\end{pmatrix} \tag{2.75}$$

The right-hand side becomes

$$b = \frac{1}{h}\begin{pmatrix} 2h \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \\ h + b_0 \end{pmatrix} \tag{2.76}$$

The unknown vector to be solved for is $c = (c_1, c_2, \ldots, c_N)^T$.

Looking at the prodct Ac, we may write the equation system on the following alternative form:

$$\frac{1}{h}(-c_{i-1} + 2c_i - c_{i+1}) = 2h, \quad i = 1, \ldots, N-1, \tag{2.77}$$

with $c_0 = b_0$, and

$$\frac{1}{h}(-c_{N-1} + c_N) = h + b_L \tag{2.78}$$

for the last equation.

Let us look at what a finite difference discretization of (2.58) will look like, using a standard second-order central difference for the second-order derivative:

$$\frac{-u_{i-1} + 2u_j - u_{i+1}}{h^2} = 2, \quad i = 1, \ldots, N-1, \tag{2.79}$$

where u_i is the value of u at mesh point i. We have assumed $N+1$ equally spaced mesh points, with spacing h. From the boundary condition $u(0) = b_0$, $u_0 = b_0$. The condition $u'(L) = b_L$ is discretized by

$$\frac{u_{N+1} - u_{N-1}}{2h} = b_L \quad \Leftrightarrow \quad u_{N+1} = u_{N-1} - 2hb_L,$$

which can be inserted in the difference equation for $i = N$, yielding

$$\frac{-2u_{N-1} + 2u_N}{h^2} = \frac{1}{h}2b_L + 2,$$

which upon multiplication by h equals

$$\frac{1}{h}(-u_{N-1} + u_N) = b_L + h\,. \tag{2.80}$$

Multiplying (2.79) by h reveals that the finite difference equations (2.79) and (2.80) are identical to the finite element equations (2.77) and (2.78). Recall that $c_i = u(x_i)$ which is denoted by u_i in the finite difference formulation.

In this (simple) test problem all the finite element machinery ends up with the same equations that a finite difference methods quickly provides. The next example shows that with a slightly more complicated $f(x)$ function, the finite element and difference methods differ in the representation of $f(x)$ in the difference equations.

A nice feature in the present example is that the exact solution fulfills the discrete equations, implying that the finite element or difference solution is exact, regardless of h. This is easily proved in a `sympy` session:

```
>>> from sympy import *
>>> i, b_L, b_0, h, N = symbols('i b_L b_0 h N')
>>> L = N*h
>>> x = i*h
>>> u_i = -x**2 + (b_L + 2*L)*x + b_0
>>> u_im1 = u_i.subs(i, i-1)
>>> u_ip1 = u_i.subs(i, i+1)
>>>
>>> # General equation
>>> R = 1/h**2*(-u_im1 + 2*u_i - u_ip1) - 2
>>> R = simplify(R)
>>> print R
0
>>>
>>> # Right boundary equation
>>> R = 1/h**2*(-u_im1 + u_i) - b_L/h - 1
>>> R = R.subs(i, N)
>>> R = simplify(R)
>>> print R
0
```

2.2 Examples on 1D Finite Element Computations

Other ways of treating boundary conditions.

Next: non-uniform partition, give result and make exercise.

Next: variable coefficient, numerical integration

Then: gamle Exer 2.7 symbolic when the software is ready :-)

Compulsory exercise: wave equation with P1 and P2 elements.

2.2.1 Variational Problems and Optimization of Functionals

2.2.2 Differential Equation Interpretation of Approximation

Suppose we are given the (trivial) equation

$$u(x) = f(x), \quad x \in \Omega = [0, L], \tag{2.81}$$

and think of it as a differential equation with a zero-th order derivative. We then want to solve this equation . Applying the Galerkin method, we get the variational formulation

$$(u, v) = (f, v) \quad \forall v \in V\,. \tag{2.82}$$

There are no boundary conditions associated with (2.81) because it only involves zero-th order derivatives. Using (2.82) for each of the basis functions φ_i of V, we get

$$(u, \varphi_i) = (f, \varphi_i), \quad i = 0, \ldots, N, \tag{2.83}$$

which, when u is expanded in basis functions, $u = \sum_{j=0}^{N} c_j \varphi_j$, leads to the linear system

$$\sum_{j=0}^{N} (\varphi_i, \varphi_j) c_j = (f, \varphi_i), \quad i = 0, \ldots, N\,. \tag{2.84}$$

This is the same linear system as we derive when we seek a least-squares best approximation u to f, the only difference being that we now start with an equation (2.81) rather than an approximation problem.

Computation of $(f, \varphi_i) = \int_\Omega f \varphi_i dx$ depends, of course, on what f is. However, using the Trapezoidal method of integration, we can get a simple expression for (f, φ_i) when ϕ_i is a finite element basis function:

$$(f, \varphi_i) = h f_i, \tag{2.85}$$

if i is an internal node and h is the constant element length.

Applying finite element basis functions to compute the matrix entries (φ_i, φ_j) leads to the following linear system:

$$\frac{h}{6}(u_{i-1} + 4u_i + u_{i+1}) = h f_i, \quad i = 1, \ldots, N-1\,. \tag{2.86}$$

The first and last equation, corresponding to $i = 0$ and $i = N$ is slightly different, see Chapter 1.4.9.

The left-hand side of (2.86) can be manipulated to equal

$$h(u_i - \frac{1}{6}(-u_{i-1} + 2u_i - u_{i+1}))\,. \tag{2.87}$$

Thinking in terms of finite differences and difference operators, this is nothing but the standard discretization of $h(u - \frac{h^2}{6}u'')$. This implies that solving $u = f$ by the finite element method with P1 elements is equivalent to solving

$$u + \frac{h^2}{6}u'' = f, \quad u'(0) = u'(L) = 0, \tag{2.88}$$

by the finite difference method:

$$[u + \frac{h^2}{6} DxDxu = f]_i\,. \tag{2.89}$$

The finite difference method applied directly to (2.81) would have given $u_i = f_i$. We therefore see that the finite element method introduce some kind of smoothing (compared to the finite difference method) as the u term in the equation is replaced by $u - ku''$ for a small constant $k = h^2/6$.

Exercises to investigate this if f is rough. Also by exact integration of a rough f. High freq sine.

Exercise: Computation of $(f,\varphi_i)=\int_\Omega f\varphi_i dx$ depends, of course, on what f is. Some more insight into what the right-hand sidee looks like can be obtained by replacing f by a finite element function which interpolates f. That is, we expand f as $f\approx\sum_j d_j\varphi_j$, and determine d_j from the fact that the expansion shall equal f at the nodes. Then $d_j=f(x_j)$, which we denote by f_j, using finite difference notation. A similar notation is introduced for c_j: since $c_j=u(x_j)$ we use u_j for c_j. The right-hand side of the equation system becomes

$$(\sum_j f_j\varphi_j,\varphi_i)=\sum_{j=0}^{N}(\varphi_i,\varphi_j)f_j\,.$$

This is the same form as the left-hand side. With $A_{i,j}(\varphi_i,\varphi_j)$

2.3 The Finite Element Method in 2D and 3D

The real power of the finite element method first becomes evident when we want to solve partial differential equations posed on two- and three-dimensional domains of non-trivial geometric shape. As in 1D, the domain Ω is divided into n_e non-overlapping elements. The elements have simple shapes: triangles and quadrilaterals are popular in 2D, see Figures 2.1 and 2.2, while tetrahedra and box-shapes elements dominate in 3D. The approximate solution u of the PDE is as usual expressed as $u=\sum_{j=0}^{N}c_j\varphi_j$, where the φ_i functions, as in 1D, are polynomials over each element. The Galerkin method or the method of weighted residuals are used, together with integration by parts, to transform the PDE problem to a variational formulation, which then leads to a linear system of algebraic equations (if the PDE is linear). The integrals in the variational formulation are split into contributions from each element, and these contributions are calculated by mapping elements to a reference element.

We will illustrate the extension of the finite element method from 1D to 2D and 3D by invoking specific examples and from these arrive at a general finite element algorithm applicable to PDEs in any dimension.

Our primary model problem will be the Poisson equation

$$-\nabla\cdot(\alpha\nabla u)=f,\qquad \text{in } \Omega, \tag{2.90}$$

$$u=g_E,\quad \text{on } \partial\Omega_E \tag{2.91}$$

$$-\alpha\frac{\partial u}{\partial n}=g_N,\quad \text{on } \partial\Omega_N \tag{2.92}$$

$$\tag{2.93}$$

Here, u, α, f, g_E, and g_N are all functions of the spatial coordiantes $\boldsymbol{x}=(x_0,\ldots,x_{d-1})$ where d here denotes the number of space dimensions. In 2D we will sometimes also use x and y instead of x_0 and x_1. The boundary $\partial\Omega$ of the domain Ω is divided into two non-overlapping parts: $\partial\Omega_D$ where u is specified as the function g_E, and $\partial\Omega_N$ where the flux of u, $\alpha\partial u/\partial n$ is known.

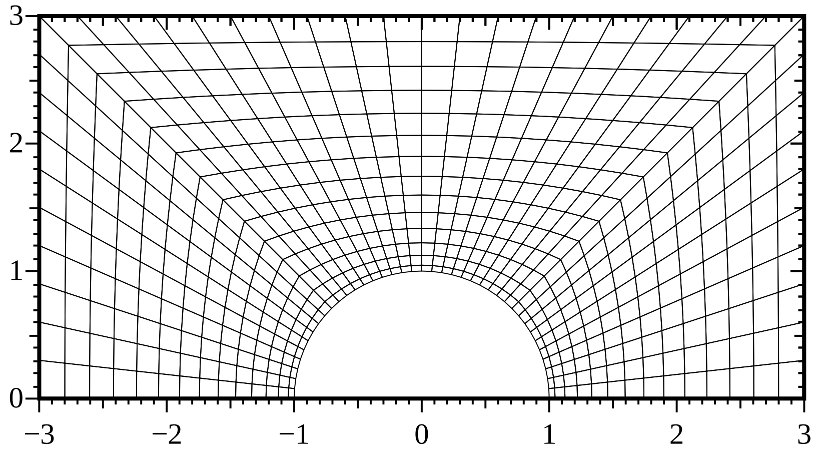


Fig. 2.1. Example of a 2D finite element mesh with quadrilateral elements.

2.3.1 Variational Formulation

2.4 Exercises

Exercise 2.1. Use a polynomial basis functions for problem (2.1.4).

For the model problem (2.1.4) with $D = 0$, use $\varphi_i = x^{i+1}(L - x)$, $i = 0, \dots, N$ and derive the linear systems corresponding to the least-squares, Galerkin, and collocation methods. ◇

Exercise 2.2. Solve the linear system from Exer. 2.1.

Choose $N = 0$ in the linear systems derived in Exercise 2.1 and the solve the systems. Compare the numerical and the exact solutions. ◇

Exercise 2.3. Use integration by parts.

Formulate a Galerkin method for the model problem (2.1.4) by using integration by parts as explained in Chapter 2.1.4. Apply the sine basis functions (2.28) and demonstrate that the solution is equivalent to the case where integration by parts is not used (see (2.36)). ◇

Exercise 2.4. Extend the examples in Chap. 2.1.3 with $u(L) = D$.

Modify the computations in Chapter 2.1.3 for the case where $u(L) = D \neq 0$. Use (2.50) and set up the general solutions corresponding to the least-squares and Galerkin methods. Find the solution for $N = 0$ in the collocation method. Compare the numerical and exact solutions in a plot. ◇

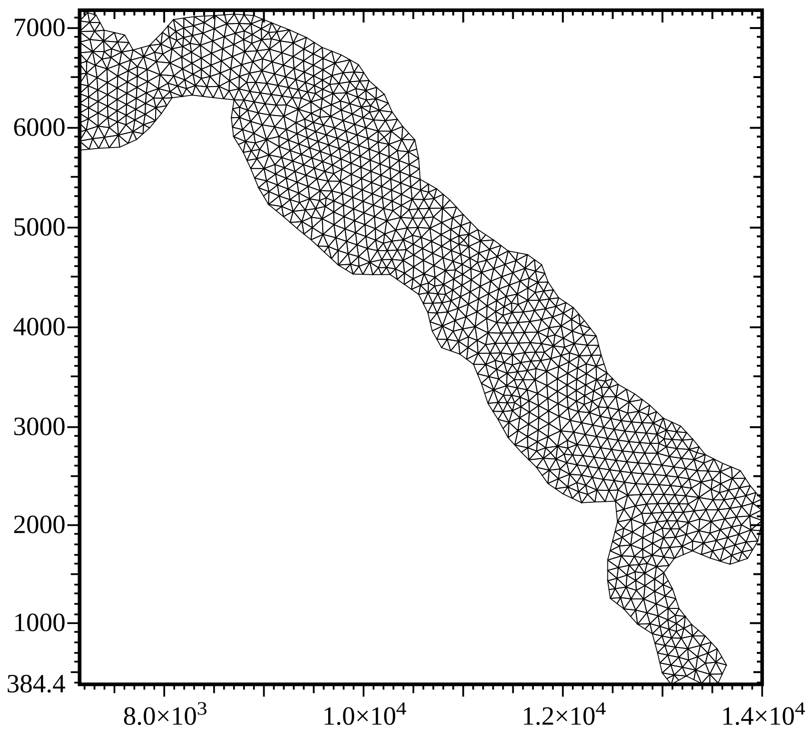


Fig. 2.2. Example of a 2D finite element mesh with triangular elements.

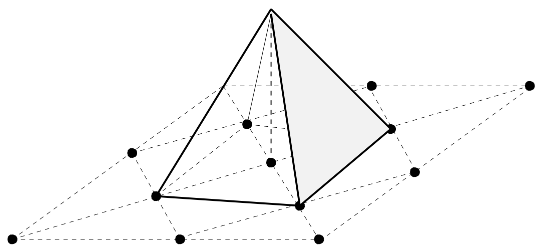


Fig. 2.3. Sketch of a typical piecewise linear basis function over a patch of linear triangular elements.

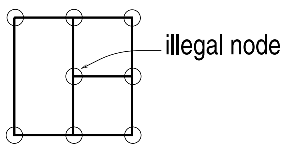


Fig. 2.4. Illustration of a hanging node, which is illegal unless the standard finite element method is modified to treat such nodes.

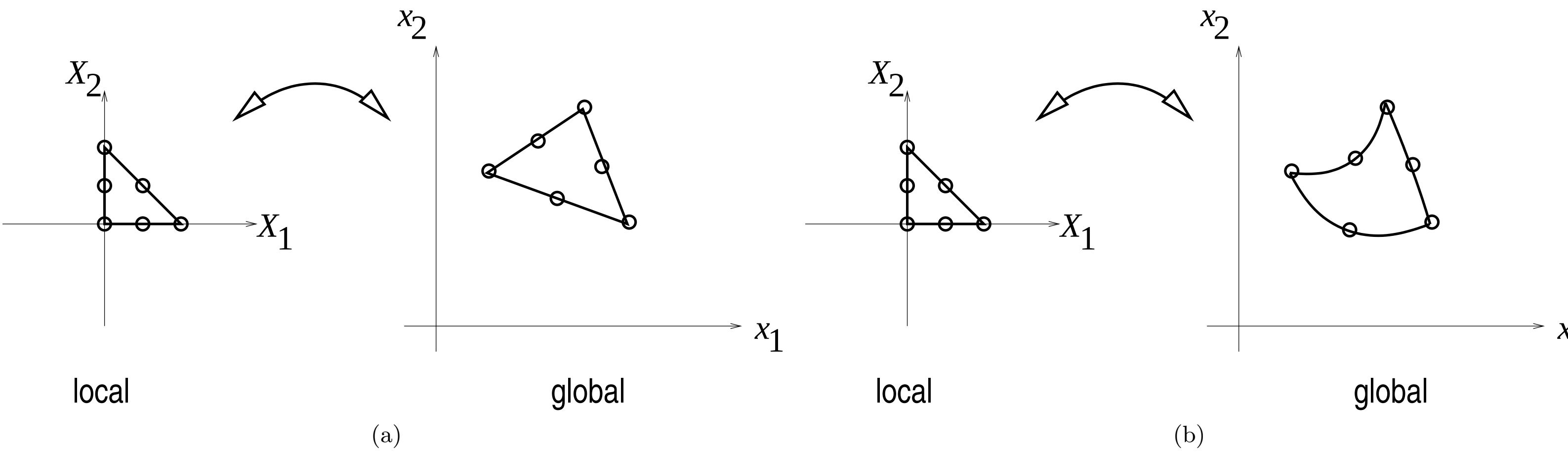


Fig. 2.5. Sketch of a two-dimensional, second-order triangular (P2) elements: (a) affine mapping resulting in straight sides; (b) isoparametric mapping resulting in sides with shapes of parabolas.

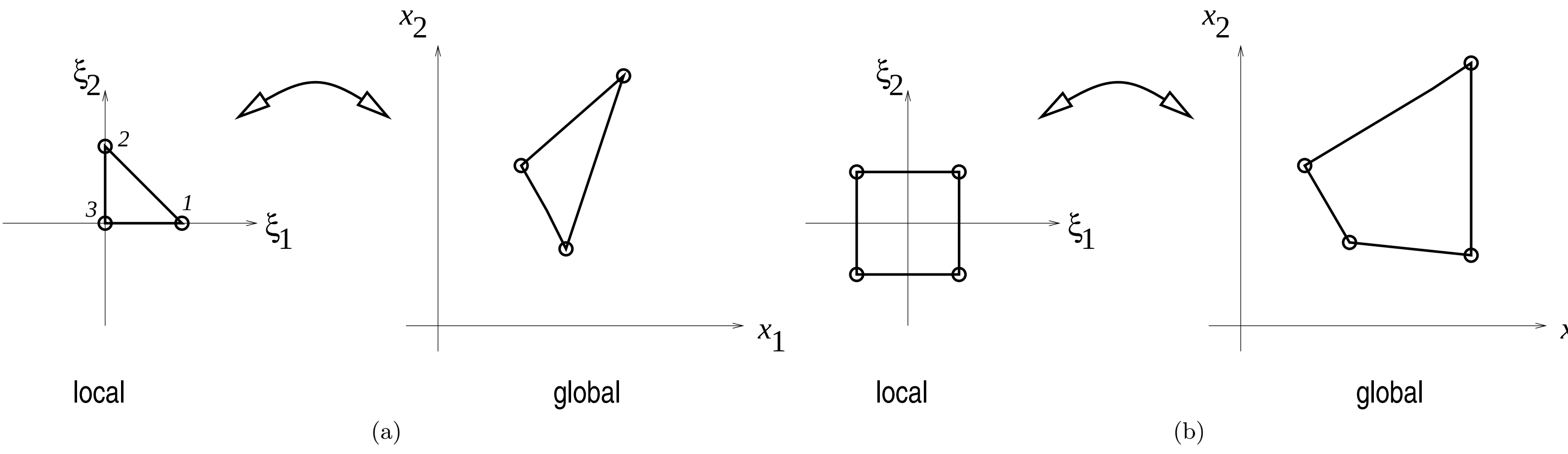


Fig. 2.6. Sketch of a two-dimensional elements with straight sides: (a) standard triangular P1 element with linear basis functions; (b) quadrilateral Q1 element with bilinear basis functions and isoparametric mapping.

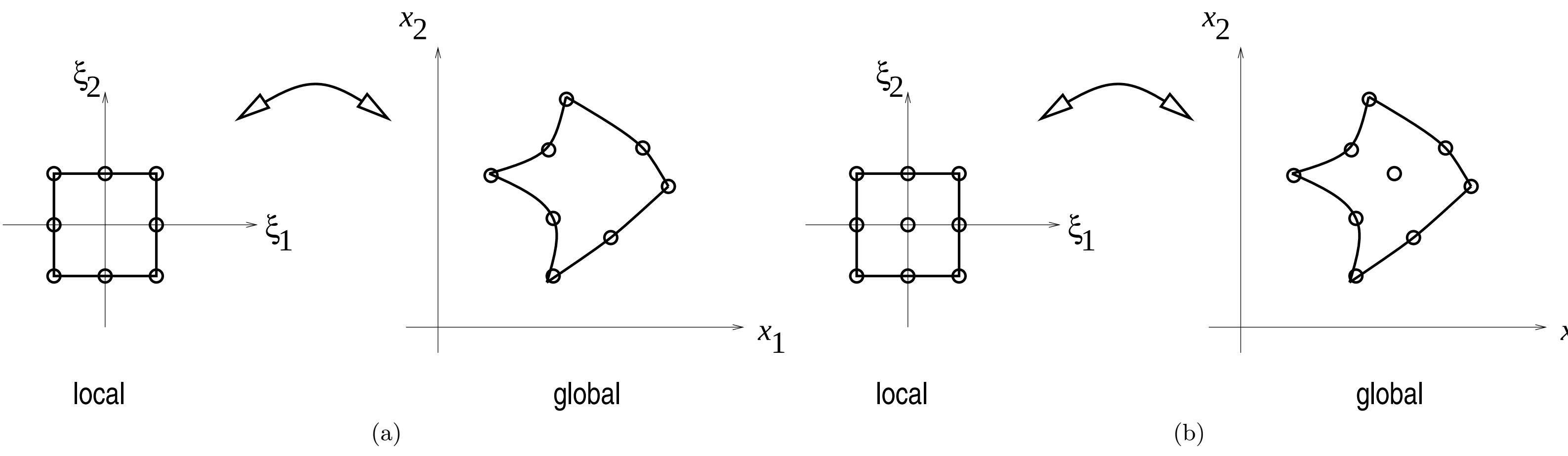


Fig. 2.7. Sketch of a two-dimensional, second-order, quadrilateral elements with isoparametric mapping: (a) 8 nodes; (b) 9 nodes.

Exercise 2.5. Use a nonzero boundary condition in Exer. 2.1.

Let $u(L) = D$ in Exercise 2.1 and derive the linear systems in this case. Find $u(x)$ for $N = 0$ and compare with the exact solution. ◇