Basic finite element methods

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Nov 8, 2012

Note: **QUITE PRELIMINARY VERSION**

Contents

List of exercises

The finite element method is a powerful tool for solving differential equations, especially in complicated domains and where higher-order approximations are desired. Figure 1 shows a two-dimensional domain with a non-trivial geometry. The idea is to divide the domain into triangles (elements) and seek a polynomial approximations to the unknown functions on each triangle. The method glues these piecewise approximations together to find a global solution. Linear and quadratic polynomials over the triangles are particularly popular.

Many successful numerical methods for differential equations, including the finite element method, aim at approximating the unknown function by a sum

$$u(x) = \sum_{i=0}^{N} c_i \varphi_i(x), \tag{1}$$

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)

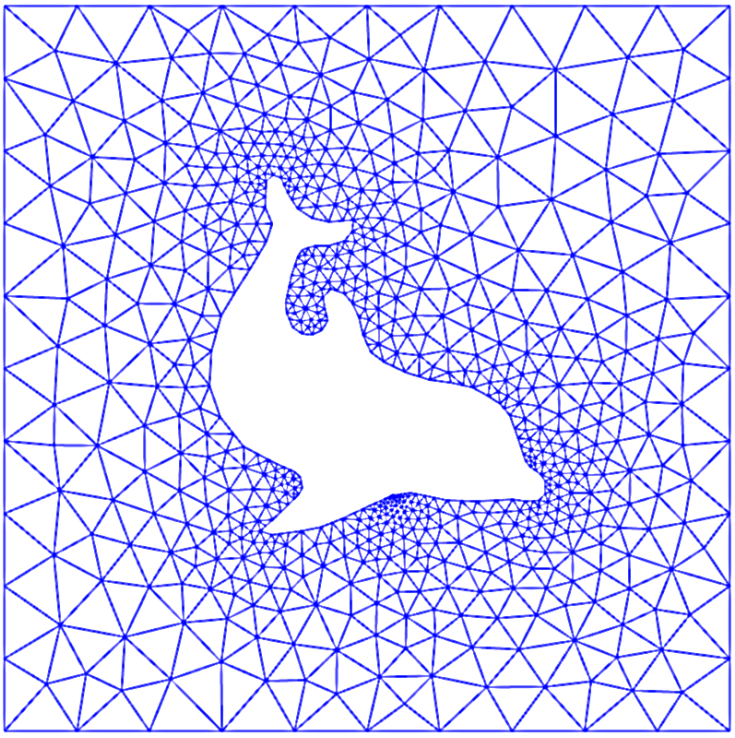

Figure 1: Domain for flow around a dolphin.

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 the solution $c_0, \ldots, c_N$ of the $N+1$ equations.

Especially in the finite element method, the machinery for constructing the discrete equations to be implemented on a computer is quite comprehensive, with many mathematical and implementational details entering the scene at the same time. From an ease-of-learning perspective it can therefore be wise to introduce the computational machinery for a trivial equation: $u = f$. Solving this equation with f given and u on the form (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 demanding topics related to differential equations (e.g., integration by parts, boundary conditions, and coordinate mappings). This is the reason why we shall first 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 or L_2 projection method, and 3) interpolation or collocation.

1 Approximation of vectors

We shall start with introducing two fundamental methods for determining the coefficients c_i in (1) and illustrate the methods on approximation of vectors, because vectors in vector spaces is more intuitive than working with functions in function spaces. The extension from vectors to functions will be trivial as soon as the fundamental ideas are understood.

The first method of approximation is called the *least squares method* and consists in finding c_i such that the difference $u - f$, measured in some norm, is minimized. That is, we aim at finding the best approximation u to f (in some norm). The second method is not as intuitive: we find u such that the error $u - f$ is orthogonal to the space where we seek u. This is known as *projection*, or we may also call it a *Galerkin method*. When approximating vectors and functions, the two methods are equivalent, but this is no longer the case when working with differential equations.

1.1 Approximation of planar vectors

Suppose we have given a vector $\boldsymbol{f} = (3, 5)$ in the xy plane and that we want to approximate this vector by a vector aligned in the direction of the vector (a, b). Figure 2 depicts the situation.

We introduce the vector space V spanned by the vector $\boldsymbol{\varphi}_0 = (a, b)$:

$$V = \text{span}\,\{\boldsymbol{\varphi}_0\}\,. \tag{2}$$

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

$$(\boldsymbol{u}, \boldsymbol{v}) = u_0v_0 + u_1v_1\,. \tag{3}$$

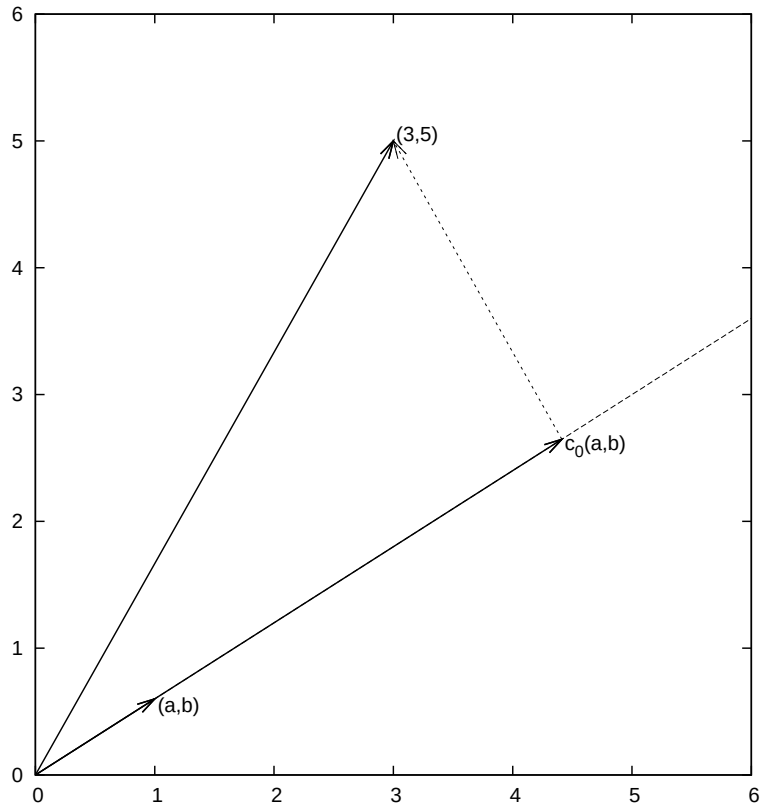

Figure 2: Approximation of a two-dimensional vector by a one-dimensional vector.

Remark 1. 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.

Remark 2. It might be wise to refresh some basic linear algebra by consulting a textbook. Exercises 1 and 2 suggest specific tasks to regain familiarity with fundamental operations on inner product vector spaces.

The least squares method. 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

$$E(c_0) = (\boldsymbol{e}, \boldsymbol{e}) = (\boldsymbol{f} - c_0\boldsymbol{\varphi}_0, \boldsymbol{f} - c_0\boldsymbol{\varphi}_0)\,. \tag{4}$$

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

$$E(c_0) = (\boldsymbol{f}, \boldsymbol{f}) - 2c_0(\boldsymbol{f}, \boldsymbol{\varphi}_0) + c_0^2(\boldsymbol{\varphi}_0, \boldsymbol{\varphi}_0)\,. \tag{5}$$

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

$$(\alpha\boldsymbol{u}, \boldsymbol{v}) = \alpha(\boldsymbol{u}, \boldsymbol{v}), \quad \alpha \in \mathbb{R}, \tag{6}$$

$$(\boldsymbol{u} + \boldsymbol{v}, \boldsymbol{w}) = (\boldsymbol{u}, \boldsymbol{w}) + (\boldsymbol{v}, \boldsymbol{w}), \tag{7}$$

$$(\boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{v}, \boldsymbol{u})\,. \tag{8}$$

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

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

Differentiating (5) with respect to c_0 gives

$$\frac{\partial E}{\partial c_0} = -2(\boldsymbol{f}, \boldsymbol{\varphi}_0) + 2c_0(\boldsymbol{\varphi}_0, \boldsymbol{\varphi}_0)\,. \tag{9}$$

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

$$c_0 = \frac{(\boldsymbol{f}, \boldsymbol{\varphi}_0)}{(\boldsymbol{\varphi}_0, \boldsymbol{\varphi}_0)}, \tag{10}$$

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

$$c_0 = \frac{3a + 5b}{a^2 + b^2}\,. \tag{11}$$

For later, it is worth mentioning that setting the key equation (9) to zero can be rewritten as

$$(\boldsymbol{f} - c0\boldsymbol{\varphi}_0, \boldsymbol{\varphi}_0) = 0,$$

or

$$(\boldsymbol{e}, \boldsymbol{\varphi}_0) = 0\,. \tag{12}$$

The Galerkin or projection method. Minimizing $||\boldsymbol{e}||^2$ implies that $\boldsymbol{e}$ is orthogonal to *any* vector $\boldsymbol{v}$ in the space V. This result is visually quite clear from Figure 2 (think of other vectors along the line (a, b): all of them will lead to a larger distance between the approximation and $\boldsymbol{f}$). To see this result mathematically, we express any $\boldsymbol{v} \in V$ as $\boldsymbol{v} = s\boldsymbol{\varphi}_0$ for any scalar parameter s, recall that two vectors are orthogonal when their inner product vanishes, and calculate the inner product

$$\begin{aligned}
(\boldsymbol{e}, s\boldsymbol{\varphi}_0) &= (\boldsymbol{f} - c_0\boldsymbol{\varphi}_0, s\boldsymbol{\varphi}_0) \\
&= (\boldsymbol{f}, s\boldsymbol{\varphi}_0) - (c_0\boldsymbol{\varphi}_0, s\boldsymbol{\varphi}_0) \\
&= s(\boldsymbol{f}, \boldsymbol{\varphi}_0) - sc_0(\boldsymbol{\varphi}_0, \boldsymbol{\varphi}_0) \\
&= s(\boldsymbol{f}, \boldsymbol{\varphi}_0) - s\frac{(\boldsymbol{f}, \boldsymbol{\varphi}_0)}{(\boldsymbol{\varphi}_0, \boldsymbol{\varphi}_0)}(\boldsymbol{\varphi}_0, \boldsymbol{\varphi}_0) \\
&= s\left((\boldsymbol{f}, \boldsymbol{\varphi}_0) - (\boldsymbol{f}, \boldsymbol{\varphi}_0)\right) \\
&= 0\,.
\end{aligned}$$

Therefore, instead of minimizing the square of the norm, we could demand that $\boldsymbol{e}$ is orthogonal to any vector in V. This approach is known as *projection*, because we it is the same as projecting the vector onto the subspace. We may also use the term *Galerkin's method*. Mathematically the approach is stated by the equation

$$(\boldsymbol{e}, \boldsymbol{v}) = 0, \quad \forall \boldsymbol{v} \in V\,. \tag{13}$$

Since an arbitrary $\boldsymbol{v} \in V$ can be expressed as $s\boldsymbol{\varphi}_0$, $s \in \mathbb{R}$, (13) implies

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

which means that the error must be orthogonal to the basis vector in the space V:

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

The latter equation gives (10) for c_0. Furthermore, the latter equation also arose from least squares computations in (12).

1.2 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

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

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.

The least squares method. 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 $c_0, \ldots, c_N$,

$$\begin{aligned} E(c_0, \ldots, c_N) &= (\boldsymbol{e}, \boldsymbol{e}) = (\boldsymbol{f} - \sum_j c_j \boldsymbol{\varphi}_j, \boldsymbol{f} - \sum_j c_j \boldsymbol{\varphi}_j) \\ &= (\boldsymbol{f}, \boldsymbol{f}) - 2\sum_{j=0}^{N} c_j (\boldsymbol{f}, \boldsymbol{\varphi}_j) + \sum_{p=0}^{N}\sum_{q=0}^{N} c_p c_q (\boldsymbol{\varphi}_p, \boldsymbol{\varphi}_q)\,. \end{aligned} \tag{14}$$

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 (14) is differentiated as follows:

$$\frac{\partial}{\partial c_i} \sum_{j=0}^{N} c_j (\boldsymbol{f}, \boldsymbol{\varphi}_j) = (\boldsymbol{f}, \boldsymbol{\varphi}_i), \tag{15}$$

since the expression to be differentiated is a sum and only one term, $c_i(\boldsymbol{f}, \boldsymbol{\varphi}_i)$, contains c_i and this term is linear in c_i. To understand this differentiation in detail, write out the sum specifically for, e.g, $N = 3$ and $i = 1$.

The last term in (14) is more tedious to differentiate. We start with

$$\frac{\partial}{\partial c_i} c_p c_q = \begin{cases} 0, & \text{if } p \neq i \text{ and } q \neq i, \\ c_q, & \text{if } p = i \text{ and } q \neq i, \\ c_p, & \text{if } p \neq i \text{ and } q = i, \\ 2c_i, & \text{if } p = q = i, \end{cases} \tag{16}$$

Then

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

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

$$\frac{\partial}{\partial c_i}\sum_{p=0}^{N}\sum_{q=0}^{N} c_p c_q(\boldsymbol{\varphi}_p,\boldsymbol{\varphi}_q) = 2\sum_{j=0}^{N} c_i(\boldsymbol{\varphi}_j,\boldsymbol{\varphi}_i)\,. \tag{17}$$

It then follows that setting

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

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

$$\sum_{j=0}^{N} A_{i,j}c_j = b_i, \quad i = 0,\ldots,N, \tag{18}$$

where

$$A_{i,j} = (\boldsymbol{\varphi}_i,\boldsymbol{\varphi}_j), \tag{19}$$
$$b_i = (\boldsymbol{\varphi}_i,\boldsymbol{f})\,. \tag{20}$$

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

The Galerkin or projection method. In analogy with the "one-dimensional" example in Section 1.1, 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 $\boldsymbol{v}\in V$:

$$(\boldsymbol{e},\boldsymbol{v}) = 0, \quad \forall \boldsymbol{v}\in V\,. \tag{21}$$

Since any $\boldsymbol{v}\in V$ can be written as $\boldsymbol{v} = \sum_{i=0}^{N} c_i\boldsymbol{\varphi}_i$, the statement (21) 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,

$$(\boldsymbol{e},\boldsymbol{\varphi}_i) = 0, \quad i = 0,\ldots,N\,. \tag{22}$$

These $N+1$ equations result in the same linear system as (18):

$$(\boldsymbol{f} - \sum_{j=0}^{N} c_j\boldsymbol{\varphi}_j, \boldsymbol{\varphi}_i) = (\boldsymbol{f}, \boldsymbol{\varphi}_i) - \sum_{j=0}^{N} (\boldsymbol{\varphi}_i, \boldsymbol{\varphi}_j)c_j = 0,$$

and hence

$$\sum_{j=0}^{N} (\boldsymbol{\varphi}_i, \boldsymbol{\varphi}_j)c_j = (\boldsymbol{f}, \boldsymbol{\varphi}_i), \quad i = 0, \ldots, N\,.$$

So, instead of differentiating the $E(c_0, \ldots, c_N)$ function, we could simply use (21) as the principle for determining $c_0, \ldots, c_N$, resulting in the $N+1$ equations (22).

The names *least squares method* or *least squares approximation* are natural since the calculations consists of minimizing $||\boldsymbol{e}||^2$, and $||\boldsymbol{e}||^2$ is a sum of squares of differences between the components in $\boldsymbol{f}$ and $\boldsymbol{u}$. We find $\boldsymbol{u}$ such that this sum of squares is minimized.

The principle (21), or the equivalent form (22), is known as *projection*. Almost the same mathematical idea was used by the Russian mathematician Boris Galerkin[1] to solve differential equations, resulting in what is widely known as *Galerkin's method*.

2 Approximation of 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:

$$u = \sum_{j=0}^{N} c_j\varphi_j\,. \tag{23}$$

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 physical spaces. The approximation (23) is typically used to discretize a problem in space. Other methods, most notably finite differences, are common for time discretization (although the form (23) can be used in time too).

2.1 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

[1] `http://en.wikipedia.org/wiki/Boris_Galerkin`

in Section 1.2. 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

$$(f, g) = \int_\Omega f(x)g(x)\, dx\,. \tag{24}$$

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

$$E = (f(x) - \sum_{j=0}^{N} c_j\varphi_j(x), f(x) - \sum_{j=0}^{N} c_j\varphi_j(x))\,. \tag{25}$$

Note the analogy with (14): 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 (25), through similar steps as used for the result (14), leading to

$$E(c_0, \ldots, c_N) = (f, f) - 2\sum_{j=0}^{N} c_j(f, \varphi_i) + \sum_{p=0}^{N}\sum_{q=0}^{N} c_p c_q(\varphi_p, \varphi_q)\,. \tag{26}$$

Minimizing this function of $N + 1$ scalar variables $c_0, \ldots, c_N$ requires differentiation with respect to c_i, for $i = 0, \ldots, N$. The resulting equations are very similar to those we had in the vector case, and we hence end up with a linear system of the form (18), with

$$A_{i,j} = (\varphi_i, \varphi_j), \tag{27}$$
$$b_i = (f, \varphi_i)\,. \tag{28}$$

2.2 The Galerkin or projection method

As in Section 1.2, the minimization of (e, e) is equivalent to

$$(e, v) = 0, \quad \forall v \in V\,. \tag{29}$$

This is known as a projection of a function f onto the subspace V. We may also call it a Galerkin method for approximating functions. Using the same reasoning as in (21)-(22), it follows that (29) is equivalent to

$$(e, \varphi_i) = 0, \quad i = 0, \ldots, N\,. \tag{30}$$

Inserting $e = f - u$ in this equation and ordering terms, as in the multi-dimensional vector case, we end up with a linear system with a coefficient matrix (27) and right-hand side vector (28).

Whether we work with vectors in the plane, general vectors, or functions in function spaces, the least squares principle and the Galerkin or projection method are equivalent.

2.3 Example: linear approximation

Let us apply the theory in the previous section to a simple problem: given a parabola $f(x) = 10(x-1)^2 - 1$ 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

$$A_{0,0} = (\varphi_0, \varphi_0) = \int_1^2 1 \cdot 1\, dx = 1, \tag{31}$$

$$A_{0,1} = (\varphi_0, \varphi_1) = \int_1^2 1 \cdot x\, dx = 3/2, \tag{32}$$

$$A_{1,0} = A_{0,1} = 3/2, \tag{33}$$

$$A_{1,1} = (\varphi_1, \varphi_1) = \int_1^2 x \cdot x\, dx = 7/3\,. \tag{34}$$

The corresponding right-hand side is

$$b_1 = (f, \varphi_0) = \int_1^2 (10(x-1)^2 - 1) \cdot 1\, dx = 7/3, \tag{35}$$

$$b_2 = (f, \varphi_1) = \int_1^2 (10(x-1)^2 - 1) \cdot x\, dx = 13/3\,. \tag{36}$$

Solving the linear system results in

$$c_0 = -38/3, \quad c_1 = 10, \tag{37}$$

and consequently

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

Figure 3 displays the parabola and its best approximation in the space of all linear functions.

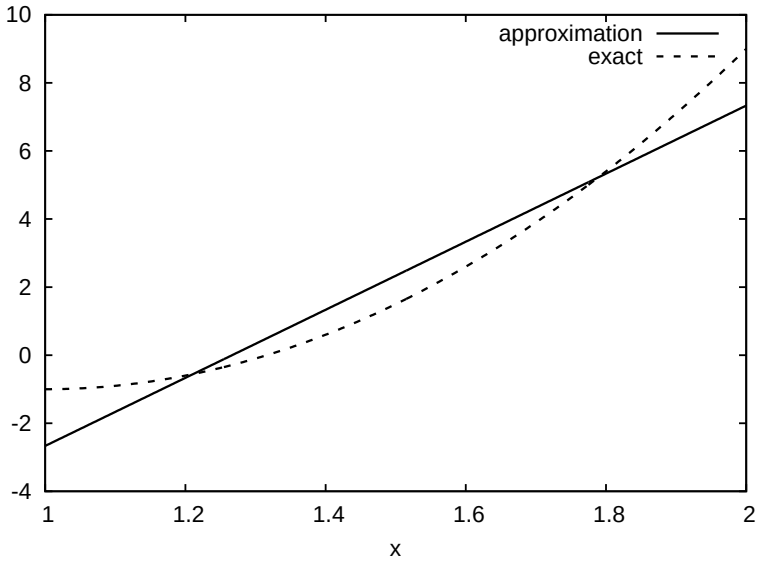

Figure 3: Best approximation of a parabola by a straight line.

2.4 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 noticeable 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.pdf'):
    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`.

2.5 Perfect approximation

Let us use the code above to recompute the problem from Section 2.3 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

```
>>> from approx1D import *
>>> x = sm.Symbol('x')
>>> f = 10*(x-1)**2-1
>>> u = least_squares(f=f, phi=[1, x, x**2], Omega=[1, 2])
>>> print u
10*x**2 - 20*x + 9
>>> print sm.expand(f)
10*x**2 - 20*x + 9
```

Now, what if we use $\phi_i(x) = x^i$ for $i = 0, \ldots, N = 40$? The output from `least_squares` gives $c_i = 0$ for $i > 2$. In fact, we have a general result that if

$f \in V$, the least squares and Galerkin/projection methods compute the exact solution $u = f$.

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 $i = 0, \ldots, N$.

2.6 Ill-conditioning

The computational example in Section 2.5 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. Below is a table showing the solution of the linear system arising from approximating a parabola by functions on the form $u(x) = \sum_{j=0}^{N} c_j x^j$, $N = 10$. Analytically, we know that $c_j = 0$ for $j > 2$, but ill-conditioning may produce $c_j \neq 0$ for $j > 2$.

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

The exact value of c_j, $j = 0, \ldots, 10$, appears in the first column while the other columns correspond to results obtained by three different methods:

- Column 2: The matrix and vector are converted to the data structure `sympy.mpmath.fp.matrix` 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 for i not very small. This phenomenon is illustrated in Figure 4. There are 15 lines in this figure, but only half of them are visually distinguishable. 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 the table above, 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 polynomials. The functions used in the finite element methods are almost orthogonal, and this property helps to avoid problems with solving matrix systems. 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.

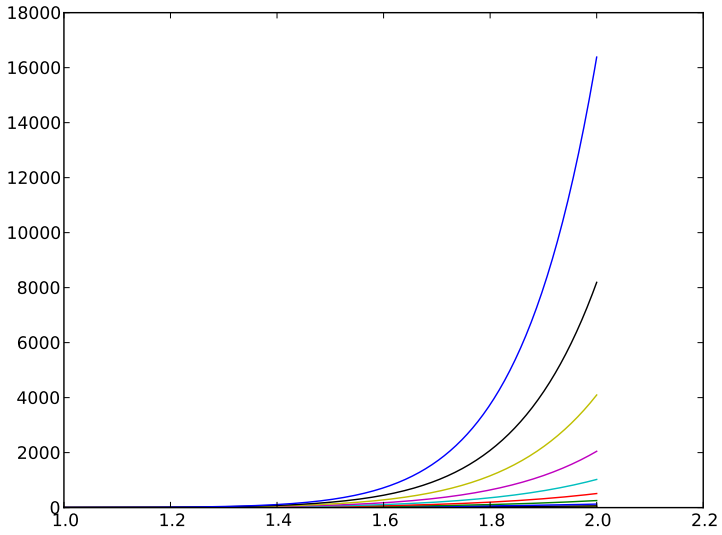

Figure 4: The 15 first basis functions x^i, $i = 0, \ldots, 14$.

2.7 Fourier series

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

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

That is,

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

An approximation to the $f(x)$ function from Section 2.3 can then be computed by the `least_squares` function from Section 2.4:

```
N = 3
from sympy import sin, pi
x = sm.Symbol('x')
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 5 (left) shows the oscillatory approximation of $\sum_{j=0}^{N} c_j \sin((j+1)\pi x)$ when $N = 3$. Changing N to 11 improves the approximation considerably, see Figure 5 (right).

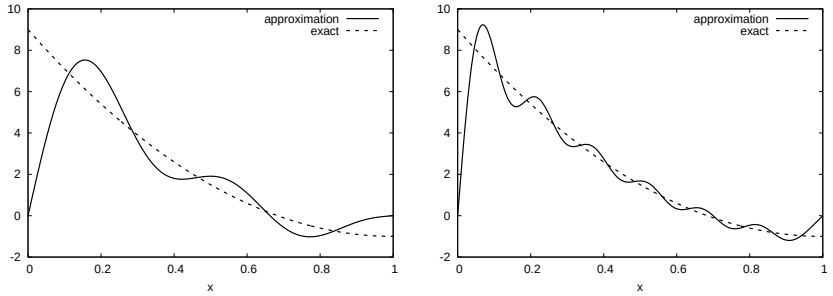

Figure 5: Best approximation of a parabola by a sum of 3 (left) and 11 (right) sine functions.

There is an error $f(0)-u(0)=9$ at $x=0$ in Figure 5 regardless of how large N is, because all $\varphi_i(0)=0$ and hence $u(0)=0$. We may help the approximation to be correct at $x=0$ by seeking

$$u(x) = f(0) + \sum_{j=0}^{N} c_j\varphi_j(x)\,. \tag{39}$$

However, this adjustments introduces a new problem at $x=1$ since we now get an error $f(1)-u(1)=f(1)-0=-1$ at this point. A more clever adjustment is to replace the $f(0)$ term by a term that is $f(0)$ at $x=0$ and $f(1)$ at $x=1$. A simple linear combination $f(0)(1-x)+xf(1)$ does the job:

$$u(x) = f(0)(1-x) + xf(1) + \sum_{j=0}^{N} c_j\varphi_j(x)\,. \tag{40}$$

This adjustment of u alters the linear system slightly as we get an extra term $-(f(0)(1-x)+xf(1), \varphi_i)$ on the right-hand side. Figure 6 shows the result of ensuring right boundary values: even 3 sines can now adjust the $f(0)(1-x)+xf(1)$ term such that u approximates the parabola really well, at least visually.

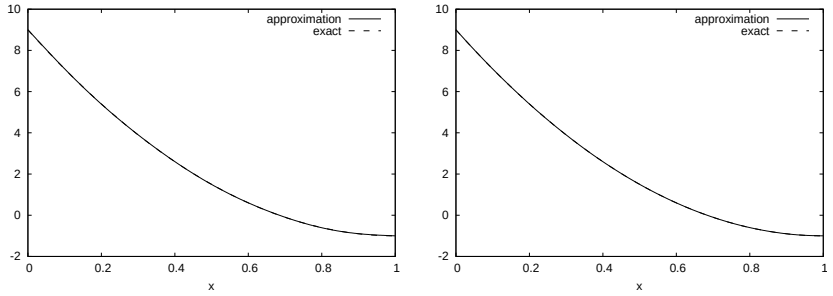

Figure 6: Best approximation of a parabola by a sum of 3 (left) and 11 (right) sine functions with a boundary term.

2.8 Orthogonal basis functions

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 WolframAlpha[2][3][4] (avoid `i` in the integrand as this symbol means the imaginary unit $\sqrt{-1}$). Also by asking WolframAlpha 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:

$$c_i = 2\int_0^1 f(x)\sin((i+1)\pi x)dx, \quad i = 0,\ldots,N, \tag{41}$$

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 a Fourier series expansion, the approximation method derived in Section 2 results in the classical Fourier series for $f(x)$ (see Exercise 6 for details).

For orthogonal basis functions we can make the `least_squares` function (much) more efficient since we know that the matrix is diagonal and only the diagonal elements need to be computed:

```
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`.

2.9 The collocation (interpolation) 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, $i = 0,\ldots,N$:

[2] `http://wolframalpha.com`
[3] `http://wolframalpha.com`
[4] `http://wolframalpha.com`

$$u(x_i) = \sum_{j=0}^{N} c_j \varphi_j(x_i) = f(x_i), \quad i = 0, \ldots, N\,. \tag{42}$$

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

$$\sum_{j=0}^{N} A_{i,j} c_j = b_i, \quad i = 0, \ldots, N, \tag{43}$$

with

$$A_{i,j} = \varphi_j(x_i), \tag{44}$$
$$b_i = f(x_i)\,. \tag{45}$$

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 $c_0, \ldots, c_N$:

```
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 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.

Example. 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 7 (left) 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 7 (right).

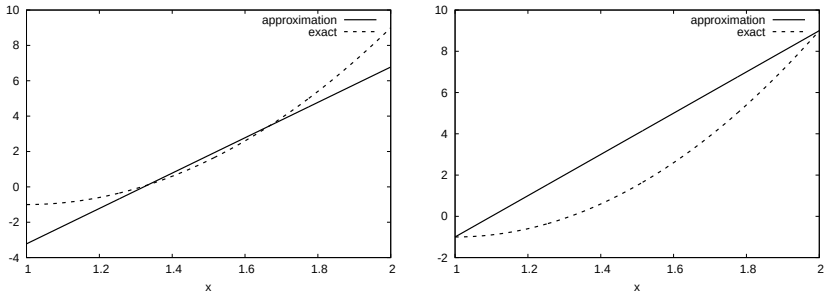

Figure 7: Approximation of a parabola by linear functions computed by two interpolation points: 4/3 and 5/3 (left) versus 1 and 2 (right).

2.10 Lagrange polynomials

In Section 2.7 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 an interpolation/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*. (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 polynomials have the form

$$\varphi_i(x) = \prod_{j=0, j\neq i}^{N} \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_i - x_0} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_N}{x_i - x_N}, \quad (46)$$

for $i = 0, \ldots, N$. We see from (46) that all the φ_i functions are polynomials of degree N which have the property

$$\varphi_i(x_s) = \begin{cases} 1, & i = s, \\ 0, & i \neq s, \end{cases} \tag{47}$$

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

$$c_i = f(x_i), \quad i = 0, \ldots, N, \tag{48}$$

and

$$u(x) = \sum_{j=0}^{N} f(x_i)\varphi_i(x)\,. \tag{49}$$

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 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 (47).

Successful example. 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 8 shows the results. There is little difference between the least squares and the interpolation technique. Increasing N gives visually better approximations.

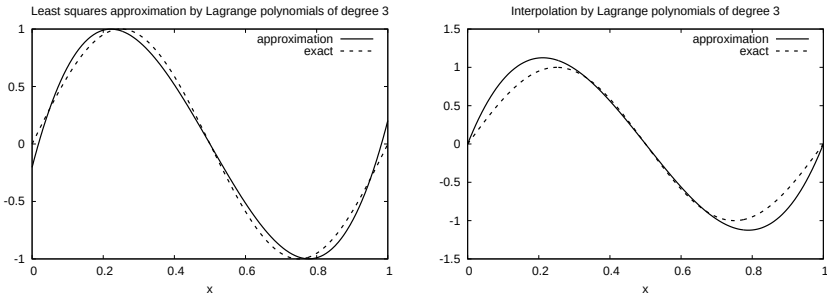

Figure 8: Approximation via least squares (left) and interpolation (right) of a sine function by Lagrange interpolating polynomials of degree 4.

Less successful example. The next example concerns interpolating $f(x) = |1 - 2x|$ on $\Omega = [0, 1]$ using Lagrange polynomials. Figure 9 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 10 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.

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

$$x_i = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos\left(\frac{2i+1}{2(N+1)}pi\right), \quad i = 0\ldots, N, \tag{50}$$

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 11 shows the improvement of using Chebyshev nodes (compared with Figure 9).

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 pursued in the finite element 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 such that it falls back on numerical integration if the symbolic integration is unsuccessful. 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`

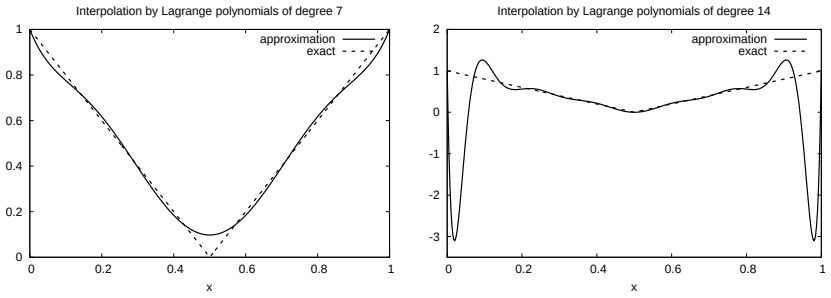

Figure 9: Interpolation of an absolute value function by Lagrange polynomials and uniformly distributed interpolation points: degree 7 (left) and 14 (right).

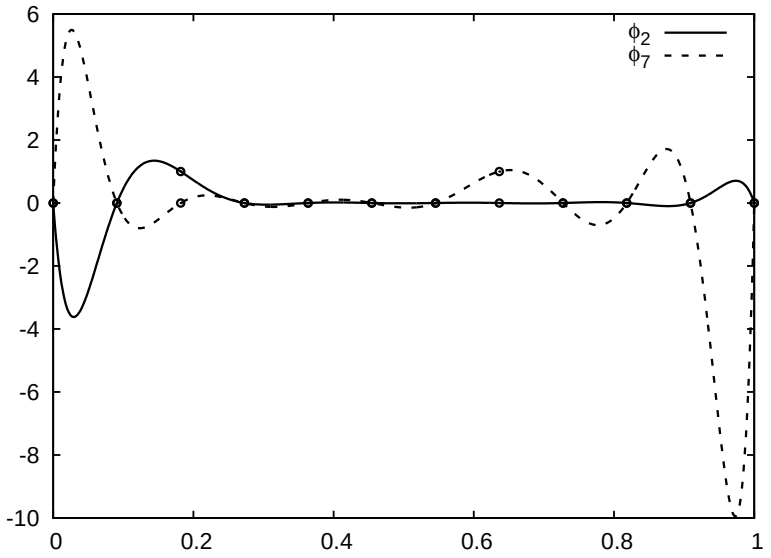

Figure 10: Illustration of the oscillatory behavior of two Lagrange polynomials for 12 uniformly spaced points (marked by circles).

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]))
```

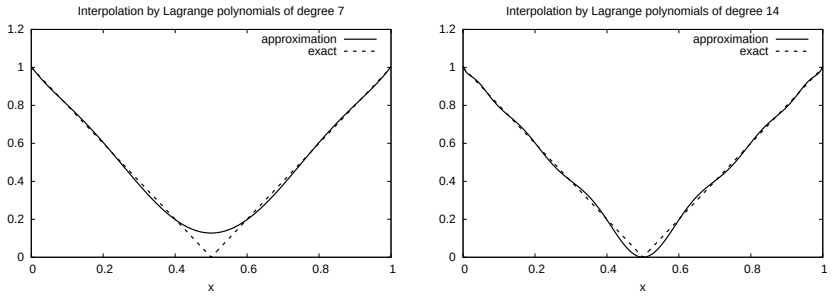

Figure 11: Interpolation of an absolute value function by Lagrange polynomials and Chebyshev nodes as interpolation points: degree 7 (left) and 14 (right).

```
            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
```

3 Finite element basis functions

The specific basis functions exemplified in Section 2 are in general nonzero on the entire domain Ω, see Figurefem:approx:fe:fig:u:sin for an example. 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, see Figure **??** for a sketch involving locally defined hat functions that make $u = \sum_j c_j\varphi_j$ piecewise linear. At the boundaries between subdomains one normally forces continuity of the function only 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*.

We first introduce the concepts of elements and nodes in a simplistic fashion as often met in the literature. Later, we shall generalize the concept of an element, which is a necessary step to treat a wider class of approximations within

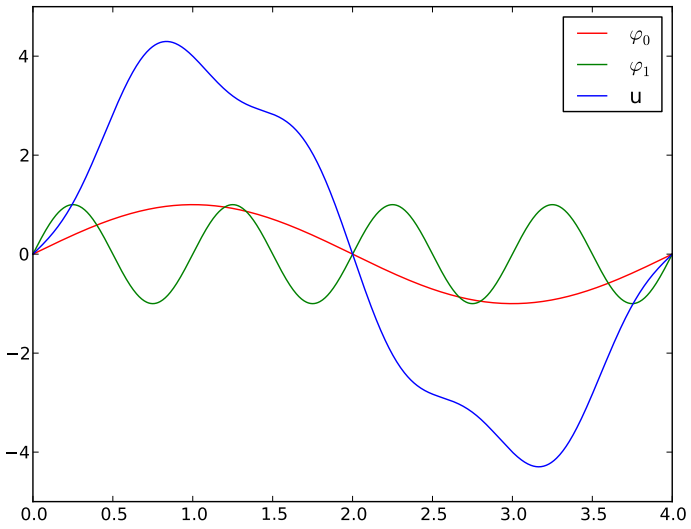

Figure 12: Approximation based on sine basis functions.

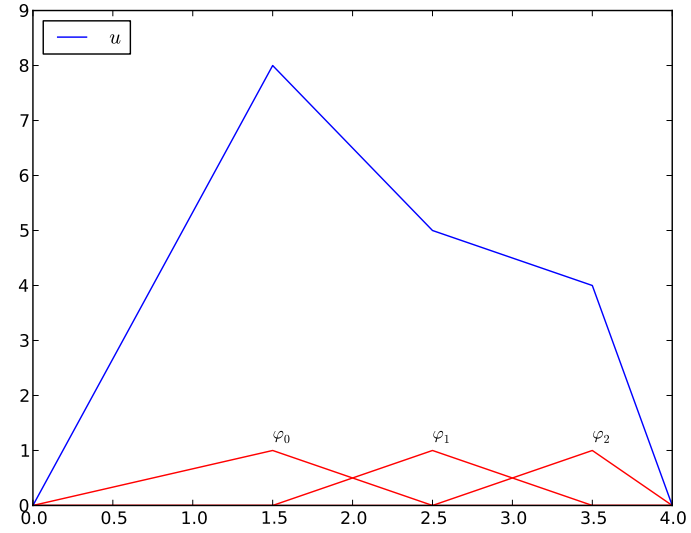

Figure 13: Approximation based on local piecewise linear (hat) functions.

the family of finite element methods. The generalization is also compatible with the concepts used in the FEniCS[56] finite element software.

3.1 Elements and nodes

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

$$\Omega = \Omega^{(0)} \cup \cdots \cup \Omega^{(n_e)} . \tag{51}$$

We shall for now refer to $\Omega^{(e)}$ as an *element*, having number e. 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 within an element and in the global domain. These are referred to as *local* and *global* node numbers, 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.

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 0 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 1, 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]]` (recall that nodes and elements start their numbering at 0).

3.2 The basis functions

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

[5] `http://fenicsproject.org`
[6] `http://fenicsproject.org`

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. On all other elements, $\varphi_i = 0$.
- 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, $\varphi_i = 0$.

A visual impression of three such basis functions are given in Figure 15. Sometimes we refer to a Lagrange polynomial on an element e, which means the basis function $\varphi_i(x)$ when $x \in \Omega^{(e)}$, and $\varphi_i(x) = 0$ when $x \notin \Omega^{(e)}$.

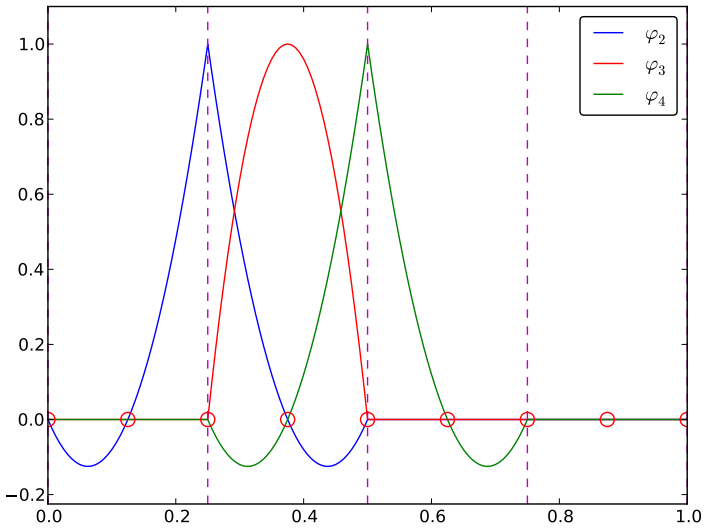

Figure 14: Illustration of the piecewise quadratic basis functions associated with nodes in element 1.

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

$$\varphi_i(x_j) = \begin{cases} 1, & i = j, \\ 0, & i \neq j, \end{cases} \tag{52}$$

when x_j is a node in the mesh with global node number j, because the Lagrange polynomials are constructed to have this property. The property also implies a convenient interpretation of c_i as the value of u at node i:

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

Because of this interpretation, the coefficient c_i is by many named u_i or U_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,
- $\varphi_i(x)\varphi_j(x) \neq 0$ 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 in computations to save memory and CPU time.

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.

Example on quadratic φ_i. Figure 15 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 vertical dashed lines.

Let us explain in detail how the basis functions are constructed according to the principles. Consider element number 1 in Figure 15, $\Omega^{(1)} = [0.25, 0.5]$, 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:

1. The polynomial that is 1 at local node 1 ($x = 0.375$, global node 3) makes up the basis function $\varphi_3(x)$ over this element, with $\varphi_3(x) = 0$ outside the element.
2. The Lagrange polynomial that is 1 at local node 0 is the "right part" of the global basis function $\varphi_2(x)$. The "left part" of $\varphi_2(x)$ consists of a Lagrange polynomial associated with local node 2 in the neighboring element $\Omega^{(0)} = [0, 0.25]$.
3. Finally, the polynomial that is 1 at local node 2 (global node 4) is the "left part" of the global basis function $\varphi_4(x)$. The "right part" comes from the Lagrange polynomial that is 1 at local node 0 in the neighboring element $\Omega^{(2)} = [0.5, 0.75]$.

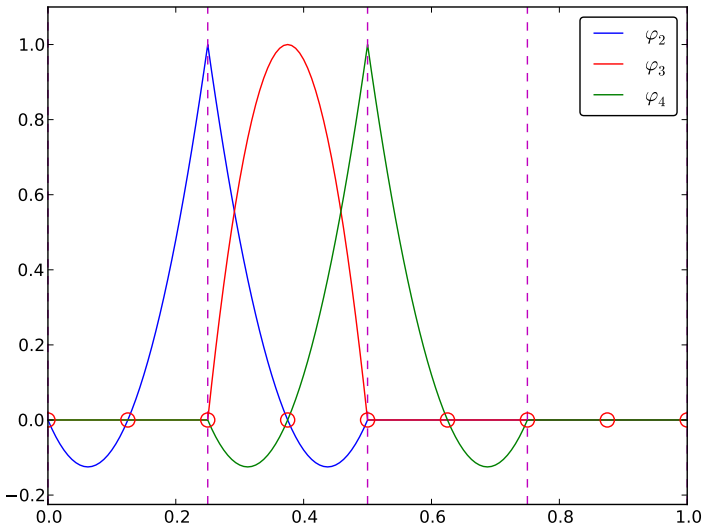

Figure 15: Illustration of the piecewise quadratic basis functions associated with nodes in element 1.

As mentioned earlier, any global basis function $\varphi_i(x)$ is zero on elements that do not share the node with global node number i.

The other global functions associated with internal nodes, φ_1, φ_5, and φ_7, are all of the same shape as the drawn φ_3, while the global basis functions associated with shared nodes also have the same shape, provided the elements are of the same length.

Example on linear φ_i. Figure 16 shows 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 the Lagrange polynomial in element 0 that is 1 at local node 1 and 0 at local node 0, combined with 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

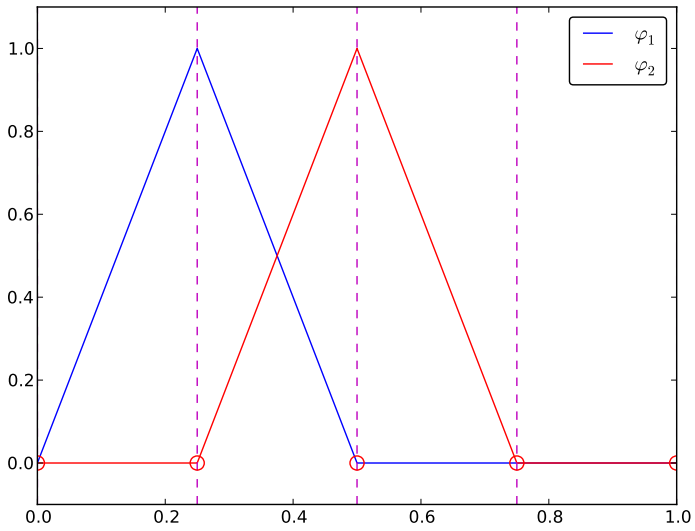

Figure 16: Illustration of the piecewise linear basis functions associated with nodes in element 1.

$$\varphi_i(x) = \begin{cases} 0, & x < x_{i-1}, \\ (x - x_{i-1})/(x_i - x_{i-1}), & x_{i-1} \leq x < x_i, \\ 1 - (x - x_i)/(x_{i+1} - x_i), & x_i \leq x < x_{i+1}, \\ 0, & x \geq x_{i+1}\,. \end{cases} \tag{53}$$

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

$$\varphi_i(x) = \begin{cases} 0, & x < x_{i-1}, \\ (x - x_{i-1})/h, & x_{i-1} \leq x < x_i, \\ 1 - (x - x_i)/h, & x_i \leq x < x_{i+1}, \\ 0, & x \geq x_{i+1} \end{cases} \tag{54}$$

Example on cubic φ_i. Piecewise cubic basis functions can be defined by introducing four nodes per element. Figure 17 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.

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 15 are seen to be of two types. "Rounded hats" associated with internal nodes in the elements and some more

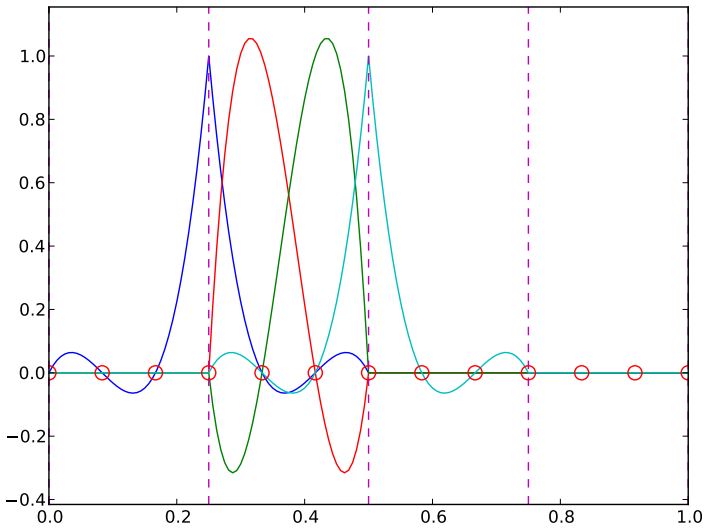

Figure 17: Illustration of the piecewise cubic basis functions associated with nodes in element 1.

"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 17.

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).

3.3 Calculating the linear system

The elements in the coefficient matrix and right-hand side, given by the formulas (27) and (28), will now be calculated for piecewise polynomial basis functions. Consider P1 (piecewise linear) elements. Nodes and elements numbered consecutively from left to right imply the nodes $x_i = ih$ and the elements

$$\Omega^{(i)} = [x_i, x_{i+1}] = [ih, (i+1)h], \quad i = 0, \ldots, N-1\,. \tag{55}$$

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 (54) and a graphical illustration is provided in

Figure 16. First we clearly see from Figure 16 that the important property $\varphi_i(x)\varphi_j(x) \neq 0$ if and only if $j = i-1$, $j = i$, or $j = i+1$, or alternatively expressed, 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

$$b_i = \int_{x_{i-1}}^{x_i} \frac{x - x_{i-1}}{h} f(x) dx + \int_{x_i}^{x_{i+1}} \left(1 - \frac{x - x_i}{h}\right) f(x) dx\,. \tag{56}$$

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 18 (left). Doubling the number of elements to four leads to the improved approximation in the right part of Figure 18.

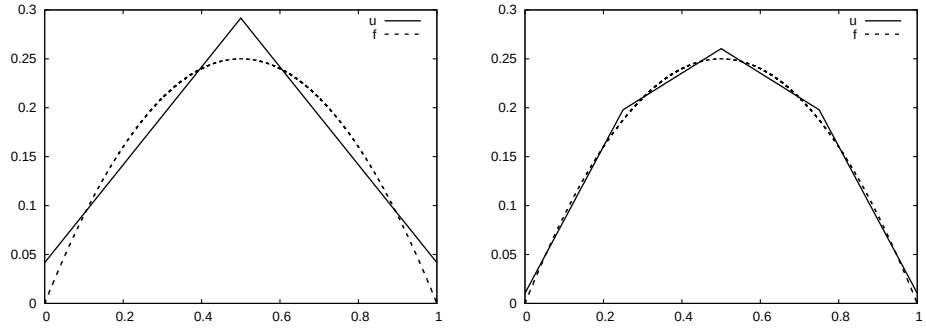

Figure 18: Least squares approximation using 2 (left) and 4 (right) P1 elements.

3.4 Assembly of elementwise computations

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.

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

$$A_{i,j} = \int_\Omega \varphi_i \varphi_j dx = \sum_e A^{(e)}_{i,j}, \quad A^{(e)}_{i,j} = \int_{\Omega^{(e)}} \varphi_i \varphi_j dx\,. \tag{57}$$

Now, $A^{(e)}_{i,j} \neq 0$ 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 number i. This is just a short mathematical 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. For the case $d = 2$ we have

$$\tilde{A}^{(e)} = \begin{bmatrix} \tilde{A}^{(e)}_{0,0} & \tilde{A}^{(e)}_{0,1} & \tilde{A}^{(e)}_{0,2} \\ \tilde{A}^{(e)}_{1,0} & \tilde{A}^{(e)}_{1,1} & \tilde{A}^{(e)}_{1,2} \\ \tilde{A}^{(e)}_{2,0} & \tilde{A}^{(e)}_{2,1} & \tilde{A}^{(e)}_{2,2} \end{bmatrix}.$$

Given the numbers $\tilde{A}^{(e)}_{r,s}$, we should according to (57) add the contributions to the global coefficient matrix by

$$A_{q(e,r),q(e,s)} := A_{q(e,r),q(e,s)} + \tilde{A}^{(e)}_{r,s}, \quad r,s = 0,\ldots,d\,. \tag{58}$$

This process of adding in elementwise contributions to the global matrix is called *finite element assembly* or simply *assembly*. Figure 19 illustrates how element matrices for elements with two nodes are added into the global matrix. More specifically, the figure shows how the element matrix associated with elements 2 and 3 assembled, assuming that global nodes are numbered from left to right in the domain.

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

$$b_i = \int_\Omega \varphi_i \varphi_j dx = \sum_e b^{(e)}_i, \quad b^{(e)}_i = \int_{\Omega^{(e)}} f(x)\varphi_i(x) dx\,. \tag{59}$$

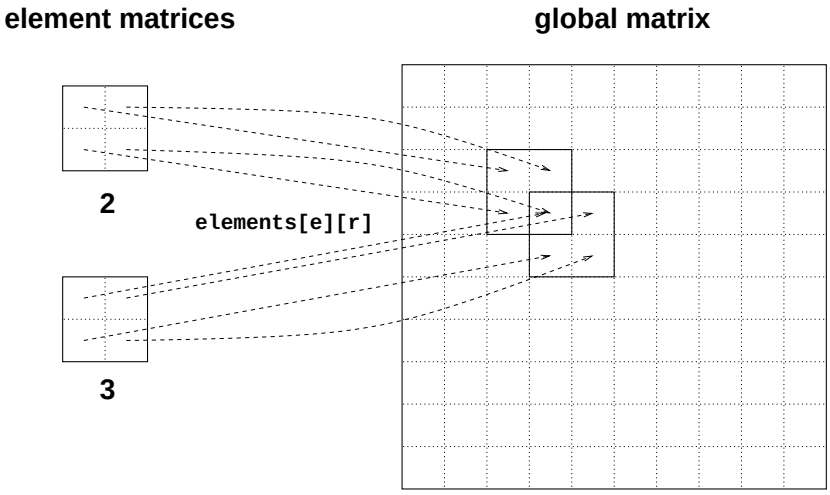

Figure 19: Illustration of matrix assembly.

We observe that $b_i^{(e)} \neq 0$ 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 $b_i^{(e)}$, 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:

$$b_{q(e,r)} := b_{q(e,r)} + \tilde{b}_r^{(e)}, \quad r, s = 0, \ldots, d\,. \tag{60}$$

3.5 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 $\Omega^{(e)} = [x_L, x_R]$, it is convenient to map the element domain $[x_L, x_R]$ to a standardized reference element domain $[-1, 1]$. (We have now introduced x_L and x_R as the left and right boundary points of an arbitrary element. With a natural numbering of nodes and elements from left to right through the domain, $x_L = x_e$ and $x_R = x_{e+1}$.) Let X be the coordinate in the reference element. A linear or *affine mapping* from X to x reads

$$x = \frac{1}{2}(x_L + x_R) + \frac{1}{2}(x_R - x_L)X\,. \tag{61}$$

This relation can alternatively be expressed by

$$x = x_m + \frac{1}{2}hX, \tag{62}$$

where we have introduced the element midpoint $x_m = (x_L + x_R)/2$ and the element length $h = x_R - x_L$.

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

$$\tilde{\varphi}_r(X) = \varphi_{q(e,r)}(x(X)) \tag{63}$$

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

$$\tilde{A}^{(e)}_{r,s} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}(x)\varphi_{q(e,s)}(x)dx = \int_{-1}^{1} \tilde{\varphi}_r(X)\tilde{\varphi}_s(X)\frac{dx}{dX}dX\,. \tag{64}$$

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, $\det J = dx/dX = h/2$. The integration over the reference element is then written as

$$\tilde{A}^{(e)}_{r,s} = \int_{-1}^{1} \tilde{\varphi}_r(X)\tilde{\varphi}_s(X)\det J\, dX\,. \tag{65}$$

The corresponding formula for the element vector entries becomes

$$\tilde{b}^{(e)}_{r} = \int_{\Omega^{(e)}} f(x)\varphi_{q(e,r)}(x)dx = \int_{-1}^{1} f(x(X))\tilde{\varphi}_r(X)\det J\, dX\,. \tag{66}$$

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 $\tilde{\varphi}_r(X)$. This is a very convenient simplification compared to specifying piecewise polynomials in the physical domain.

The $\tilde{\varphi}_r(x)$ 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

$$\tilde{\varphi}_0(X) = \frac{1}{2}(1 - X) \tag{67}$$

$$\tilde{\varphi}_1(X) = \frac{1}{2}(1 + X) \tag{68}$$

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

$$\tilde{\varphi}_0(X) = \frac{1}{2}(X-1)X \tag{69}$$

$$\tilde{\varphi}_1(X) = 1 - X^2 \tag{70}$$

$$\tilde{\varphi}_2(X) = \frac{1}{2}(X+1)X \tag{71}$$

In general,

$$\tilde{\varphi}_r(x) = \prod_{s=0, s\neq r}^{d} \frac{X - X_{(s)}}{X_{(r)} - X_{(s)}}, \tag{72}$$

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

3.6 Integration over a reference element

To illustrate the concepts from the previous section in a specific example, 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 the general expressions (65) and (66) for $\tilde{A}^{(e)}_{r,s}$ and $\tilde{b}^{(e)}_r$. Writing these out for the choices (67) and (68), and using that $\det J = h/2$, we get

$$\begin{aligned}
\tilde{A}^{(e)}_{0,0} &= \int_{-1}^{1} \tilde{\varphi}_0(X)\tilde{\varphi}_0(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} \frac{1}{2}(1-X)\frac{1}{2}(1-X)\frac{h}{2}dX = \frac{h}{8}\int_{-1}^{1}(1-X)^2 dX = \frac{h}{3}, \qquad (73)\\
\tilde{A}^{(e)}_{1,0} &= \int_{-1}^{1} \tilde{\varphi}_1(X)\tilde{\varphi}_0(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} \frac{1}{2}(1+X)\frac{1}{2}(1-X)\frac{h}{2}dX = \frac{h}{8}\int_{-1}^{1}(1-X^2) dX = \frac{h}{6}, \qquad (74)\\
\tilde{A}^{(e)}_{0,1} &= \tilde{A}^{(e)}_{1,0}, \qquad (75)\\
\tilde{A}^{(e)}_{1,1} &= \int_{-1}^{1} \tilde{\varphi}_1(X)\tilde{\varphi}_1(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} \frac{1}{2}(1+X)\frac{1}{2}(1+X)\frac{h}{2}dX = \frac{h}{8}\int_{-1}^{1}(1+X)^2 dX = \frac{h}{3}\,. \qquad (76)
\end{aligned}$$

$$
\begin{aligned}
\tilde{b}_0^{(e)} &= \int_{-1}^{1} f(x(X))\tilde{\varphi}_0(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} (x_m + \frac{1}{2}hX)(1 - (x_m + \frac{1}{2}hX))\frac{1}{2}(1-X)\frac{h}{2}dX \\
&= -\frac{1}{24}h^3 + \frac{1}{6}h^2 x_m - \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m\tilde{b}_1^{(e)} \qquad (77) \\
&= \int_{-1}^{1} f(x(X))\tilde{\varphi}_0(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} (x_m + \frac{1}{2}hX)(1 - (x_m + \frac{1}{2}hX))\frac{1}{2}(1+X)\frac{h}{2}dX \\
&= -\frac{1}{24}h^3 - \frac{1}{6}h^2 x_m + \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m\,. \qquad (78)
\end{aligned}
$$

In the last two expressions we have used the element midpoint x_m.

Integration of lower-degree polynomials above is tedious, and higher-degree polynomials that very much more algebra, but `sympy` may help. For example,

```
>>> import sympy as sm
>>> x, x_m, h, X = sm.symbols('x x_m h X')
>>> sm.integrate(h/8*(1-X)**2, (X, -1, 1))
h/3
>>> sm.integrate(h/8*(1+X)*(1-X), (X, -1, 1))
h/6
>>> x = x_m + h/2*X
>>> b_0 = sm.integrate(h/4*x*(1-x)*(1-X), (X, -1, 1))
>>> print b_0
-h**3/24 + h**2*x_m/6 - h**2/12 - h*x_m**2/2 + h*x_m/2
```

For inclusion of formulas in documents 9like the present one), `sympy` can print expressions in LaTeX format:

```
>>> print sm.latex(b_0, mode='plain')
- \frac{1}{24} h^{3} + \frac{1}{6} h^{2} x_{m}
- \frac{1}{12} h^{2} - \frac{1}{2} h x_{m}^{2}
+ \frac{1}{2} h x_{m}
```

4 Implementation

Based on the experience from the previous example, it makes sense to write some code to automate the integration process for any choice of finite element basis functions. In addition, we can automate the assembly process and linear system solution. Appropriate functions for this purpose document all details of all steps in the finite element computations and can found in the module file `fe_approx1D.py`. Some of the functions are explained below.

4.1 Integration

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)  # node spacing
        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 `Symbol` object from `sympy`. 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 Section 2.7, 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:

```
>>> from fe_approx1D import *
>>> 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)  # substitute mapping formula for 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 the symbol `x` in the expression for `f` by the mapping formula such that `f` contains the variable `X`.

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`. The result of the integral is then an `Integral` object and not a number as when symbolic integration is successful. It may therefore be wise to introduce a fallback on numerical integration. The symbolic integration can also take much time before an unsuccessful conclusion so we may introduce a parameter `symbolic` and set it to `False` to avoid symbolic integration:

```
def element_vector(f, phi, Omega_e, symbolic=True):
        ...
        if symbolic:
            I = sm.integrate(f*phi[r]*detJ, (X, -1, 1))
        if not symbolic or 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 a real number.

4.2 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)
    if symbolic:
        A = sm.zeros((n_n, n_n))
        b = sm.zeros((n_n, 1))    # note: (n_n, 1) matrix
    else:
        A = np.zeros((n_n, n_n))
        b = np.zeros(n_n)
    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 coefficients $c_0, \ldots, c_N$ in the expansion $u(x) = \sum_j c_j\varphi_j$ as the solution vector `c` of the linear system:

```
if symbolic:
    c = A.LUsolve(b)
else:
    c = np.linalg.solve(A, b)
```

When `A` and `b` are `sympy` arrays, solution procedure implied by `A.LUsolve` is symbolic, otherwise, when `A` and `b` are `numpy` arrays, a standard numerical solver is called. The symbolic version is suited for small problems only (small N values) since the calculation time becomes prohibitively large otherwise. Normally, the symbolic integration will be more time consuming in small problems than the symbolic solution of the linear system.

4.3 Example on computing approximations

We can exemplify the use of `assemble` on the computational case from Section 3.3 with two P1 elements (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 for an interpolation/collocation method, taking the nodes as collocation points. This is carried out by evaluating `f` numerically at the nodes:

```
>>> fn = sm.lambdify([x], f)
>>> c = [fn(xc) for xc in nodes]
>>> c
[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.pdf'):
```

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 evaluating $\sum_r c_r\tilde{\varphi}_r(X)$ 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`.

4.4 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 is encouraged to assemble the element matrices by hand and verify this result, as this exercise will give a hands-on understanding of what the assembly is about.) In general we have a coefficient matrix that is tridiagonal:

$$A = \frac{h}{6}\begin{pmatrix} 2 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 1 & 4 & 1 & \ddots & & & & & \vdots \\ 0 & 1 & 4 & 1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & 1 & 4 & 1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & 1 & 4 & 1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 & 2 \end{pmatrix} \tag{79}$$

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 the particular choice of $f(x)$. It is easier to look at the integration in x coordinates, which gives the general formula (56). For equal-sized elements of length h, we can apply the Trapezoidal rule at the global node points to arrive at a somewhat more specific expression than (56):

$$b_i = h\left(\frac{1}{2}\phi_i(x_0)f(x_0) + \frac{1}{2}\phi_i(x_N)f(x_N) + \sum_{j=1}^{N-1}\phi_i(x_i)f(x_i)\right) \tag{80}$$

$$= \begin{cases} \frac{1}{2}hf(x_i), & i = 0 \text{ or } i = N, \\ hf(x_i), & 1 \le i \le N-1 \end{cases} \tag{81}$$

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

$$A^{(e)} = \frac{h}{30}\begin{pmatrix} 4 & 2 & -1 \\ 2 & 16 & 2 \\ -1 & 2 & 4 \end{pmatrix} \tag{82}$$

and the following global assembled matrix from four elements:

$$A = \frac{h}{30}\begin{pmatrix} 4 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 16 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & 8 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 16 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & 8 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 16 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & 8 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 16 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 & 4 \end{pmatrix} \tag{83}$$

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.

4.5 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 20 shows how linear and quadratic elements approximates the polynomial $f(x) = x(1-x)^8$ on $\Omega = [0, 1]$, using equal-sized elements. The results arise from the program

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

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

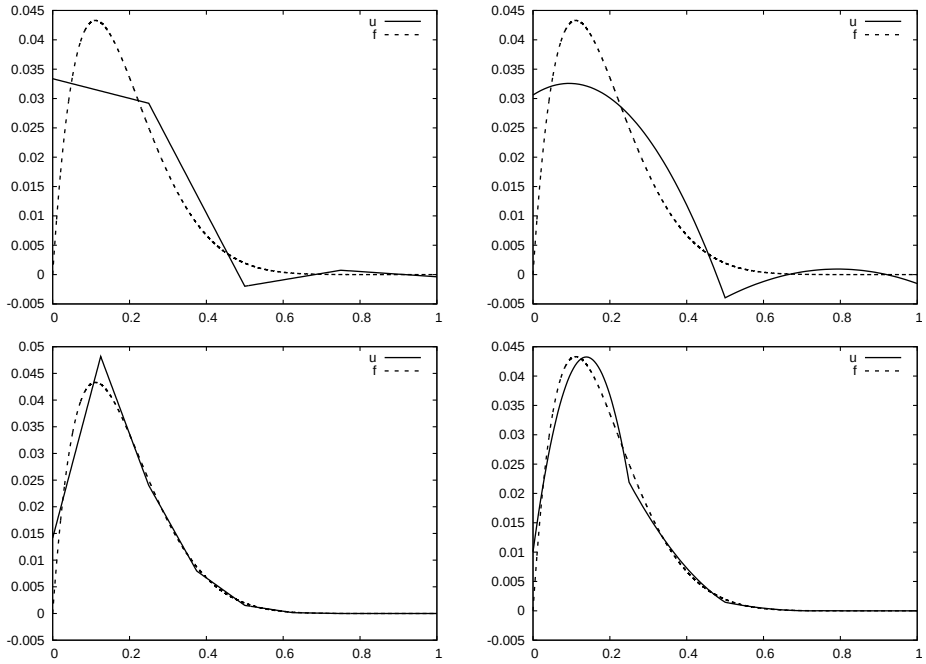

Figure 20: Comparison of the finite element approximations: 4 P1 elements with 5 nodes (upper left), 2 P2 elements with 5 nodes (upper right), 8 P1 elements with 9 nodes (lower left), and 4 P2 elements with 9 nodes (lower right).

4.6 Sparse matrix storage and solution

Some of the examples in the preceding 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 efficient array objects from `numpy`.

A matrix whose majority of entries are zeros, are known as a *sparse* matrix. We know beforehand that matrices from finite element approximations are sparse. The sparsity should be utilized in software as it dramatically decreases the storage demands and the CPU-time needed to compute the solution of the linear system. This optimization is not critical in 1D problems where modern computers can afford computing with all the zeros in the complete square matrix, but in 2D and especially in 3D, sparse matrices are fundamental for feasible finite element computations.

For one-dimensional finite element approximation problems, using a numbering of nodes and elements from left to right over the domain, the assembled coefficient matrix has only a few diagonals different from zero. More precisely, $2d + 1$ diagonals are different from zero. With a different numbering of global nodes, say a random ordering, the diagonal structure is lost, but the number of nonzero elements is unaltered. Figures 21 and 22 exemplifies sparsity patterns.

Figure 21: Matrix sparsity pattern for left-to-right numbering (left) and random numbering (right) of nodes in P1 elements.

The `scipy.sparse` library supports creation of sparse matrices and linear system solution.

- `scipy.sparse.diags` for matrix defined via diagonals
- `scipy.sparse.lil_matrix` for creation via setting elements
- `scipy.sparse.dok_matrix` for creation via setting elements

Examples to come....

Figure 22: Matrix sparsity pattern for left-to-right numbering (left) and random numbering (right) of nodes in P3 elements.

5 Comparison of finite element and finite difference approximation

The previous sections on approximating f by a finite element function u utilize the projection/Galerkin or least squares approaches to minimize the approximation error. We may, alternatively, use the collocation/interpolation method. Here we shall compare these three approaches with what one does in the finite difference method when representing a given function on a mesh.

5.1 Collocation or interpolation

Let x_i, $i = 0, \ldots, N$, be the nodes in the mesh. Collocation means

$$u(x_i) = f(x_i), \quad i = 0, \ldots, N, \tag{84}$$

which translates to

$$\sum_{j=0}^{N} c_j \varphi_j(x_i) = f(x_i),$$

but $\varphi_j(x_i) = 0$ if $i \neq j$ so the sum collapses to one term $c_i \varphi_i(x_i) = c_i$, and we have the result

$$c_i = f(x_i)\,. \tag{85}$$

That is, u *interpolates* f at the node points (the values coincide at these points, but the variation between the points is dictated by the type of polynomials used in the expansion for u). The collocation/interpolation approach is obviously much simpler and faster to use than the least squares or projection/Galerkin approach.

Remark. When dealing with approximation of functions via finite elements, all the three methods are in use, while the least squares and collocation methods are used to only a small extend when solving differential equations.

5.2 Finite difference approximation of given functions

Approximating a given function $f(x)$ on a mesh in a finite difference context will typically just sample f at the grid points. That is, the discrete version of $f(x)$ is the set of point values $f(x_i)$, $i = 0, \ldots, N$, where x_i denotes a mesh point. The collocation/interpolation method above gives exactly the same representation.

How does a finite element Galerkin or least squares approximation differ from this straightforward interpolation of f? This is the question to be addressed next.

5.3 Finite difference interpretation of a finite element approximation

We now limit the scope to P1 elements since this is the element type that gives formulas closest to what one gets from the finite difference method.

The linear system arising from a Galerkin or least squares approximation reads

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

For P1 elements and a uniform mesh of element length h we have calculated the matrix with entries (φ_i, φ_j) in Section 3.3. Equation number i reads

$$\frac{h}{6}(u_{i-1} + 4u_i + u_{i+1}) = (f, \varphi_i)\,. \tag{86}$$

The finite difference counterpart of this equation is just $u_i = f_i$. (The first and last equation, corresponding to $i = 0$ and $i = N$ are slightly different, see Section 4.4.)

The left-hand side of (86) can be manipulated to equal

$$h(u_i - \frac{1}{6}(-u_{i-1} + 2u_i - u_{i+1}))\,. \tag{87}$$

Thinking in terms of finite differences, this is nothing but the standard discretization of

$$h(u - \frac{h^2}{6}u''),$$

or written as

$$[h(u - \frac{h^2}{6}D_xD_xu]_i,$$

with difference operators.

Before interpreting the approximation procedure as solving a differential equation, we need to work out what the right-hand side is in the context of P1 elements. Since φ_i is the linear function that is 1 at x_i and zero at all other

nodes, only the interval $[x_{i-1}, x_{i+1}]$ contributes to the integral on the right-hand side. This integral is naturally split into two parts according to (54):

$$(f, \varphi_i) = \int_{x_{i-1}}^{x_i} f(x) \frac{1}{h}(x - x_{i-1}) dx + \int_{x_i}^{x_{i+1}} f(x) \frac{1}{h}(1 - (x - x_i)) dx \,.$$

However, if f is not known we cannot do much else with this expression. It is clear that many values of f around x_i contributes to the right-hand side, not just the single point value $f(x_i)$ as in the finite difference method.

To proceed with the right-hand side, we turn to numerical integration schemes. Let us say we use the Trapezoidal method for (f, φ_i), based on the node points $x_i = ih$:

$$(f, \varphi_i) = \int_\Omega f \varphi_i dx \approx h \frac{1}{2}(f(x_0)\varphi_i(x_0) + f(x_N)\varphi_i(x_N)) + h \sum_{j=1}^{N-1} f(x_j)\varphi_i(x_j) \,.$$

Since φ_i is zero at all these points, except at x_i, the Trapezoidal rule collapses to one term:

$$(f, \varphi_i) \approx h f(x_i), \tag{88}$$

for $i = 1, \ldots, N-1$, which is the same result as with collocation/interpolation, and of course the same result as in the finite difference method. For $i = 0$ and $i = N$ we get contribution from only one element so

$$(f, \varphi_i) \approx \frac{1}{2} h f(x_i), \quad i = 0, \ i = N \,. \tag{89}$$

Turning to Simpson's rule with sample points also in the middle of the elements, $x_i = ih/2$, $i = 0, \ldots, 2N$, it reads in general

$$\int_\Omega f(x) dx \approx \frac{\tilde{h}}{3} \left(f(x_0) + 2 \sum_{j=2,4,6,\ldots} f(x_j) + 4 \sum_{j=1,3,5,\ldots} f(x_j) + f(x_{2N}) \right),$$

where $\tilde{h} = x_i - x_{i-1} = h/2$ is the spacing between the sample points. We see that the midpoints with odd numbers have the weight $2h/3$ while the node points with even numbers have the weight $h/3$. Since $\varphi_i = 0$ at the even numbers, except for $x_{2i} = x_i$, and $\varphi_i = 0$ at all the midpoints, on the midpoints and $4h/3$ on the node points. Since φ_i vanishes at all the node points, except ξi, and except $x_{2i-1} = \xi i - h/2$ and $x_{2i+1} = \xi i + h/2$, where $\varphi_i = 1/2$, we get

$$(f, \varphi_i) \approx \frac{h}{3}(f(x_i - \frac{1}{2}h) + f(x_i) + f(x_i + \frac{1}{2}h) \,. \tag{90}$$

In a finite difference context we would typically express this formula as

$$\frac{h}{3}(f_{i-\frac{1}{2}} + f_i + f_{i+\frac{1}{2}})\,.$$

This shows that, with Simpson's rule, the finite element method operates with the average of f over three points, while the finite difference method just applies f at one point. We may interpret this as a "smearing" or smoothing of f by the finite element method.

We can now summarize our findings. With the approximation of (f, φ_i) by the Trapezoidal rule, P1 elements give rise to equations that can be expressed as a finite difference discretization of

$$u + \frac{h^2}{6}u'' = f, \quad u'(0) = u'(L) = 0, \tag{91}$$

expressed with operator notation as

$$[u + \frac{h^2}{6}D_xD_xu = f]_i\,. \tag{92}$$

As $h \to 0$, the extra term proportional to u'' goes to zero, and the two methods are then equal.

With the Simpson's rule, we may say that we solve

$$[u + \frac{h^2}{6}D_xD_xu = \bar{f}]_i, \tag{93}$$

where $\bar{f}_i$ means the average $\frac{1}{3}(f_{i-1/2} + f_i + f_{i+1/2})$.

The extra term $\frac{h^2}{6}u''$ represents a smoothing: with just this term, we would find u by integrating f twice and thereby smooth f considerably. In addition, the finite element representation of f involves an average, or a smoothing, of f on the right-hand side of the equation system. If f is a noisy function, direct interpolation $u_i = f_i$ may result in a noisy u too, but with a Galerkin or least squares formulation and P1 elements, we should expect that u is smoother than f unless h is very small.

The interpretation that finite elements tend to smooth the solution is valid in applications far beyond approximation of 1D function.

5.4 Making finite elements behave as finite differences

With a simple trick, using numerical integration, we can easily produce the same result $u_i = f_i$ with the Galerkin or least square formulation with P1 elements. This is useful in many occasions when we deal with more difficult differential equations and want the finite element method to have properties like the finite difference method (solving standard linear wave equations is one primary example).

We have already seen that applying the Trapezoidal rule to the right-hand side (f, φ_i) simply gives f sampled at x_i. Using the Trapezoidal rule on the

matrix entries (φ_i, φ_j) involves a sum

$$\sum_k \varphi_i(x_k)\varphi_j(x_k),$$

but $\varphi_i(x_k) = 0$ for all k, except $k = i$, and $\varphi_j(x_k) = 0$ for all k, except $k = j$. The product $\varphi_i\varphi_j$ is then different from zero only when sampled at x_i and $i = j$. The approximation to the integral is then

$$(\varphi_i, \varphi_j) \approx h, \quad i = j,$$

and zero if $i \neq j$. This means that we have obtained a diagonal matrix! The first and last diagonal elements, (φ_0, φ_0) and (φ_N, φ_N) get contribution only from the first and last element, respectively, resulting in the approximate integral value $h/2$. The corresponding right-hand side also has a factor $1/2$ for $i = 0$ and $i = N$. Therefore, the least squares or Galerkin approach with P1 elements and *Trapezoidal* integration results in

$$c_i = f_i\,.$$

Simpsons's rule can be used to achieve a similar result for P2 elements, i.e, a diagonal coefficient matrix, but with the previously derived average of f on the right-hand side.

Elementwise computations. Identical results to those above will arise if we perform elementwise computations. The idea is to use the Trapezoidal rule on the reference element for computing the element matrix and vector. When assembled, the same equations $c_i = f(x_i)$ arise. Exercise 16 encourages you to carry out the details.

Terminology. The matrix with entries (φ_i, φ_j) typically arises from terms proportional to u in a differential equation where u is the unknown function. This matrix is often called the *mass matrix*, because in the early days of the finite element method, the matrix arose from the mass times acceleration term in Newton's second law of motion. Making the mass matrix diagonal by, e.g., numerical integration, as demonstrated above, is a widely used technique and is called *mass lumping*. In time-dependent problems it can enhance the numerical accuracy and computational efficiency of the finite element method. However, there are also examples where mass lumping destroys accuracy.

6 A generalized element concept

So far, finite element computing has employed the `nodes` and `element` lists together with the definition of the basis functions in the reference element. Suppose we want to introduce a piecewise constant approximation with one basis function $\tilde{\varphi}_0(x) = 1$ in the reference element. Although we could associate

the function value with a node in the middle of the elements, there are no nodes at the ends, and the previous code snippets will not work because we cannot find the element boundaries from the `nodes` list.

6.1 Cells, vertices, and degrees of freedom

We now introduce *cells* as the subdomains $\Omega^{(e)}$ previously referred as elements. The cell boundaries are denoted as *vertices*. The reason for this name is that cells are recognized by their vertices in 2D and 3D. Then we define a set of *degrees of freedom*, which are the quantities we aim to compute. The most common type of degree of freedom is the value of the unknown function u at some point. For example, we can introduce nodes as before and say the degrees of freedom are the values of u at the nodes. The basis functions are constructed so that they equal unity for one particular degree of freedom and zero for the rest. This property ensures that when we evaluate $u = \sum_j c_j\varphi_j$ for degree of freedom number i, we get $u = c_i$. Integrals are performed over cells, usually by mapping the cell of interest to a *reference cell*.

With the concepts of cells, vertices, and degrees of freedom we increase the decoupling the geometry (cell, vertices) from the space of basis functions. We can associate different sets of basis functions with a cell. In 1D, all cells are intervals, while in 2D we can have cells that are triangles with straight sides, or any polygon, or in fact any two-dimensional geometry. Triangles and quadrilaterals are most common, though. The popular cell types in 3D are tetrahedra and hexahedra.

6.2 Extended finite element concept

The concept of a *finite element* is now

- a *reference cell* in a local reference coordinate system;
- a set of *basis functions* $\tilde{\varphi}_i$ defined on the cell;
- a set of *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;
- a mapping between local and global degree of freedom numbers;
- a *mapping* of the reference cell onto to cell in the physical domain.

There must be a geometric description of a cell. This is trivial in 1D since the cell is an interval and is described by the interval limits, here called vertices. If the cell is $\Omega^{(e)} = [x_L, x_R]$, vertex 0 is x_L and vertex 1 is x_R. The reference cell in 1D is $[-1, 1]$ in the reference coordinate system X.

Our previous P1, P2, etc., elements are defined by introducing $d+1$ equally spaced nodes in the reference cell and saying that the degrees of freedom are the $d+1$ function values at these nodes. The basis functions must be 1 at one node

and 0 at the others, and the Lagrange polynomials have exactly this property. The nodes can be numbered from left to right with associated degrees of freedom that are numbered in the same way. The degree of freedom mapping becomes what was previously represented by the `elements` lists. The cell mapping is the same affine mapping (61) as before.

The expansion of u over one cell is often used. In terms of reference coordinates we have

$$u(x) = \sum_r c_r \tilde{\varphi}_r(X), \tag{94}$$

where the sum is taken over the numbers of the degrees of freedom and c_r is the value of u for degree of freedom number r.

6.3 Implementation

Implementationwise,

- we replace `nodes` by `vertices`;
- we introduce `cells` such that `cell[e][r]` gives the mapping from local vertex `r` in cell `e` to the global vertex number in `vertices`;
- we replace `elements` by `dof_map` (the contents are the same).

Consider the example from Section 3.1 where $\Omega = [0, 1]$ is divided into two cells, $\Omega^{(0)} = [0, 0.4]$ and $\Omega^{(1)} = [0.4, 1]$. The vertices are $[0, 0.4, 1]$. Local vertex 0 and 1 are 0 and 0.4 in cell 0 and 0.4 and 1 in cell 1. A P2 element means that the degrees of freedom are the value of u at three equally spaced points (nodes) in each cell. The data structures become

```
vertices = [0, 0.4, 1]
cells = [[0, 1], [1, 2]]
dof_map = [[0, 1, 2], [1, 2, 3]]
```

If we would approximate f by piecewise constants, we simply introduce one point or node in an element, preferably $X = 0$, and choose $\tilde{\varphi}_0(X) = 1$. Only the `dof_map` is altered:

```
dof_map = [[0], [1], [2]]
```

We use the `cells` and `vertices` lists to retrieve information on the geometry of a cell, while `dof_map` is used in the assembly of element matrices and vectors. For example, the `Omega_e` variable (representing the cell interval) in previous code snippets must now be computed as

```
Omega_e = [vertices[cells[e][0], vertices[cells[e][1]]
```

The assembly is done by

```
A[dof_map[e][r], dof_map[e][s]] += A_e[r,s]
b[dof_map[e][r]] += b_e[r]
```

We will hereafter work with `cells`, `vertices`, and `dof_map`.

6.4 Cubic Hermite polynomials

The finite elements considered so far represent u as piecewise polynomials with discontinuous derivatives at the cell boundaries. Sometimes it is desired to have continuous derivatives. A primary examples is the solution of differential equations with fourth-order derivatives where standard finite element formulations lead to a need for basis functions with continuous first-order derivatives. The most common type of such basis functions in 1D is the cubic Hermite polynomials.

There are ready-made formulas for the cubic Hermite polynomials, but it is instructive to apply the principles for constructing basis functions in detail. Given a reference cell $[-1,1]$, we seek cubic polynomials with the values of the function its first-order derivative at $X=-1$ and $X=1$ as the four degrees of freedom. Let us number the degrees of freedom as

- 0: value of function at $X=-1$
- 1: value of first derivative at $X=-1$
- 2: value of function at $X=1$
- 3: value of first derivative at $X=1$

By having the derivatives as unknowns, we ensure that the derivative for the a basis function in two neighboring elements is the same at the node points.

The four basis functions can be written in a general form

$$\tilde{\varphi}_i(X) = \sum_{j=0}^{3} C_{i,j} X^j,$$

with four coefficients $C_{i,j}$, $j=0,1,2,3$, to be determined for each i. The constraints that basis function number i must be 1 for degree of freedom number i and zero for the other three degrees of freedom gives four equations to determine $C_{i,j}$ for each i. In mathematical detail,

$$\begin{aligned}
\tilde{\varphi}_0(-1) &= 1, \quad \tilde{\varphi}_0(1) = \tilde{\varphi}_0'(-1) = \tilde{\varphi}_i'(1) = 0,\\
\tilde{\varphi}_1'(-1) &= 1, \quad \tilde{\varphi}_1(-1) = \tilde{\varphi}_1(1) = \tilde{\varphi}_1'(1) = 0,\\
\tilde{\varphi}_2(1) &= 1, \quad \tilde{\varphi}_2(-1) = \tilde{\varphi}_2'(-1) = \tilde{\varphi}_2'(1) = 0,\\
\tilde{\varphi}_3'(1) &= 1, \quad \tilde{\varphi}_3(-1) = \tilde{\varphi}_3'(-1) = \tilde{\varphi}_3(1) = 0\,.
\end{aligned}$$

The 4 4×4 linear equations can be solved, yielding these formulas for the cubic basis functions:

$$\tilde{\varphi}_0(X) = 1 - \frac{3}{4}(X+1)^2 + \frac{1}{4}(X+1)^3 \tag{95}$$

$$\tilde{\varphi}_1(X) = -(X+1)(1 - \frac{1}{2}(X+1))^2 \tag{96}$$

$$\tilde{\varphi}_2(X) = \frac{3}{4}(X+1)^2 - \frac{1}{2}(X+1)^3 \tag{97}$$

$$\tilde{\varphi}_3(X) = -\frac{1}{2}(X+1)(\frac{1}{2}(X+1)^2 - (X+1)) \tag{98}$$

$$\tag{99}$$

Remaining tasks:

- Global numbering of the dofs
- `dof_map`
- 4x4 element matrix

7 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 in a common form,

$$\int_{-1}^{1} g(X)dX \approx \sum_{j=0}^{M} w_j \bar{X}_j, \tag{100}$$

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.

The very simplest method is the *Midpoint rule*,

$$\int_{-1}^{1} g(X)dX \approx 2g(0), \quad \bar{X}_0 = 0,\ w_0 = 2, \tag{101}$$

which integrates linear functions exactly.

7.1 Newton-Cotes rules

The Newton-Cotes[7] rules are based on a fixed uniform distribution of the points. The first two formulas in this family is the well-known *Trapezoidal rule*,

[7] `http://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas`

$$\int_{-1}^{1} g(X)dX \approx g(-1) + g(1), \quad \bar{X}_0 = -1,\ \bar{X}_1 = 1,\ w_0 = w_1 = 1, \tag{102}$$

and *Simpson's rule*,

$$\int_{-1}^{1} g(X)dX \approx \frac{1}{3}\left(g(-1) + 4g(0) + g(1)\right), \tag{103}$$

where

$$\bar{X}_0 = -1,\ \bar{X}_1 = 0,\ \bar{X}_2 = 1,\ w_0 = w_2 = \frac{1}{3},\ w_1 = \frac{4}{3}. \tag{104}$$

Newton-Cotes rules up to five points is supported in the module file `numint.py`.

For higher accuracy one can divide the reference cell into a set of subintervals and use the rules above on each subinterval. This approach results in *composite* rules, well-known from basic introductions to numerical integration of $\int_a^b f(x)dx$.

7.2 Gauss-Legendre rules with optimized points

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[8] (also known as Gauss-Legendre quadrature or Gaussian quadrature) constitute one such class of integration methods. Two widely applied Gauss-Legendre rules in this family have the choice

$$M = 1: \quad \bar{X}_0 = -\frac{1}{\sqrt{3}},\ \bar{X}_1 = \frac{1}{\sqrt{3}},\ w_0 = w_1 = 1 \tag{105}$$

$$M = 2: \quad \bar{X}_0 = -\sqrt{\frac{3}{5}},\ \bar{X}_0 = 0,\ \bar{X}_2 = \sqrt{\frac{3}{5}},\ w_0 = w_2 = \frac{5}{9},\ w_1 = \frac{8}{9}. \tag{106}$$

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. The code `numint.py` contains a large collection of Gauss-Legendre rules.

8 Approximation of functions in 2D

All the concepts and algorithms developed for approximation of 1D functions $f(x)$ can readily be extended to 2D functions $f(x, y)$ and 3D functions $f(x, y, z)$. Basically, the extensions consists of defining basis functions $\varphi_i(x, y)$ or $\varphi_i(x, y, z)$ over some domain Ω, and for the least squares and Galerkin methods, the integration is done over Ω.

[8] `http://en.wikipedia.org/wiki/Gaussian_quadrature`

8.1 Global basis functions

An example will demonstrate the necessary extensions to use global basis functions and the least squares, Galerkin/projection, or interpolation/collocation methods in 2D. The former two lead to linear systems

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

where the inner product of two functions $f(x, y)$ and $g(x, y)$ is defined completely analogously to the 1D case (24):

$$(f, g) = \int_\Omega f(x, y)g(x, y)dxdy \tag{107}$$

Constructing 2D basis functions from 1D functions. One straightforward way to construct a basis in 2D is to combine 1D basis functions. Say we have the 1D basis

$$\{\hat{\varphi}_0(x), \ldots, \hat{\varphi}_{N_x}(x)\}.$$

We can now form 2D basis functions as products of 1D basis functions: $\hat{\varphi}_p(x)\hat{\varphi}_q(y)$ for $p = 0, \ldots, N_x$ and $q = 0, \ldots, N_y$. We can either work with double indices, $\varphi_{p,q}(x, y) = \hat{\varphi}_p(x)\hat{\varphi}_q(y)$, and write

$$u = \sum_{p=0}^{N_y}\sum_{q=0}^{N_x} c_{p,q}\varphi_{p,q}(x, y),$$

or we may transform the double index (p, q) to a single index i, using $i = pN_y+q$ or $i = qN_x + p$.

Suppose we choose $\hat{\varphi}_p(x) = x^p$, and try an approximation with $N_x = N_y = 1$:

$$\varphi_{0,0} = 1, \quad \varphi_{1,0} = x, \quad \varphi_{0,1} = y, \quad \varphi_{1,1} = xy.$$

Using a mapping to one index like $i = qN_x + p$, we get

$$\varphi_0 = 1, \quad \varphi_1 = x, \quad \varphi_2 = y, \quad \varphi_3 = xy.$$

Hand calculations. With the specific choice $f(x, y) = (1 + x^2)(1 + 2y^2)$ on $\Omega = [0, L_x] \times [0, L_y]$, we can perform actual calculations:

$$A_{0,0} = (\varphi_0, \varphi_0) = \int_0^{L_y}\int_0^{L_x} \varphi_0(x,y)^2 dxdy = \int_0^{L_y}\int_0^{L_x} dxdy = L_x L_y,$$
$$A_{1,0} = (\varphi_1, \varphi_0) = \int_0^{L_y}\int_0^{L_x} xdxdy = \frac{1}{2}L_x^2 L_y,$$
$$A_{0,1} = (\varphi_0, \varphi_1) = \int_0^{L_y}\int_0^{L_x} ydxdy = \frac{1}{2}L_y^2 L_x,$$
$$A_{0,1} = (\varphi_0, \varphi_1) = \int_0^{L_y}\int_0^{L_x} xydxdy = \int_0^{L_y} ydy \int_0^{L_x} xdx \frac{1}{4}L_y^2 L_x^2 .$$

The right-hand side vector has the entries

$$b_0 = (\varphi_0, f) = \int_0^{L_y}\int_0^{L_x} 1\cdot(1+x^2)(1+2y^2)dxdy = \int_0^{L_y}(1+2y^2)dy \int_0^{L_x}(1+x^2)dx$$
$$= (L_y + \frac{2}{3}L_y^3)(L_x + \frac{1}{3}L_x^3)$$
$$b_1 = (\varphi_1, f) = \int_0^{L_y}\int_0^{L_x} x(1+x^2)(1+2y^2)dxdy = \int_0^{L_y}(1+2y^2)dy \int_0^{L_x} x(1+x^2)dx$$
$$= (L_y + \frac{2}{3}L_y^3)(\frac{1}{2}L_x^2 + \frac{1}{4}L_x^4)$$
$$b_2 = (\varphi_2, f) = \int_0^{L_y}\int_0^{L_x} y(1+x^2)(1+2y^2)dxdy = \int_0^{L_y} y(1+2y^2)dy \int_0^{L_x}(1+x^2)dx$$
$$= (\frac{1}{2}L_y + \frac{1}{2}L_y^4)(L_x + \frac{1}{3}L_x^3)$$
$$b_3 = (\varphi_2, f) = \int_0^{L_y}\int_0^{L_x} xy(1+x^2)(1+2y^2)dxdy = \int_0^{L_y} y(1+2y^2)dy \int_0^{L_x} x(1+x^2)dx$$
$$= (\frac{1}{2}L_y^2 + \frac{1}{2}L_y^4)(\frac{1}{2}L_x^2 + \frac{1}{4}L_x^4) .$$

There is a general pattern in these calculations that we can explore. An arbitrary matrix entry has the formula

$$A_{i,j} = (\varphi_i, \varphi_j) = \int_0^{L_y}\int_0^{L_x} \varphi_i\varphi_j dxdy$$
$$= \int_0^{L_y}\int_0^{L_x} \varphi_{p,q}\varphi_{r,s} dxdy = \int_0^{L_y}\int_0^{L_x} \hat{\varphi}_p(x)\hat{\varphi}_q(y)\hat{\varphi}_r(x)\hat{\varphi}_s(y)dxdy$$
$$= \int_0^{L_y} \hat{\varphi}_q(y)\hat{\varphi}_s(y)dy \int_0^{L_x} \hat{\varphi}_p(x)\hat{\varphi}_r(x)dx$$
$$= \hat{A}^{(x)}_{p,r}\hat{A}^{(y)}_{q,s},$$

where

$$\hat{A}^{(x)}_{p,r} = \int_0^{L_x} \hat{\varphi}_p(x)\hat{\varphi}_r(x)dx, \quad \hat{A}^{(y)}_{q,s} = \int_0^{L_y} \hat{\varphi}_q(y)\hat{\varphi}_s(y)dy,$$

are matrix entries for one-dimensional approximations. Moreover, $i = qN_y + q$ and $j = sN_y + r$.

With $\hat{\varphi}_p(x) = x^p$ we have

$$\hat{A}^{(x)}_{p,r} = \frac{1}{p+r+1}L_x^{p+r+1}, \quad \hat{A}^{(y)}_{q,s} = \frac{1}{q+s+1}L_y^{q+s+1},$$

and

$$A_{i,j} = \hat{A}^{(x)}_{p,r}\hat{A}^{(y)}_{q,s} = \frac{1}{p+r+1}L_x^{p+r+1}\frac{1}{q+s+1}L_y^{q+s+1},$$

for $p, r = 0, \ldots, N_x$ and $q, s = 0, \ldots, N_y$.

Corresponding reasoning for the right-hand side leads to

$$\begin{aligned}
b_i = (\varphi_i, f) &= \int_0^{L_y}\int_0^{L_x} \varphi_i f \, dxdx \\
&= \int_0^{L_y}\int_0^{L_x} \hat{\varphi}_p(x)\hat{\varphi}_q(y) f \, dxdx \\
&= \int_0^{L_y} \hat{\varphi}_q(y)(1+2y^2)dy \int_0^{L_y} \hat{\varphi}_p(x)x^p(1+x^2)dx \\
&= \int_0^{L_y} y^q(1+2y^2)dy \int_0^{L_y} x^p(1+x^2)dx \\
&= (\frac{1}{q+1}L_y^{q+1} + \frac{2}{q+3}x^{q+3})(\frac{1}{p+1}L_y^{p+1} + \frac{2}{p+3}x^{p+3})
\end{aligned}$$

Choosing $L_x = L_y = 2$, we have

$$A = \begin{bmatrix} 4 & 4 & 4 & 4 \\ 4 & \frac{16}{3} & 4 & \frac{16}{3} \\ 4 & 4 & \frac{16}{3} & \frac{16}{3} \\ 4 & \frac{16}{3} & \frac{16}{3} & \frac{64}{9} \end{bmatrix}, \quad b = \begin{bmatrix} \frac{308}{9} \\ \frac{140}{3} \\ 44 \\ 60 \end{bmatrix}, \quad c = \begin{bmatrix} -\frac{1}{9}, \\ \frac{4}{3}, \\ -\frac{2}{3}, \\ 8 \end{bmatrix}.$$

Figure 23 illustrates the result.

8.2 Implementation

The `least_squares` function from Section 2.8 and/or the file `approx1D.py` can with very small modifications solve 2D approximation problems. First, let `Omega` now be a list of the intervals in x and y direction. For example, $\Omega = [0, L_x] \times [0, L_y]$ can be represented by `Omega = [[0, L_x], [0, L_y]]`.

Second, the symbolic integration must be extended to 2D:

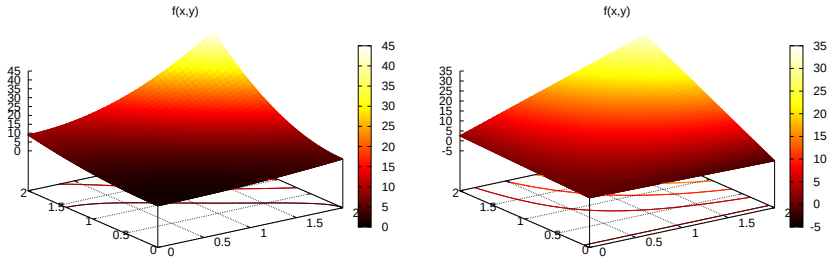

Figure 23: Approximation of a 2D quadratic function (left) by a 2D bilinear function (right) using the Galerkin or least squares method.

```
import sympy as sm

integrand = phi[i]*phi[j]
I = sm.integrate(integrand,
                 (x, Omega[0][0], Omega[0][1]),
                 (y, Omega[1][0], Omega[1][1]))
```

provided `integrand` is an expression involving the `sympy` symbols `x` and `y`. The 2D version of numerical integration becomes

```
if isinstance(I, sm.Integral):
    integrand = sm.lambdify([x,y], integrand)
    I = sm.mpmath.quad(integrand,
                       [Omega[0][0], Omega[0][1]],
                       [Omega[1][0], Omega[1][1]])
```

The right-hand side integrals are modified in a similar way.

Third, we must construct a list of 2D basis functions, e.g.,

```
def taylor(x, y, Nx, Ny):
    return [x**i*y**j for i in range(Nx+1) for j in range(Ny+1)]

def sines(x, y, Nx, Ny):
    return [sm.sin(sm.pi*(i+1)*x)*sm.sin(sm.pi*(j+1)*y)
            for i in range(Nx+1) for j in range(Ny+1)]
```

The complete code appears in `approx2D.py`.

The previous hand calculation where a quadratic f was approximated by a bilinear function can be computed symbolically by

```
>>> f = (1+x**2)*(1+2*y**2)
>>> phi = taylor(x, y, 1, 1)
>>> Omega = [[0, 2], [0, 2]]
>>> u = least_squares(f, phi, Omega)
>>> print u
8*x*y - 2*x/3 + 4*y/3 - 1/9
>>> print sm.expand(f)
2*x**2*y**2 + x**2 + 2*y**2 + 1
```

We may continue with adding higher powers to the basis and check that with $N_x \geq 2$ and $N_y \geq 2$ we recover the exact function f:

```
>>> phi = taylor(x, y, 2, 2)
>>> u = least_squares(f, phi, Omega)
>>> print u
2*x**2*y**2 + x**2 + 2*y**2 + 1
>>> print u-f
0
```

9 Finite elements in 2D and 3D

Finite element approximation is particularly powerful in 2D and 3D because the method can handle a geometrically complex domain Ω with ease. The principal idea is, as in 1D, to divide the domain into cells use polynomials for approximating a function over a cell. Two popular cell shapes are triangles and the quadrilaterals. Figures 24, 25, and 26 provide examples. P1 elements means linear functions $(a_0 + a_1 x + a_2 y)$ over triangles, while Q1 elements have bilinear functions $(a_0 + a_1 x + a_2 y + a_3 xy)$ over rectangular cells. Higher-order elements can easily be defined.

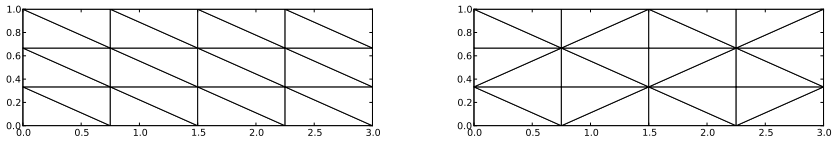

Figure 24: Examples on 2D P1 elements.

9.1 Basis functions over triangles in the physical domain

Cells with triangular shape will be in main focus here. With the P1 triangular element, u is a linear function over each cell, with discontinuous derivatives at the cell boundaries, as depicted in Figure 27.

We give the vertices of the cells global and local numbers as in 1D. The degrees of freedom in the P1 element are the function values at a set of nodes, which are the three vertices. The basis function $\varphi_i(x, y)$ is then 1 at the vertex with global vertex number i and zero at all other vertices. On an element, the three degrees of freedom uniquely determine the linear basis functions in that element, as usual. The global $\varphi_i(x, y)$ function is then a combination of the linear functions (planar surfaces) over all the neighboring cells that have vertex number i in common. Figure 28 tries to illustrate the shape of such a "pyramid"-like function.

Element matrices and vectors. As in 1D, we split the integral over Ω into a sum of integrals over cells. Also as in 1D, φ_i overlaps φ_j (i.e., $\varphi_i \varphi_j \neq 0$) if

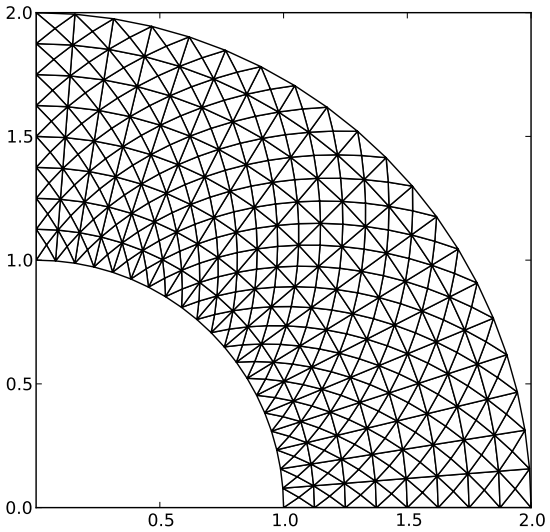

Figure 25: Examples on 2D P1 elements in a deformed geometry.

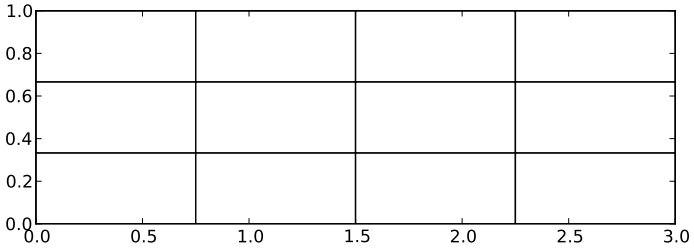

Figure 26: Examples on 2D Q1 elements.

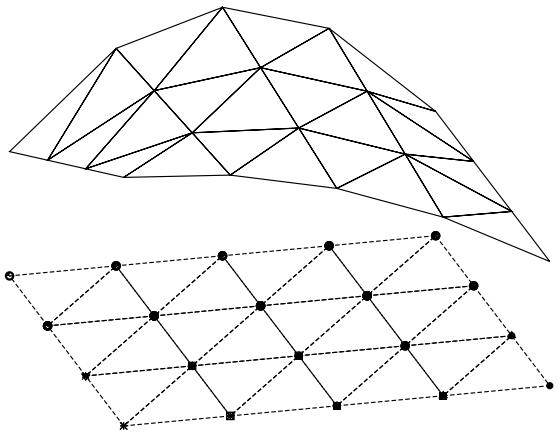

Figure 27: Example on piecewise linear 2D functions defined on triangles.

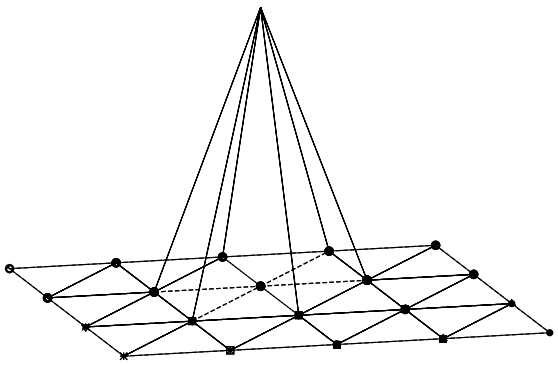

Figure 28: Example on a piecewise linear 2D basis function over a patch of triangles.

and only if i and j are vertices in the same cell. Therefore, the integral of $\varphi_i \varphi_j$ over an element is nonzero only when i and j run over the vertex numbers in the

element. These nonzero contributions to the coefficient matrix are, as in 1D, collected in an element matrix. The size of the element matrix becomes 3×3 since there are three degrees of freedom that i and j run over. Again, as in 1D, we number the local vertices in a cell, starting at 0, and add the entries in the element matrix into the global system matrix, exactly as in 1D. All details and code appear below.

9.2 Basis functions over triangles in the reference cell

As in 1D, we can define the basis functions and the degrees of freedom in a reference cell and then use a mapping from the reference coordinate system to the physical coordinate system. We also have a mapping of local degrees of freedom numbers to global degrees of freedom numbers.

The reference cell in an (X, Y) coordinate system has vertices $(0, 0)$, $(1, 0)$, and $(0, 1)$, corresponding to local vertex numbers 0, 1, and 2, respectively. The P1 element has linear functions $\tilde{\varphi}_r(X, Y)$ as basis functions, $r = 0, 1, 2$. Since a linear function $\tilde{\varphi}_r(X, Y)$ in 2D is on the form $C_{r,0} + C_{r,1}X + C_{r,2}Y$, and hence has three parameters $C_{r,0}$, $C_{r,1}$, and $C_{r,2}$, we need three degrees of freedom. These are in general taken as the function values at a set of nodes. For the P1 element the set of nodes is the three vertices. Figure 29 displays the geometry of the element and the location of the nodes.

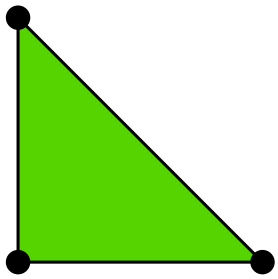

Figure 29: 2D P1 element.

Requiring $\tilde{\varphi}_r = 1$ at node number r and $\tilde{\varphi}_r = 0$ at the two other nodes, gives three linear equations to determine $C_{r,0}$, $C_{r,1}$, and $C_{r,2}$. The result is

$$\tilde{\varphi}_0(X, Y) = 1 - X - Y, \tag{108}$$

$$\tilde{\varphi}_1(X, Y) = X, \tag{109}$$

$$\tilde{\varphi}_2(X, Y) = Y \tag{110}$$

Higher-order approximations are obtained by increasing the polynomial order, adding additional nodes, and letting the degrees of freedom be function values at the nodes. Figure 30 shows the location of the six nodes in the P2 element.

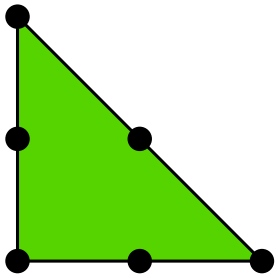

Figure 30: 2D P2 element.

A polynomial of degree p in X and Y has $n_p = (p+1)(p+2)/2$ terms and hence needs n_p nodes. The values at the nodes constitute n_p degrees of freedom. The location of the nodes for $\tilde{\varphi}_r$ up to degree 6 is displayed in Figure 31.

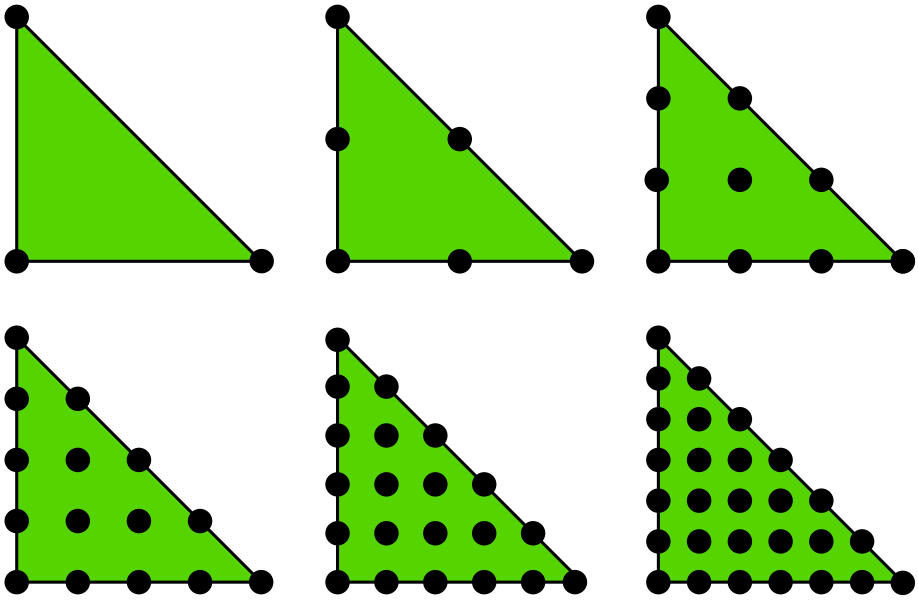

Figure 31: 2D P1, P2, P3, P4, P5, and P6 elements.

The generalization to 3D is straightforward: the reference element is a tetrahedron[9] with vertices $(0,0,0)$, $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$ in a X,Y,Z reference coordinate system. The P1 element has its degrees of freedom as four nodes, which are the four vertices, see Figure 32. The P2 element adds additional nodes along the edges of the cell, yielding a total of 10 nodes and degrees of freedom, see Figure 33.

[9] `http://en.wikipedia.org/wiki/Tetrahedron`

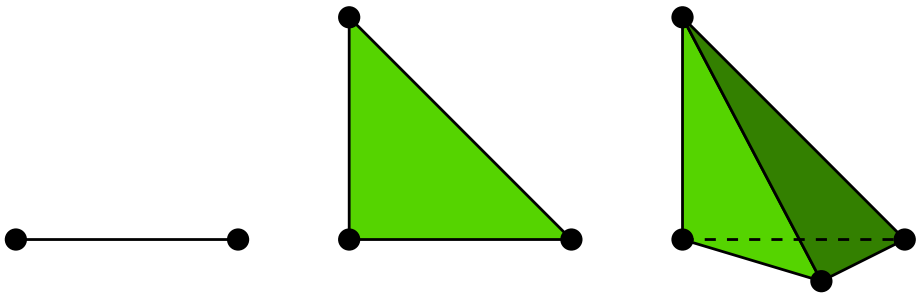

Figure 32: P1 elements in 1D, 2D, and 3D.

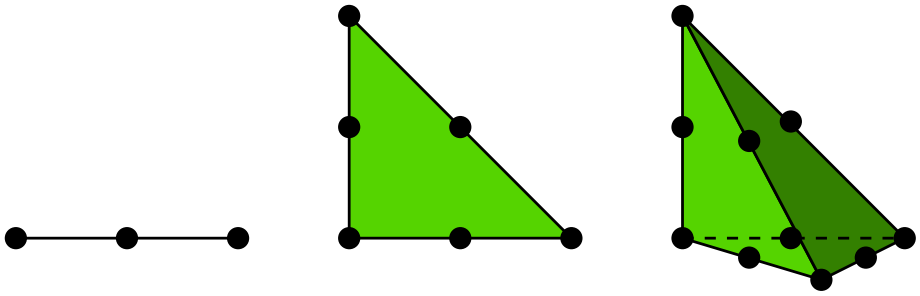

Figure 33: P2 elements in 1D, 2D, and 3D.

The interval in 1D, the triangle in 2D, the tetrahedron in 3D, and its generalizations to higher space dimensions are known as *simplex* cells (the geometry) or *simplex* elements (the geometry, basis functions, degrees of freedom, etc.). The plural forms simplices[10] and simplexes are also a much used shorter terms when referring to this type of cells or elements. The side of a simplex is called a *face*, while the tetrahedron also has *edges*.

Acknowledgment. Figures 29 to 33 are created by Anders Logg and taken from the FEniCS book: *Automated Solution of Differential Equations by the Finite Element Method*[11], edited by A. Logg, K.-A. Mardal, and G. N. Wells, published by Springer[12], 2012.

9.3 Affine mapping of the reference cell

Let $\tilde{\varphi}_r^{(1)}$ denote the basis functions associated with the P1 element in 1D, 2D, or 3D, and let $\boldsymbol{x}_{q(e,r)}$ be the physical coordinates of local vertex number r in cell e. Furthermore, let $\boldsymbol{X}$ be a point in the reference coordinate system corresponding

[10] `http://en.wikipedia.org/wiki/Simplex`
[11] `https://launchpad.net/fenics-book`
[12] `http://www.springer.com/mathematics/computational+science+%26+engineering/book/978-3-642-23098-1`

to the point $\boldsymbol{x}$ in the physical coordinate system. The affine mapping of any $\boldsymbol{X}$ onto $\boldsymbol{x}$ is then defined by

$$\boldsymbol{x} = \sum_r \tilde{\varphi}_r^{(1)}(\boldsymbol{X}) \boldsymbol{x}_{q(e,r)}, \tag{111}$$

where r runs over the local vertex numbers in the cell. The affine mapping maps the straight or planar faces of the reference cell onto straight or planar faces in the physical coordinate system. The mapping can be used for both P1 and higher-order elements.

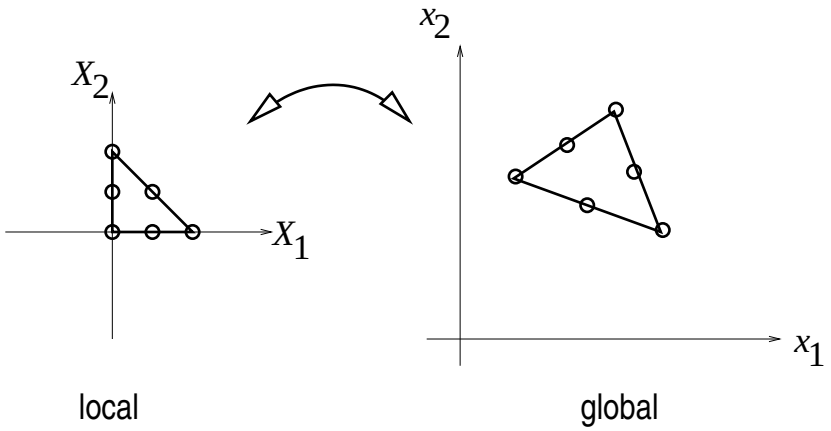

Figure 34: Affine mapping of a P1 element.

9.4 Isoparametric mapping of the reference cell

Instead of using the P1 basis functions in the mapping (111), we may use the basis functions of the actual element:

$$\boldsymbol{x} = \sum_r \tilde{\varphi}_r(\boldsymbol{X}) \boldsymbol{x}_{q(e,r)}, \tag{112}$$

where r runs over all nodes, i.e., all points associated with the degrees of freedom. This is called an *isoparametric mapping*. For P1 elements it is identical to the affine mapping (111), but for higher-order elements the mapping of the straight or planar faces of the reference cell will result in a *curved* face in the physical coordinate system. For example, when we use the basis functions of the triangular P2 element in 2D in (112), the straight faces of the reference triangle are mapped onto curved faces of parabolic shape in the physical coordinate system, see Figure 35.

From (111) or (112) it is easy to realize that the vertices are correctly mapped. Consider a vertex with local number also a much used term when referring to this type of cells or elements$s$. Then $\tilde{\varphi}_s = 1$ at this vertex and

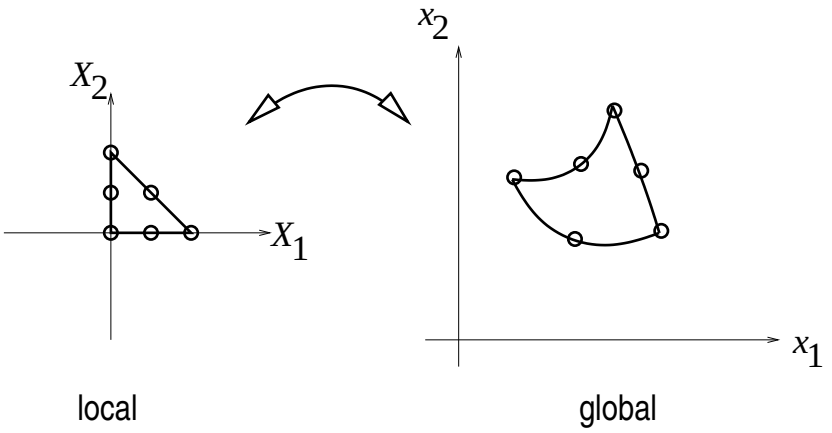

Figure 35: Isoparametric mapping of a P2 element.

zero at the others. This means that only one term in the sum is nonzero and $\boldsymbol{x} = \boldsymbol{x}_{q(e,s)}$, which is the coordinate of this vertex in the global coordinate system.

9.5 Computing integrals

Let $\tilde{\Omega}^r$ denote the reference cell and $\Omega^{(e)}$ the cell in the physical coordinate system. The transformation of the integral from the physical to the reference coordinate system reads

$$\int_{\Omega^{(e)}} \varphi_i(\boldsymbol{x})\varphi_j(\boldsymbol{x})d\boldsymbol{x} = \int_{\tilde{\Omega}^r} \tilde{\varphi}_i(\boldsymbol{X})\tilde{\varphi}_j(\boldsymbol{X}) \det J \, d\boldsymbol{X}, \tag{113}$$

$$\int_{\Omega^{(e)}} \varphi_i(\boldsymbol{x})f(\boldsymbol{x})d\boldsymbol{x} = \int_{\tilde{\Omega}^r} \tilde{\varphi}_i(\boldsymbol{X})f(\boldsymbol{x}(\boldsymbol{X})) \det J \, d\boldsymbol{X}, \tag{114}$$

where $d\boldsymbol{x} = dxdy$ in 2D and $d\boldsymbol{x} = dxdydz$ in 3D, with a similar definition of $d\boldsymbol{X}$. The quantity $\det J$ is the determinant of the Jacobian of the mapping $\boldsymbol{x}(\boldsymbol{X})$. In 2D,

$$J = \begin{bmatrix} \frac{\partial x}{\partial X} & \frac{\partial x}{\partial Y} \\ \frac{\partial y}{\partial X} & \frac{\partial y}{\partial Y} \end{bmatrix}, \quad \det J = \frac{\partial x}{\partial X}\frac{\partial y}{\partial Y} - \frac{\partial x}{\partial Y}\frac{\partial y}{\partial X}\,. \tag{115}$$

With the affine mapping (111), $\det J = 2\Delta$, where Δ is the area or volume of the cell in the physical coordinate system.

Remark. Observe that finite elements in 2D and 3D builds on the same *ideas* and *concepts* as in 1D, but there is simply more to compute because the specific mathematical formulas in 2D and 3D are more complicated.

10 Exercises

Exercise 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, $a, b \in \mathbb{R}$.

On the contrary, show that all quadratic functions of the form $1 + ax^2 + bx$ *do not* constitute a vector space. Filename: `linalg1`.

Exercise 2: Linear algebra refresher II

As an extension of Exercise 1, check out the topic of *inner product spaces*. Suggest a possible inner product for the space of all linear functions of the form $ax + b$, $a, b \in \mathbb{R}$. Show that this inner product satisfies the general requirements of an inner product in a vector space. Filename: `linalg2`.

Exercise 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 $(0, 1)$. Repeat the calculations using the vectors $(2, 1)$ and $(1, 2)$. Filename: `vec111_approx`.

Exercise 4: Approximate the sine function by power functions

Let V be a function space with basis functions x^{2i+1}, $i = 0, 1, \ldots, N$. Find the best approximation to $f(x) = \sin(x)$ among all functions in V, using $N = 8$ for a domain that includes more and more half-periods of the sine function: $\Omega = [0, k\pi/2]$, $k = 2, 3, \ldots, 12$. How does a Taylor series of $\sin(x)$ around x up to degree 9 behave for the largest domain?

Hint. One can make a loop over k and call the `least_squares` and `comparison_plot` from the `approx1D` module. Filename: `sin_powers.py`.

Exercise 5: Approximate a steep function by sines

Find the best approximation of $f(x) = \tanh(s(x - \pi))$ on $[0, 2\pi]$ in the space V with basis $\varphi_i(x) = sin((2i + 1)x)$, $i = 0, \ldots, N$. Make a movie showing how $u = \sum_{j=0}^{N} c_j \varphi_j(x)$ approximates $f(x)$ as N grows. Choose s such that f is steep ($s = 20$ may be appropriate).

Hint. One may naively call the `least_squares_orth` and `comparison_plot` from the `approx1D` module in a loop and extend the basis with one new element in each pass. This approach implies a lot of recomputations. A more efficient strategy is to let `least_squares_orth` compute with only one basis function at a time and accumulate the corresponding `u` in the total solution. Filename: `tanh_sines_approx1.py`.

Exercise 6: 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 Section 2 leads to a linear system whose solution coincides with the standard formulas for the coefficients in a Fourier series of $f(x)$ (see also Section 2.7). 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 $f(x) = \tanh(s(x - \frac{1}{2}))$ on $\Omega = [0, 1]$, which is a smooth function, but with considerable steepness around $x = 1/2$ as s grows in size. Calculate the coefficients in the Fourier expansion by solving the linear system, arising from the least squares or Galerkin methods, by hand. Plot some truncated versions of the series together with $f(x)$ to show how the series expansion converges for $s = 10$ and $s = 100$. Filename: `Fourier_approx.py`.

Exercise 7: Approximate a steep function by Lagrange polynomials

Use interpolation/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 $N = 3, 6, 9, 11$. Make separate plots of the approximation for each combination of s, point type (Chebyshev or uniform), and N. Filename: `tanh_Lagrange.py`.

Exercise 8: Define finite element meshes

Consider a domain $\Omega = [0, 2]$ divided into the three elements $[0, 1]$, $[1, 1.2]$, and $[1.2, 2]$, with two nodes in each element (P1 elements). Set up the list of

coordinates and nodes (`nodes`) and the numbers of the nodes that belong to each element (`elements`) in two cases: 1) nodes and elements numbered from left to right, and 2) nodes and elements numbered from right to left.

Thereafter, subdivide the element $[1.2, 2]$ into two new equal-sized elements. Add the new node and the two new elements to the data structures created above, and try to minimize the modifications. Filename: `fe_numberings.py`..

Exercise 9: Construct matrix sparsity patterns

Exercise 8 describes a element mesh with a total of five elements, but with two different element and node orderings. For each of the two orderings, make a 5×5 matrix and fill in the entries that will be nonzero.

Hint. A matrix entry (i, j) is nonzero if i and j are nodes in the same element. Filename: `fe_sparsity_pattern.pdf`.

Exercise 10: Perform symbolic finite element computations

Find formulas for the coefficient matrix and right-hand side when approximating $f(x) = sin(x)$ on $\Omega = [0, \pi]$ by two P1 elements of size $\pi/2$. Solve the system and compare $u(\pi/2$ with the exact value 1.

Filename: `sin_approx_P1.py`.

Exercise 11: Approximate a steep function by P1 and P2 elements

Given

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

use the Galerkin or least squares method with finite elements to find an approximate function $u(x)$. Choose $s = 40$ and try $n_e = 4, 8, 16$ P1 elements and $n_e = 2, 4, 8$ P2 elements. Integrate $f\varphi_i$ numerically. Filename: `tanh_fe_P1P2_approx.py`.

Exercise 12: Approximate a tanh function by P3 and P4 elements

Solve Exercise 11 using $n_e = 1, 2, 4$ P3 and P4 elements. How will a collocation/interpolation method work in this case with the same number of nodes? Filename: `tanh_fe_P3P4_approx.py`.

Exercise 13: Investigate the approximation errors in finite elements

A fundamental question is how accurate the finite element approximation is in terms of the cell length h and the degree d of the basis functions. We can

investigate this empirically by choosing an f function, say $f(x) = A\sin(\omega x)$ on $\Omega = [0, 2\pi/\omega]$, and compute the approximation error for a series of h and d values. The theory predicts that the error should behave as h^{d+1}. Use experiments to verify this asymptotic behavior (i.e., for small enough h). Filename: `Asinwt_interpolation_error.py`.

Exercise 14: Approximate a step function by finite elements

Approximate the step function

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

by 2, 4, and 8 P1 and P2 elements. Compare approximations visually.

Hint. This f can also be expressed in terms of the Heaviside function $H(x)$: $f(x) = H(x-1/2)$. Therefore, 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 this particular integrand, 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`. It is better to make special-purpose code for this case or perform all caluclations by hand. Filename: `Heaviside_approx_P1P2.py`..

Exercise 15: 2D approximation with orthogonal functions

Assume we have basis functions $\varphi_i(x, y)$ in 2D that are orthogonal such that $(\varphi_i, \varphi_j) = 0$ when $i \neq j$. The function `least_squares` in the file `approx2D.py` will then spend much time on computing off-diagonal terms in the coefficient matrix that we know are zero. To speed up the computations, make a version `least_squares_orth` that utilizes the orthogonality among the basis functions. Apply the function to approximate

$$f(x, y) = x(1-x)y(1-y)e^{-x-y}$$

on $\Omega = [0, 1] \times [0, 1]$ via basis functions

$$\varphi_i(x, y) = \sin(p\pi x)\sin(q\pi y), \quad i = qN_x + p\,.$$

Hint. Get ideas from the function `least_squares_orth` in Section 2.8 and file `approx1D.py`. Filename: `approx2D_lsorth_sin.py`.

Exercise 16: Use the Trapezoidal rule and P1 elements

Consider approximation of some $f(x)$ on an interval Ω using the least squares or Galerkin methods with P1 elements. Derive the element matrix and vector using

the Trapezoidal rule (102) for calculating integrals on the reference element. Assemble the contributions, assuming a uniform cell partitioning, and show that the resulting linear system has the form $c_i = f(x_i)$ for $i = 0, \ldots, N$. Filename: `fe_trapez.pdf`.

11 Basic principles for approximating differential equations

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 the coming sections. Extensions to time-dependent problems may, for instance, use finite difference approximations in time.

Before studying how finite element methods are used to tackle differential equation, we first look at how global basis functions and the least squares, Galerkin, and collocation principles can be used to solve differential equations.

11.1 Differential equation models

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

$$\mathcal{L}(u) = 0, \quad x \in \Omega\,. \tag{116}$$

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

$$\mathcal{L}(u) = \frac{d^2u}{dx^2} - f(x), \tag{117}$$

$$\mathcal{L}(u) = \frac{d}{dx}\left(a(x)\frac{du}{dx}\right) + f(x), \tag{118}$$

$$\mathcal{L}(u) = \frac{d}{dx}\left(a(u)\frac{du}{dx}\right) - \alpha u + f(x), \tag{119}$$

$$\mathcal{L}(u) = \frac{d}{dx}\left(a(u)\frac{du}{dx}\right) + f(u, x)\,. \tag{120}$$

Both $a(x)$ and $f(x)$ are considered as specified functions, while α is a prescribed parameter. Differential equations corresponding to (117)-(118) arise in diffusion phenomena, such as steady transport of heat in solids and flow of viscous fluids between flat plates. The form (119) arises when transient diffusion or wave phenomenon are discretized in time by finite differences. The equation (120) appear in chemical models when diffusion of a substance is combined with chemical reactions. Also in biology, (120) plays an important role, both for spreading of species and in 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 condition at each of the (two) boundary points, here abstractly specified as

$$\mathcal{B}_0(u) = 0,\ x = 0, \quad \mathcal{B}_1(u) = 0,\ x = L \tag{121}$$

There are three common choices of boundary conditions:

$$\mathcal{B}_i(u) = u - g, \qquad \text{Dirichlet condition}, \tag{122}$$

$$\mathcal{B}_i(u) = -a\frac{du}{dx} - g, \qquad \text{Neumann condition}, \tag{123}$$

$$\mathcal{B}_i(u) = -a\frac{du}{dx} - a(u - g), \qquad \text{Robin condition}\,. \tag{124}$$

Here, g and a are specified quantities.

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

$$\mathcal{L}(u_{\mathrm{e}}) = 0, \quad x \in \Omega, \tag{125}$$

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 the forthcoming examples is

$$-u''(x) = f(x), \quad x \in \Omega = [0, L], \quad u(0) = 0,\ u(L) = D\,. \tag{126}$$

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

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

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

$$-u''(x) = f(x), \quad x \in \Omega = [0, L], \quad u'(0) = C,\ u(L) = D\,. \tag{127}$$

A third variant has a variable coefficient,

$$-(a(x)u'(x))' = f(x), \quad x \in \Omega = [0, L], \quad u'(0) = C,\ u(L) = D\,. \tag{128}$$

11.2 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

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

for some unknown coefficients $c_0, \ldots, c_N$. (Later, in Section 13, 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.) As in Section 2, we need principles for deriving $N+1$ equations to determine the $N+1$ unknowns $c_0, \ldots, c_N$. A key idea of Section 2 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,

$$R = \mathcal{L}(u) = \mathcal{L}(\sum_j c_j \varphi_j), \tag{129}$$

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

$$R = R(x; c_0, \ldots, c_N)\,. \tag{130}$$

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

$$\int_\Omega R^2 dx \tag{131}$$

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

$$(f, g) = \int_\Omega f(x) g(x) dx, \tag{132}$$

the least-squares method can be defined as

$$\min_{c_0,\ldots,c_N} E = (R,R)\,. \tag{133}$$

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

$$\int_\Omega 2R\frac{\partial R}{\partial c_i}dx = 0 \quad \Leftrightarrow \quad (R,\frac{\partial R}{\partial c_i}) = 0, \quad i = 0,\ldots,N\,. \tag{134}$$

The Galerkin method. The least-squares principle is equivalent to demanding the error to be orthogonal to the space V when approximating a function f by $u \in V$. With a differential equation we do not know the true error so we must instead require the residual R to be orthogonal to V. This idea implies seeking $c_0,\ldots,c_N$ such that

$$(R,v) = 0, \quad \forall v \in V\,. \tag{135}$$

This is the Galerkin method for differential equations. As shown in (21) and (22), this statement is equivalent to R being orthogonal to the $N+1$ basis functions only:

$$(R,\varphi_i) = 0, \quad i = 0,\ldots,N, \tag{136}$$

resulting in $N+1$ 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, but not necessarily the same space as V where we seek the unknown function. This generalization is naturally called the *method of weighted residuals*:

$$(R,v) = 0, \quad \forall v \in W\,. \tag{137}$$

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

$$(R,w_i) = 0, \quad i = 0,\ldots,N\,. \tag{138}$$

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 (137) (or (138)) and (135) (or (136)) are known as *variational formulations*. These equations are in this text primarily 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 be an *infinite-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 and acts as 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 Section 13 we shall see that boundary conditions may lead to a difference between the test and trial spaces in the Galerkin method.

Remark. It may be subject to debate whether it is only the form of (137) or (135) after integration by parts, as explained in Section 11.4, 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 Section 11.9. 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.

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:

$$R(x_i; c_0, \ldots, c_N) = 0, \quad i = 0, \ldots, N\,. \tag{139}$$

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

$$\int_\Omega f(x)\delta(x - x_i)dx = f(x_i), \quad x_i \in \Omega\,. \tag{140}$$

Intuitively, we may think of $\delta(x - x_i)$ as a very peak-shaped function around $x = x_i$ with integral 1, roughly visualized in Figure 36. Because of (140), we can let $w_i = \delta(x - x_i)$ be weighting functions in the method of weighted residuals, and (138) becomes equivalent to (139).

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:

$$\int_{\Omega_i} R\,dx = 0, \quad i = 0, \ldots, N\,. \tag{141}$$

This statement can also be expressed as a weighted residual method

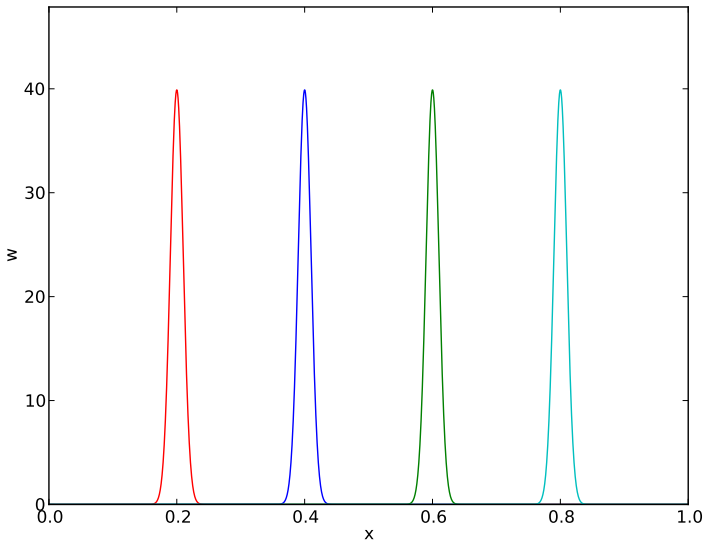

Figure 36: Approximation of delta functions by narrow Gaussian functions.

$$\int_\Omega Rw_i\,dx = 0, \quad i = 0, \ldots, N, \tag{142}$$

where $w_i = 1$ for $x \in \Omega_i$ and $w_i = 0$ otherwise.

11.3 Examples on using the principles

Let us now apply global basis functions to illustrate the principles for minimizing R.

The model problem. We consider the differential equation problem

$$-u''(x) = f(x), \quad x \in \Omega = [0, L], \quad u(0) = 0,\ u(L) = 0\,. \tag{143}$$

Basis functions. Our choice of basis functions φ_i for V is

$$\varphi_i(x) = \sin\left((i+1)\pi\frac{x}{L}\right), \quad i = 0, \ldots, N\,. \tag{144}$$

An important property of these functions is that $\varphi_i(0) = \varphi_i(L) = 0$, which means that the boundary conditions on u are fulfilled:

$$u(0) = \sum_j c_j\varphi_j(0) = 0, \quad u(L) = \sum_j c_j\varphi_j(L)\,.$$

Another nice property is that the chosen sine functions are orthogonal on Ω:

$$\int_0^L \sin\left((i+1)\pi\frac{x}{L}\right)\sin\left((j+1)\pi\frac{x}{L}\right)\,dx = \begin{cases} \frac{1}{2}L & i=j \\ 0, & i\neq j \end{cases} \tag{145}$$

provided i and j are integers.

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

$$\begin{aligned} R(x;c_0,\ldots,c_N) &= u''(x)+f(x), \\ &= \frac{d^2}{dx^2}\left(\sum_{j=0}^{N} c_j\varphi_j(x)\right)+f(x), \\ &= -\sum_{j=0}^{N} c_j\varphi_j''(x)+f(x)\,. \end{aligned} \tag{146}$$

The least squares method. The equations (134) in the least squares method require an expression for $\partial R/\partial c_i$. We have

$$\frac{\partial R}{\partial c_i} = \frac{\partial}{\partial c_i}\left(\sum_{j=0}^{N} c_j\varphi_j''(x)+f(x)\right) = \sum_{j=0}^{N}\frac{\partial c_j}{\partial c_i}\varphi_j''(x) = \varphi_i''(x)\,. \tag{147}$$

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

$$(\sum_j c_j\varphi_j''+f,\varphi_i'') = 0, \quad i=0,\ldots,N, \tag{148}$$

which can be rearranged as

$$\sum_{j=0}^{N}(\varphi_i'',\varphi_j'')c_j = -(f,\varphi_i''), \quad i=0,\ldots,N\,. \tag{149}$$

This is nothing but a linear system

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

with

$$
\begin{aligned}
A_{i,j} &= (\varphi_i'', \varphi_j'') \\
&= \pi^4 (i+1)^2 (j+1)^2 L^{-4} \int_0^L \sin\left((i+1)\pi\frac{x}{L}\right) \sin\left((j+1)\pi\frac{x}{L}\right)\, dx \\
&= \begin{cases} \frac{1}{2} L^{-3}\pi^4 (i+1)^4 & i = j \\ 0, & i \neq j \end{cases} \qquad (150)
\end{aligned}
$$

$$
b_i = -(f, \varphi_i'') = (i+1)^2 \pi^2 L^{-2} \int_0^L f(x) \sin\left((i+1)\pi\frac{x}{L}\right)\, dx \qquad (151)
$$

Since the coefficient matrix is diagonal we can easily solve for

$$
c_i = \frac{2L}{\pi^2 (i+1)^2} \int_0^L f(x) \sin\left((i+1)\pi\frac{x}{L}\right)\, dx\,. \qquad (152)
$$

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

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

according to WolframAlpha[13][14][15] (use `j` and not `i`, which means $= \sqrt{-1}$, when asking). Hence,

$$
c_i = \frac{4L^2(1 + (-1)^i)}{\pi^3 (i+1)^3}\,.
$$

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

$$
u(x) = \sum_{k=0}^{N/2} \frac{8L^2}{\pi^3 (2k+1)^3} \sin\left((2k+1)\pi\frac{x}{L}\right)\,. \qquad (153)
$$

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,

$$
u(x) \approx \frac{8L^2}{\pi^3} \sin\left(\pi\frac{x}{L}\right)\,.
$$

The Galerkin method. The Galerkin principle (135) applied to (143) consists of inserting our special residual (146) in (135)

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

or

[13] `http://wolframalpha.com`
[14] `http://wolframalpha.com`
[15] `http://wolframalpha.com`

$$(u'', v) = -(f, v), \quad \forall v \in V\,. \tag{154}$$

This is the variational formulation, based on the Galerkin principle, of our differential equation. The $\forall v \in V$ requirement is equivalent to demanding the equation $(u'', v) = -(f, v)$ to be fulfilled for all basis functions $v = \varphi_i$, $i = 0, \ldots, N$ (cf. (135) and (136)). This gives

$$(\sum_{j=0}^{N} c_j \varphi_j'', \varphi_i) = -(f, \varphi_i), \quad i = 0, \ldots, N\,. \tag{155}$$

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

$$\sum_{j=0}^{N} (\varphi_i, \varphi_j'') c_j = (f, \varphi_i), \quad i = 0, \ldots, N\,. \tag{156}$$

For the particular choice of the basis functions (144) we get in fact the same linear system as in the least squares method (because $\varphi'' = -(i+1)^2\pi^2 L^{-2}\varphi$).

The collocation method. For the collocation method (139) 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 or points uniformly distributed in the interior of Ω. Demanding the residual to vanish at these points leads, in our model problem (143), to the equations

$$-\sum_{j=0}^{N} c_j \varphi_j''(x_i) = f(x_i), \quad i = 0, \ldots, N, \tag{157}$$

which is seen to be a linear system with entries

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

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, resulting in $A_{0,0} = -\varphi_0''(x_0) = \pi^2 L^{-2}$, $f(x_0) = 2$, and hence $c_0 = 2L^2/\pi^2$.

Comparison. In the present model problem, with $f(x) = 2$, the exact solution is $u(x) = x(L - x)$, while for $N = 0$ the Galerkin and least squares method result in $u(x) = 8L^2\pi^{-3}\sin(\pi x/L)$ and the collocation method leads to $u(x) = 2L^2\pi^{-2}\sin(\pi x/L)$. Since all methods fulfill the boundary conditions $u(0) = u(L) = 0$, we expect the largest discrepancy to occur at the midpoint of the domain: $x = L/2$. The error at the midpoint becomes $-0.008L^2$ for the Galerkin and least squares method, and $0.047L^2$ for the collocation method.

11.4 Integration by parts

A problem arises if we want to use the finite element functions from Section 3 to solve our model problem (143) by the least squares, Galerkin, or collocation methods: the piecewise polynomials $\varphi_i(x)$ have discontinuous derivatives at the cell 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. (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. Also, differentiating discontinuous functions can actually be handled by distribution theory in mathematics.) The Galerkin method and the method of weighted residuals can, however, be applied together with finite element basis functions if we use *integration by parts* as a means for transforming a second-order derivative to a first-order one.

Consider the model problem (143) 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:

$$\begin{aligned}\int_0^L u''(x)v(x)dx &= -\int_0^L u'(x)v'(x)dx + [vu']_0^L \\ &= -\int_0^L u'(x)v'(x)dx + u'(L)v(L) - u'(0)v(0)\,. \end{aligned} \tag{158}$$

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

$$\begin{aligned}\int_0^L \varphi_i(x)\varphi_j''(x)dx &= -\int_0^L \varphi_i'(x)\varphi_j'(x)dx + [\varphi_i\varphi_j']_0^L \\ &= -\int_0^L \varphi_i'(x)\varphi_j'(x)dx + \varphi_i(L)\varphi_j'(L) - \varphi_i(0)\varphi_j'(0)\,. \end{aligned} \tag{159}$$

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 Section 14, integration by parts also provides a method for implementing boundary conditions involving u'.

With the choice (144) of basis functions we see that the "boundary terms" $\varphi_i(L)\varphi_j'(L)$ and $\varphi_i(0)\varphi_j'(0)$ vanish since $\varphi_i(0) = \varphi_i(L) = 0$. A boundary term associated with a location at the boundary where we have Dirichlet conditions will always vanish because $\varphi_i = 0$ at such locations.

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.

11.5 Boundary function

So far we have assumed zero Dirichlet boundary conditions, typically $u(0) = u(L) = 0$, and we have demanded that $\varphi_i(0)\varphi_i(L) = 0$ for $i = 0, \ldots, N$. What about a boundary condition like $u(L) = D \neq 0$? This condition immediately faces a problem with $u = \sum_{j=0}^{N} c_j\varphi_j$, because $u(L) = 0$ since all the basis functions vanish at $x = L$.

A boundary condition of the form $u(L) = D$ can be implemented by demanding that all $\varphi_i(L) = 0$, but adding a *boundary function* $B(x)$ with the right boundary value, $B(L) = D$, to the expansion for u:

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

This u gets the right value at $x = L$:

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

The idea is that for any boundary where u is known we demand φ_i to vanish and construct a function $B(x)$ to attain the boundary value of u. There are no restrictions how $B(x)$ varies with x in the interior of the domain, so this variation needs to be constructed in some way.

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

$$u(x) = \frac{x}{L}D + \sum_{j=0}^{N} c_j \varphi_j(x), \tag{160}$$

with $\varphi_i(0) = \varphi_i(L) = 0$.

The $B(x)$ function can be chosen in many ways as long as its boundary values are correct. For example, $B(x) = D(x/L)^p$ for any power p will work fine in the above example. As another example, consider a domain $\Omega = [a, b]$ where the boundary conditions are $u(a) = U_a$ and $u(b) = U_b$. A class of possible $B(x)$ functions is

$$B(x) = U_a + \frac{U_b - U_a}{(b-a)^p}(x-a)^p, \quad p > 0\,.$$

To summarize, the procedure goes 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 $\partial\Omega_E$ and seek u as

$$u(x) = B(x) + \sum_{j=0}^{N} c_j \varphi_j(x), \tag{161}$$

where $B(x)$ equals the boundary conditions on u at $\partial\Omega_E$.

Remark. 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.

11.6 Abstract notation for variational formulations

We have seen that variational formulations end up with a formula involving u and v, such as (u', v') and a formula involving v and known functions, such as (f, v). A common notation is to introduce an abstract variational statement written as $a(u, v) = L(v)$, where $a(u, v)$ is a so-called *bilinear form* involving all the terms that contain both the test and trial function, while $L(v)$ is a *linear form* containing all the terms without the trial function. For example, the statement

$$\int_\Omega u'v'dx = \int_{Omega} fvdx \quad \text{or} \quad (u', v') = (f, v) \quad \forall v \in V$$

can be written in abstract form: *find u such that*

$$a(u, v) = L(v) \quad \forall v \in V,$$

where we have the definitions

$$a(u,v) = (u', v'), \quad L(v) = (f, v)\,.$$

The term *linear* means that $L(\alpha_1 v_1 + \alpha_2 v_2) = \alpha_1 L(v_1) + \alpha_2 L(v_2)$ for two test functions v_1 and v_2, and scalar parameters α_1 and α_2. Similarly, the term *bilinear* means that $a(u,v)$ are linear in both its arguments:

$$a(\alpha_1 u_1 + \alpha_2 u_2, v) = \alpha_1 u(u_1, v) + \alpha_2 a(u_2, v), \quad a(u, \alpha_1 v_1 + \alpha_2 v_2) = \alpha_1 u(u, v_1) + \alpha_2 a(u, v_2)\,.$$

In nonlinear problems these linearity properties do not hold in general and the abstract notation is then $F(u; v) = 0$.

The matrix system associated with $a(u,v) = L(v)$ can also be written in an abstract form by inserting $v = \varphi_i$ and $u = \sum_j c_j \varphi_j$ in $a(u,v) = L(v)$. Using the linear properties, the system becomes

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

From this we see that the matrix element $A_{i,j}$ equals $a(\varphi_j, \varphi_i)$. In many problems, $a(u,v)$ is symmetric so that $a(\varphi_j, \varphi_i) = a(\varphi_i, \varphi_j)$.

The abstract notation $a(u,v) = L(v)$ for linear problems is much used in the literature and in description of finite element software (in particular the FEniCS[16][17] documentation). We shall frequently summarize variational forms using this notation.

11.7 More examples on variational formulations

Variable coefficient. Consider the problem

$$-\frac{d}{dx}\left(a(x)\frac{du}{dx}\right) = f(x), \quad x \in \Omega = [0, L],\ u(0) = C,\ u(L) = D\,. \tag{162}$$

There are two new features of this problem compared with previous examples: a variable coefficient $a(x)$ and nonzero Dirichlet conditions at both boundary points.

Let us first deal with the boundary conditions. We seek

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

with $\varphi_i(0) = \varphi_i(L) = 0$ for $i = 0, \ldots, N$. The function $B(x)$ must then fulfill $B(0) = C$ and $B(L) = D$. How B varies in between $x = 0$ and $x = L$ is not of importance. One possible choice is

[16] `http://fenicsproject.org`
[17] `http://fenicsproject.org`

$$B(x) = C + \frac{1}{L}(D - C)x\,.$$

The residual arises by inserting our u in the differential equation:

$$R = -\frac{d}{dx}\left(a\frac{du}{dx}\right) - f\,.$$

Galerkin's method is

$$(R, v) = 0, \quad \forall v \in V,$$

or written with explicit integrals,

$$\int_\Omega \left(\frac{d}{dx}\left(a\frac{du}{dx}\right) - f\right) v dx = 0, \quad \forall v \in V\,.$$

We proceed with integration by parts to lower the derivative from second to first order:

$$-\int_\Omega \frac{d}{dx}\left(a(x)\frac{du}{dx}\right) v dx = \int_\Omega a(x)\frac{du}{dx}\frac{dv}{dx}dx - \left[a\frac{du}{dx}v\right]_0^L\,.$$

Since all functions in v have the property $v(0) = v(L) = 0$, the boundary term vanishes. The variational formulation is then

$$\int_\Omega a(x)\frac{du}{dx}\frac{dv}{dx}dx = \int_\Omega f(x)v dx, \quad \forall v \in V\,.$$

The variational formulation can alternatively be written in a more compact form:

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

The corresponding abstract notation reads

$$a(u, v) = L(v) \quad \forall v \in V,$$

with

$$a(u, v) = (au', v'), \quad L(v) = (f, v)\,.$$

Note that the a in the notation $a(\cdot, \cdot)$ is not to be mixed with the variable coefficient $a(x)$ in the differential equation.

We may insert $u = B + \sum_j c_j\varphi_j$ and $v = \varphi_i$ to derive the linear system:

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

Isolating everything with the c_j coefficients on the left-hand side and all known terms on the right-hand side gives

$$\sum_{j=0}^{N}(a\varphi_j',\varphi_i')c_j = (f,\varphi_i) + (aCL^{-1},\varphi_i'),\quad i=0,\ldots,N\,.$$

This is nothing but a linear system $\sum_j A_{i,j}c_j = b_i$ with

$$A_{i,j} = (a\varphi_j',\varphi_i') = \int_\Omega a(x)\varphi_j'(x),\varphi_i'(x)dx,$$
$$b_i = (f,\varphi_i) + (aCL^{-1},\varphi_i') = \int_\Omega \left(f(x)\varphi_i(x) + a(x)\frac{C}{L}\varphi_i'(x)\right)dx\,.$$

First-order derivative in the equation and boundary condition. The next problem to formulate in variational form reads

$$-u''(x) + bu'(x) = f(x),\quad x\in\Omega=[0,L],\ u(0)=C,\ u'(L)=E\,. \tag{163}$$

The new features are a first-order derivative u' in the equation and the boundary condition involving the derivative: $u'(L)=E$. Since we have a Dirichlet condition at $x=0$, we force $\varphi_i(0)=0$ and use a boundary function $B(x)=C(L-x)/L$ to take care of the condition $u(0)=C$. Because there is no Dirichlet condition on $x=L$ we do not make any requirements to $\varphi_i(L)$. The expansion for u becomes

$$u = \frac{C}{L}(L-x) + \sum_{j=0}^{N} c_j\varphi_i(x)\,.$$

The variational formulation arises from multiplying the equation by a test function $v\in V$ and integrating over Ω:

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

We apply integration by parts to the $u''v$ term only. Although we could also integrate $u'v$ by parts, this is not common. The result becomes

$$(u' + bu', v') = (f,v) + [u'v]_0^L,\quad \forall v\in V\,.$$

Now, $v(0)=0$ so $[u'v]_0^L = u'(L)v(L) = Ev(L)$. We realize that when integrating by parts, *the boundary term can be used to implement Neumann conditions.*

Another very important result is that *omitting the boundary term* implies, in general, that we actually impose the condition $u'=0$ unless there is a Dirichlet condition at that point! This result has great practical consequences, because it is easy to forget the boundary term, and this mistake implicitly sets a boundary condition. Since Neumann conditions can be incorporated without doing anything, or simply inserting the condition in a boundary term, such conditions are

known as *natural boundary conditions*. Dirichlet conditions requires more essential steps in the mathematical formulation and are therefore known as *essential boundary conditions*.

The variational form now reads

$$(u', v') + (bu', v) = (f, v) + Ev(L), \quad \forall v \in V\,.$$

In the abstract notation we have

$$a(u, v) = L(v) \quad \forall v \in V,$$

with the particular formulas

$$a(u, v) = (u', v') + (bu', v), \quad L(v) = (f, v) + Ev(L)\,.$$

The associated linear system is derived by inserting $u = B + \sum_j c_j\varphi_j$ and replacing v by φ_i for $i = 0, \ldots, N$. Some algebra results in

$$\sum_{j=0}^{N} \underbrace{((\varphi_j', \varphi_i') + (b\varphi_j', \varphi_i))}_{A_{i,j}} c_j = \underbrace{(f, \varphi_i) + (bCL^{-1}, \varphi_i') + E\varphi_i(L)}_{b_i}\,.$$

Observe that in this problem, the coefficient matrix is not symmetric, because of the term

$$(b\varphi_j', \varphi_i) = \int_\Omega b\varphi_j'\varphi_i dx \neq \int_\Omega b\varphi_i'\varphi_j dx = (\varphi_i', b\varphi_j)\,.$$

For finite element basis functions, it is worth noticing that the boundary term $E\varphi_i(L)$ is nonzero only in the entry b_N since all φ_i, $i \neq N$, are zero at $x = L$, provided the degrees of freedom are numbered from left to right in 1D so that $x_N = L$.

11.8 Example on computing with Dirichlet and Neumann conditions

Let perform the necessary calculations to solve

$$-u''(x) = 2, \quad x \in \Omega = [0, 1], \quad u'(0) = C,\ u(1) = D,$$

using a global polynomial basis $\varphi_i \sim x^i$. 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 $B(x) = Dx$. The variational formulation becomes

$$(u', v') = (2, v) - Cv(0) \quad \forall v \in V\,.$$

The entries in the linear system are then

$$A_{i,j} = (\varphi_j, \varphi_i) = \int_0^1 \varphi_i'(x)\varphi_j'(x)dx = \int_0^1 (i+1)(j+1)(1-x)^{i+j}dx,$$

$$b_i = (2, \varphi_i) - (D, \varphi_i') - C\varphi_i(0) = \int_0^1 \left(2\varphi_i(x) - D\varphi_i'(x)\right)dx - C\varphi_i(0) = \int_0^1 \left(2(1-x)^{i+1} - D(i+1)(1-\right.$$

With $N = 1$ we can calculate the global matrix system to be

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

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

$$u(x) = 1 - x^2 + D + C(x-1), \tag{164}$$

The exact solution is easily obtained by integrating twice and applying the boundary conditions:

```
>>> from sympy import *
>>> x, C, D = symbols('x C D')
>>> f = 2
>>> f1 = integrate(f, x)
>>> f2 = integrate(f1, x)
>>> C1, C2 = symbols('C1 C2')       # integration constants
>>> u = -f2 + C1*x + C2
>>> BC1 = diff(u,x).subs(x, 0) - C  # x=0 condition
>>> BC2 = u.subs(x,1) - D           # x=1 condition
>>> s = solve([BC1, BC2], [C1, C2])
>>> u_e = -f2 + s[C1]*x + s[C2]
>>> print u_e
C*x - C + D - x**2 + 1
>>> simplify(-diff(u, x, x) - f)  # check diff.eq.
0
>>> diff(u, x).subs(x, 0) - C     # check x=0 condition
0
>>> u.subs(x, L) - D              # check x=L condition
0
```

We observe that the numerical solution coincides with the exact one, which is to be expected since the expansion for u contains the exact solution as special case.

Nonlinear terms. Finally, we show that the techniques used above to derive variational forms also apply in nonlinear cases. Here is a model problem with a nonlinear coefficient and right-hand side:

$$-\left(a(u)u'\right)' = f(u), \quad x \in [0, L],\ u(0) = 0,\ u'(L) = E\,. \tag{165}$$

Using the Galerkin principle, we multiply by $v \in V$ and integrate,

$$-\int_0^L \frac{d}{dx}\left(a(u)\frac{du}{dx}\right)vdx = \int_0^L f(u)vdx \quad \forall v \in V\,.$$

The integration by parts does not differ from the case where we had an $a(x)$ and not an $a(u)$:

$$\int_0^L a(u)\frac{du}{dx}\frac{dv}{dx}vdx = \int_0^L f(u)vdx + [vu']_0^L \quad \forall v \in V\,.$$

We require that $v(0) = 0$ since $u(0)$ is known. The other term, $v(L)u'(L)$, is used to impose the other boundary condition $u'(L) = E$, resulting in

$$\int_0^L a(u)\frac{du}{dx}\frac{dv}{dx}vdx = \int_0^L f(u)vdx + vE \quad \forall v \in V,$$

or alternatively written more compactly as

$$(a(u)u', v') = (f(u), v) + vE \quad \forall v \in V\,.$$

Since the problem is nonlinear, we cannot identify $a(u, v)$ and $L(v)$. An abstract notation is typically *find u such that*

$$F(u; v) = 0 \quad \forall v \in V,$$

with

$$F(u; v) = (a(u)u', v') - (f(u), v) - vE\,.$$

By inserting $u = \sum_j c_j\varphi_j$ we get a *nonlinear system of algebraic equations* for the unknowns $c_0, \ldots, c_N$. Such systems must be solved by constructing a sequence of linear systems whose solutions converge to the solution of the nonlinear system. Frequently applied methods are Picard iteration and Newton's method.

11.9 Variational problems and optimization of functionals

If $a(u, v) = a(v, u)$, it can be shown that the variational statement $a(u, v) = L(v) \quad \forall v \in V$ is equivalent to minimizing the functional

$$F(v) = \frac{1}{2}a(v, v) - L(v)$$

over all functions $v \in V$. That is,

$$F(u) \leq F(v) \quad \forall v \in V\,.$$

Inserting a $v = \sum_j c_j\varphi_j$ turns minimization of $F(v)$ into minimization of a quadratic function

$$\bar{F}(c_0, \ldots, c_N) = \sum_{j=0}^N \sum_{i=0}^N a(\varphi_i, \varphi_j)c_ic_j - \sum_{j=0}^N L(\varphi_j)c_j$$

of $N + 1$ parameters.

Many traditional applications of the finite element method, especially in solid mechanics and structural analysis, start with formulating $F(v)$ from physical principles, such as minimization of energy, and then proceeds with deriving $a(u, v) = L(v)$, which is the equation usually desired in implementations.

12 Computing with finite elements

The purpose of this section is to demonstrate in detail how the finite element method can the be applied to the model problem

$$-u''(x) = 2, \quad x \in (0, L),\ u(0) = u(L) = 0,$$

with variational formulation

$$(u', v') = (2, v) \quad \forall v \in V\,.$$

The variational formulation is derived in (158).

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: the remaining φ_i are all zero at $x = 0$ and $x = L$. This means that only $c_1, \ldots, c_{N-1}$ are unknowns and the variational statement in (158) holds only for $i = 1, \ldots, N-1$. For simplicity, we introduce uniformly spaced nodes, numbered from left to right:

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

The simplest choice of elements is P1 elements with piecewise linear functions. The nodes then coincides with the cell vertices and $\Omega^{(e)} = [x_e, x_{e+1}]$.

12.1 Computation in the global physical domain

We shall first perform a computation in the x coordinate system because the integrals can be easily computed here by simple, visual, 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 entries in the coefficient matrix and right-hand side are

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

The $\varphi_i(x)$ function is specified in (54) on page 34. However, we need the derivative of $\varphi_i(x)$ to compute the coefficient matrix. These are

$$\varphi_i'(x) = \begin{cases} 0, & x < x_{i-1}, \\ h^{-1}, & x_{i-1} \leq x < x_i, \\ -h^{-1}, & x_i \leq x < x_{i+1}, \\ 0, & x \geq x_{i+1} \end{cases} \tag{166}$$

Figure 37 shows $\varphi_1'(X)$ and $\varphi_2'(X)$.

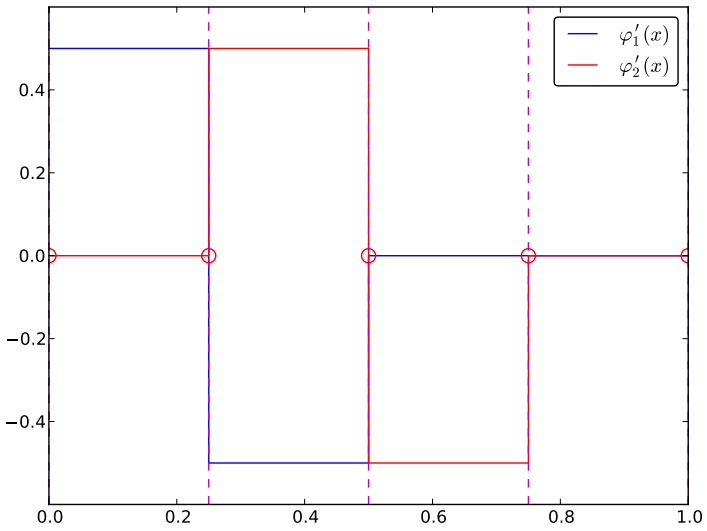

Figure 37: Illustration of the derivative of piecewise linear basis functions associated with nodes in cell 1.

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

$$
\begin{aligned}
A_{i-1,i} &= \int_0^L \varphi_{i-1}'(x)\varphi_i'(x)dx, \\
A_{i,i} &= \int_0^L \varphi_i'(x)^2 dx, \\
A_{i,i+1} &= \int_0^L \varphi_i'(x)\varphi_{i+1}'(x)dx,
\end{aligned}
$$

for $i = 1, \ldots, N-1$. We see that $\varphi_{i-1}'(x)$ and $\varphi_i'(x)$ have overlap of one cell $\Omega^{(i-1)} = [x_{i-1}, x_i]$ and that their product then is $-1/h^2$. The integrand is constant and therefore $A_{i-1,i} = -h^{-2}h = -h^{-1}$. 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 cells, $\Omega^{(i-1)} = [x_{i-1}, x_i]$ and $\Omega^{(i)} = [x_i, x_{i+1}]$, but $\varphi_i'(x)^2 = h^{-2}$ in both cells, and the length of the integration interval is $2h$ so we get $A_{i,i} = 2h^{-1}$. The right-hand side involves an integral of $\varphi_i(x)$, $i = 1, \ldots, N-1$, which is just the area under a hat function of height 1 and width $2h$, i.e., equal to h. Hence, $b_i = 2h$.

Note that there are no entries $A_{0,0}$ and $A_{N,N}$ since c_0 and c_N are left out

of the matrix system. The equation system to be solved only involves the unknowns $c_1, \ldots, c_{N-1}$, the right-hand side coefficients $b_1, \ldots, b_{N-1}$, and the matrix entries $A_{1,1}$, $A_{1,2}$, $A_{2,1}$, $A_{2,2}$, ..., $A_{N-1,N-1}$, and $A_{N-1,N-1}$. We can collect these numbers in a matrix system that takes the following form:

$$\frac{1}{h}\begin{pmatrix} 2 & -1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} c_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_{N-1} \end{pmatrix} = \begin{pmatrix} 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \end{pmatrix} \tag{167}$$

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

$$-\frac{1}{h}u_{i-1} + \frac{2}{h}u_i - \frac{1}{h}u_{i+1} = 2h\,. \tag{168}$$

A finite difference discretization of $-u''(x) = 2$ by a centered, second-order finite difference approximation $u''(x_i) \approx [D_x D_x u]_i$ with $\Delta x = h$ yields

$$-\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} = 2, \tag{169}$$

which is, in fact, equivalent to (168) if (168) 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, especially in time-dependent problems.

12.2 Elementwise computations

We now employ the element by element (or cell by cell) computational procedure as explained in Sections 3.4, 3.5, and 3.6. All integrals need to be mapped to the local reference coordinate system (X) according to Section 3.5. In the present case, the matrix entries contain derivatives with respect to x,

$$A^{(e)}_{i,j} = \int_{\Omega^{(e)}} \varphi_i'(x)\varphi_j'(x)dx = \int_{-1}^{1} \frac{d}{dx}\tilde{\varphi}_r(X)\frac{d}{dx}\tilde{\varphi}_s(X)\frac{h}{2}dX, \quad i = q(e,r),\ j = q(e,s),\ r,s = 1,2\,.$$

The basis functions $\tilde{\varphi}_r(X)$ are known as functions of X. However, we now need to find the derivative of $\tilde{\varphi}_r(X)$ with respect to x. Given

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

we can easily compute $d\tilde{\varphi}_r/dX$:

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

From the chain rule,

$$\frac{d\tilde{\varphi}_r}{dx} = \frac{d\tilde{\varphi}_r}{dX}\frac{dX}{dx} = \frac{2}{h}\frac{d\tilde{\varphi}_r}{dX}\,. \tag{170}$$

The transformed integral is then

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

The right-hand side is transformed according to

$$b^{(e)}_i = \int_{\Omega^{(e)}} 2\varphi_i(x)dx = \int_{-1}^{1} 2\tilde{\varphi}_r(X)\frac{h}{2}dX, \quad i = q(e,r),\ r = 1,2\,.$$

Specifically for P1 elements we arrive at the following calculations for the element matrix entries:

$$\begin{aligned}
\tilde{A}^{(e)}_{0,0} &= \int_{-1}^{1} \frac{2}{h}\left(-\frac{1}{2}\right)\frac{2}{h}\left(-\frac{1}{2}\right)\frac{2}{h}dX = \frac{1}{h}\\
\tilde{A}^{(e)}_{0,1} &= \int_{-1}^{1} \frac{2}{h}\left(-\frac{1}{2}\right)\frac{2}{h}\left(\frac{1}{2}\right)\frac{2}{h}dX = -\frac{1}{h}\\
\tilde{A}^{(e)}_{1,0} &= \int_{-1}^{1} \frac{2}{h}\left(\frac{1}{2}\right)\frac{2}{h}\left(-\frac{1}{2}\right)\frac{2}{h}dX = -\frac{1}{h}\\
\tilde{A}^{(e)}_{1,1} &= \int_{-1}^{1} \frac{2}{h}\left(\frac{1}{2}\right)\frac{2}{h}\left(\frac{1}{2}\right)\frac{2}{h}dX = \frac{1}{h}
\end{aligned}$$

The element vector entries become

$$\begin{aligned}
\tilde{b}^{(e)}_0 &= \int_{-1}^{1} 2\frac{1}{2}(1-X)\frac{h}{2}dX = h\\
\tilde{b}^{(e)}_1 &= \int_{-1}^{1} 2\frac{1}{2}(1+X)\frac{h}{2}dX = h\,.
\end{aligned}$$

Expressing these entries in matrix and vector notation, we have

$$\tilde{A}^{(e)} = \frac{1}{h}\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(e)} = h\begin{pmatrix} 1 \\ 1 \end{pmatrix}. \tag{171}$$

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 $\tilde{A}^{(0)}_{1,1}$ and $\tilde{A}^{(N-1)}_{0,0}$ from the first and last element, respectively. The result becomes identical to (167), which is not surprising since the procedures are mathematically equivalent.

A fundamental problem with the matrix system we have assembled is that the boundary conditions are not incorporated if $D \neq 0$. The next sections deals with this issue.

13 Boundary conditions: specified value

We have to take special actions to incorporate Dirichlet conditions, such as $u(L) = D$, into the computational procedures. The present section outlines alternative, but mathematically equivalent, methods.

13.1 General construction of a boundary function

In Section 13.1 we introduced a boundary function $B(x)$ to deal with nonzero Dirichlet boundary conditions for u. The construction of such a function is not always trivial, especially not in multiple dimensions. However, a simple and general construction idea exists when the basis functions have the property

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

where x_j is a boundary point. Examples on such functions are the Lagrange interpolating polynomials and finite element functions. With $\Omega = [0, L]$, $u(0) = U_0$, and $u(L) = U_N$ we can then let

$$B(x) = U_0\varphi_0(x) + U_N\varphi_N(x)\,. \tag{172}$$

Since $\varphi_0(x_0 = 0) = 1$ and $\varphi_N(x_0 = 0) = 0$, we have $B(0) = U_0$. Similarly, since $\varphi_0(x_N = L) = 0$ and $\varphi_N(x_N = L) = 1$, we have $B(L) = U_N$.

In fact, the construction of $B(x)$ identifies $c_0 = U_0$ and $c_N = U_N$. The remaining parameters $c_1, \ldots, c_{N-1}$ therefore the unknowns in the linear system. For the unknown $u(x)$ we then let the expansion be

$$u(x) = U_0\varphi_0(x) + U_N\varphi_N(x) + \sum_{j=1}^{N-1} c_j\varphi_j(x)\,. \tag{173}$$

In the case where u has a Dirichlet boundary condition 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, because it allows us u to easily fulfill boundary values on a boundary of any geometrical complexity.

Example. Let us see how our previous model problem $-u'' = 2$, $u(0) = 0$, $u(L) = D$, is affected by a $B(x)$ to incorporate boundary values. The expansion for $u(x)$ reads

$$u(x) = 0 \cdot \varphi_0(x) + D\varphi_N(x) + \sum_{j=1}^{N-1} c_j \varphi_j(x)\,.$$

Inserting this expression in $-(u'', \varphi_i) = (f, \varphi_i)$ and integrating by parts results in a linear system with

$$A_{i,j} = \int_0^L \varphi_i'(x)\varphi_j'(x)dx, \quad b_i = \int_0^L (f(x) - D\varphi_N'(x))\varphi_i(x)dx,$$

for $i, j = 1, \ldots, N-1$. The integral $D\int_0^L \varphi_N'(x))\varphi_i(x)dx$ can only get a nonzero contribution from the last cell, $\Omega^{(N-1)} = [x_{N-1}, x_N]$ since $\varphi_N(x) = 0$ on all other cells. Moreover, $\varphi_N'(x)\varphi_i(x)dx$ only for $i = N-1$ and $i = N$, but the latter value is not included anymore. From the explanations of the calculations in Section 3.3 we find that $\int_0^L \varphi_N(x))\varphi_{N-1}(x)dx$ must equal $h/6$, resulting in the need to add $-Dh/6$ to b_{N-1}.

As an alternative, we now turn to *elementwise computations* and realize that $B(x) = D\varphi_N = 0$ on all cells except the last one. On the last cell we regard the local degree of freedom c_1 as known ($c_1 = D$). There is only one unknown in a P1 element and the element matrix is therefore a 1×1 matrix. We have $\tilde{A}_{0,0}^{(N-1)} = 1/h$ as in the other elements, while

$$\tilde{b}_0^{(N-1)} = \int_{-1}^{1} (f - D\tilde{\varphi}_1)\tilde{\varphi}_0 \frac{h}{2} = h - Dh/6\,.$$

When assembling these contributions, we see that b_{N-1} gets an extra term $-Dh/6$, as in the computations in the physical domain.

13.2 Modification of the linear system

From an implementational point of view, there is a convenient alternative to adding the $B(x)$ function and considering only the c_i coefficients corresponding to nodes without Dirichlet conditions as unknowns. Since Dirichlet values of u basically means that we know the corresponding c_i values, we may assemble the entire linear system for all degrees of freedom, without taking Dirichlet conditions into account, and then modify the linear system such that the c_i values corresponding to Dirichlet conditions get their right values.

Consider the computation of the global coefficient matrix (**??**). If we include all the $c_0, \ldots, c_N$ values in the system, we have some additional matrix entries from the first and last cell. We start with

$$A_{0,0} = \int_0^L \left(\frac{d\varphi_0}{dx}\right)^2 dx = \frac{1}{h},$$

because the interval of integration, with nonzero contributions, is just the first cell (and not two cells as for the other $A_{i,j}$ associated with cells not touching the boundaries). Similarly, in the last cell we get $A_{N,N} = 1/h$. Furthermore, $A_{0,1} = \int_0^L \varphi_0\varphi_1' dx = -1/h$. A similar reasoning is used for $A_{N-1,N} = \int_0^L \varphi_{N-1}\varphi_N' dx = -1/h$.

Regarding the right-hand side, we must include b_0 and b_N. Because $\varphi_0 = 0$ on all cells except the first one and $\varphi_N = 0$ on all cells except the last one, b_0 and b_N only involves integration over the first and last element, respectively:

$$b_0 = \int_0^L 2\varphi_0(x)dx = h,$$
$$b_N = \int_0^L 2\varphi_N(x)dx = h\,.$$

The complete matrix system, involving all degrees of freedom, takes the form

$$\frac{1}{h}\begin{pmatrix} 1 & -1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} h \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \\ h \end{pmatrix} \tag{174}$$

The idea now is that we replace the first and last equation by the equations $c_0 = 0$ and $c_N = D$, which guarantees that the boundary values of u becomes correct. This action changes the system to

$$\frac{1}{h}\begin{pmatrix} 1 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} 0 \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \\ D \end{pmatrix} \tag{175}$$

13.3 Symmetric modification of the linear system

The original matrix system (167) is symmetric, but the modifications in (175) destroy the symmetry. Our described modification will in general destroy an initial symmetry in the matrix system. This is not a particular computational disadvantage for tridiagonal systems arising in 1D problems, but may be more serious in 2D and 3D when the systems are large and exploiting symmetry can be very important for the solution method. Therefore, a modification that preserves symmetry is frequently applied.

Let c_k be a coefficient corresponding to a known value K. We want to replace equation k in the system by $c_k = K$, i.e., insert zeroes in row number k in the coefficient matrix, set 1 on the diagonal, and replace b_k by K. A symmetry-preserving modification consists in first subtracting column number k in the coefficient matrix, i.e., $A_{i,k}$ for $i = 0, \ldots, N$, times the boundary value K, from the right-hand side: $b_i \leftarrow b_i - A_{i,k}K$. Then we put zeroes in row number k *and* column number k in the coefficient matrix, and finally set $b_k = K$.

This modification goes as follows for the above system. First we subtract the first column in the coefficient matrix, times the boundary value, from the right-hand side. Because $c_0 = 0$, this subtraction has no effect. Then we subtract the last column, times the boundary value D, from the right-hand side. This action results in $b_{N-1} = 2h + D/h$ and $b_N = h - 2D/h$. Thereafter, we place zeros in the first and last row and column in the coefficient matrix and 1 on the two corresponding diagonal entries. Finally, we set $b_0 = 0$ and $b_N = D$. The result becomes

$$\frac{1}{h}\begin{pmatrix} 1 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} 0 \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h + D/h \\ D \end{pmatrix} \tag{176}$$

13.4 Modification of the element matrix and vector

The modifications of the global linear system can alternatively be done for the element matrix and vector. (The assembled system will get n on the main diagonal if n elements contribute to the same unknown, but the factor n will also appear on the right-hand side and hence cancel out.)

We have, in the present computational example, the element matrix and vector (171). The modifications are needed in cells where one of the degrees of freedom is known. Here, this means the first and last cell. In the first cell, local degree of freedom number 0 is known and the modification becomes

$$\tilde{A}^{(0)} = A = \frac{1}{h}\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(0)} = \begin{pmatrix} 0 \\ h \end{pmatrix}. \tag{177}$$

In the last cell we set

$$\tilde{A}^{(N-1)} = A = \frac{1}{h}\begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}, \quad \tilde{b}^{(N-1)} = \begin{pmatrix} h \\ D \end{pmatrix}. \tag{178}$$

We can also perform the symmetric modification. This operation affects only the last cell with a nonzero Dirichlet condition. The result becomes

$$\tilde{A}^{(N-1)} = A = \frac{1}{h}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \tilde{b}^{(N-1)} = \begin{pmatrix} h + D/h \\ D \end{pmatrix}. \tag{179}$$

The reader should assemble the element matrices and vectors and check that the result coincides with the system (176).

14 Boundary conditions: specified derivative

Suppose our model problem $-u''(x) = f(x)$ features the boundary conditions $u'(0) = C$ and $u(L) = D$. As already indicated in Section 11.7, the former

condition can be incorporated through the boundary term that arises from integration by parts. This details of this method will now be illustrated in the context of finite elements.

14.1 The variational formulation

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

$$\int_0^L u'(x)'\varphi_i'(x)dx - (u'(L)\varphi_i(L) - u'(0)\varphi_i(0)) = \int_0^L f(x)\varphi_i(x)dx,$$

for $i = 0, \ldots, N$. The first boundary term vanishes since $\varphi_i(L) = 0$ when $u(L)$ is known. Or, if we prefer to modify the linear system and include φ_N in the calculations ($\varphi_N(L) = 1 \neq 0$), it appears that the only nonzero term $u'(L)\varphi_N(L)$ is erased on the right-hand side when we set $b_N = D$. With either approach to incorporating Dirichlet conditions, the term $u'(L)\varphi_i(L)$ does not contribute to the final matrix system.

The second boundary term, $u'(0)\varphi_i(0)$, can be used to implement the condition $u'(0) = C$, provided $\varphi_i(0) \neq 0$ for some i (but with finite elements we fortunately have $\varphi_0(0) = 1$). 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

$$u(x) = B(x) + \sum_{j=0}^{N-1} c_j\varphi_j(x), \quad B(x) = D\varphi_N(x),$$

leads to the linear system

$$\sum_{j=0}^{N-1}\left(\int_0^L \varphi_i'(x)\varphi_j'(x)dx\right)c_j = \int_0^L \left(f(x)\varphi_i(x) - D\varphi_N'(x)\varphi_i(x)\right)dx - E\varphi_i(0), \tag{180}$$

for $i = 0, \ldots, N-1$. Alternatively, we may just work with

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

and modify the last equation to $c_N = D$ in the linear system. The linear system to be assembled then corresponds to (180) with the $D\varphi_N'$ term removed, and $j =$

N must be included in the summation. Similarly, the equation corresponding to $i = N$ must be included:

$$\sum_{j=0}^{N}\left(\int_0^L \varphi_i'(x)\varphi_j'(x)dx\right)c_j = \int_0^L (f(x)\varphi_i(x))\,dx - E\varphi_i(0), \quad i = 0,\ldots,N\,. \tag{181}$$

We now turn to actual computations with P1 finite elements. The focus is on how the linear system and the element matrices and vectors are modified by the condition $u'(0) = C$.

14.2 Direct computation of the global linear system

Consider first the approach where Dirichlet conditions are incorporated by a $B(x)$ function and leaving out the known degrees of freedom from the linear system. The relevant formula for the linear system is given by (180). There are two differences compared to the extensively computed case where $u(0) = 0$. Because we do not have a Dirichlet condition at the left boundary, we need to extend the linear system (167) with an equation corresponding to $i = 0$. According to Section 13.2, this consists of including $A_{0,0} = 1/h$, $A_{0,1} = -1/h$, and $b_0 = h$. Second, we need to include the extra term $-E\varphi_i(0)$ on the right-hand side. Since all $\varphi_i(0) = 0$ for $i = 1,\ldots,N$, this term reduces to $-E\varphi_0(0) = -E$ and affects only the first equation ($i = 0$). We simply add $-E$ to b_0 such that $b_0 = h - E$.

Next we consider the technique where we modify the linear system to incorporate Dirichlet conditions. The only difference from the case in Section 13.2 is simply the extra term $-E$ in the b_0 entry so we can just add this value and continue with modifying the last equation to incorporate $c_N = D$.

14.3 Elementwise computations

In the case we compute with one element at a time, we need to see how the $u'(0) = C$ condition affects the element matrix and vector. As above for the case of forming a global system directly, the extra term $-E\varphi_i(0)$ in the variational formulation only affects the element vector in the first element. On the reference cell, $-E\varphi_i(0)$ is transformed to $-E\tilde{\varphi}_r(-1)$, where r counts local degrees of freedom. Only $\tilde{\varphi}_0(-1) \neq 0$ so we are left with the contribution $-E\tilde{\varphi}_0(-1) = -E$ to $\tilde{b}_0^{(0)}$:

$$\tilde{A}^{(0)} = A = \frac{1}{h}\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(0)} = \begin{pmatrix} h - E \\ h \end{pmatrix}. \tag{182}$$

No other element matrices or vectors are affected.

15 Implementation

It is tempting to create a program with symbolic calculations to perform all the steps in the computational machinery, both for automating the work and for documenting the complete algorithms. As we have seen, there are quite many details involved with finite element computations and incorporation of boundary conditions. An implementation will also act as a structured summary of all these details.

15.1 Global basis functions

A function similar to `least_squares` from Section 2.4 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, since the formulas must be derived by a Galerkin or least squares principle and depend on the differential equation at hand. 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 entry (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 cell 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   # assume symmetry
        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 = sum(c[i,0]*phi[0][i] for i in range(len(phi[0])))
        return u
```

It turns out that symbolic solution of differential equations, discretized by a Galerkin or least squares 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 WolframAlpha[18][19][20] (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 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 Section 2.10). Note that for a numerical integration, symbolic expressions must be converted to Python functions (using `lambdify`), and the expressions cannot contain other symbols than `x`. The real `solve` routine in the `varform1D.py` file 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 or least squares 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]])
```

[18] http://wolframalpha.com
[19] http://wolframalpha.com
[20] http://wolframalpha.com

```
        if boundary_rhs is not None:
            I += boundary_rhs(phi, i)
        b[i,0] = I
    c = A.LUsolve(b)
    u = sum(c[i,0]*phi[0][i] for i in range(len(phi[0])))
    return u
```

15.2 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 arbitrary choice from this family is $B(x) = 1 - x^3$. The unknown function is then written as a sum of a known (B) and an unknown ($\bar{u}$) function:

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

Let us use the Galerkin method to derive the variational formulation. Multiplying the differential equation by v and integrate by parts yield

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

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

$$\int_0^1 \bar{u}'v'dx = \int_0^1 (f - B')vdx \quad \forall v \in V\,. \tag{183}$$

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

$$\sum_{j=0}^{N}\left(\int_0^1 \varphi_i'\varphi_j'dx\right)c_j = \int_0^1 (f - B')\varphi_i dx, \quad i = 0,\ldots,N\,. \tag{184}$$

The application can be coded as follows in `sympy`:

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

# Compute basis functions and their derivatives
```

```
N = 3
phi = {0: [x**(i+1)*(1-x) for i in range(N+1)]}
phi[1] = [sm.diff(phi_i, x) for phi_i in 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,
              verbose=True, numint=False)
u = B + u_bar
print 'solution u:', sm.simplify(sm.expand(u))
```

The printout of `u` reads `-b*x**2/2 + b*x/2 - x + 1`. Note that expanding `u` and then simplifying is in the present case desirable to get a compact, final expression with `sympy`. A non-expanded `u` might be preferable in other cases - this depends on the problem in question.

The exact solution $u_{\mathrm{e}}(x)$ can be derived by the following `sympy` code:

```
# Solve -u''=f by integrating f twice
f1 = sm.integrate(f, x)
f2 = sm.integrate(f1, x)
# Add integration constants
C1, C2 = sm.symbols('C1 C2')
u_e = -f2 + C1*x + C2
# Find C1 and C2 from the boundary conditions u(0)=0, u(1)=1
s = sm.solve([u_e.subs(x,0) - 1, u_e.subs(x,1) - 0], [C1, C2])
# Form the exact solution
u_e = -f2 + s[C1]*x + s[C2]
print 'analytical solution:', u_e
print 'error:', sm.expand(u - u_e)
```

The last line prints `0`, which is not surprising when $u_{\mathrm{e}}(x)$ is a parabola and our approximate u contains polynomials up to degree 4. It suffices to have $N = 1$, i.e., polynomials of degree 2, to recover the exact solution.

We can play around with the code and test that with $f \sim x^p$, the solution is a polynomial of degree $p+2$, and $N = p+1$ guarantees that the approximate solution is exact.

Although the symbolic code is capable of integrating many choices of $f(x)$, the symbolic expressions for u quickly become lengthy and non-informative, so numerical integration in the code, and hence numerical answers, have the greatest application potential.

15.3 Finite elements

Implementation of the finite element algorithms for differential equations follows closely the algorithm for approximation of functions. The new additional ingredients are

1. other types of integrands (as implied by the variational formulation)
2. additional boundary terms in the variational formulation for Neumann boundary conditions
3. modification of element matrices and vectors due to Dirichlet boundary conditions

Point 1 and 2 can be taken care of by letting the user supply functions defining the integrands and boundary terms on the left- and right-hand side of the equation system:

```
integrand_lhs(phi, r, s, x)
boundary_lhs(phi, r, s, x)
integrand_rhs(phi, r, x)
boundary_rhs(phi, r, x)
```

Here, `phi` is a dictionary where `phi[q]` holds a list of the derivatives of order `q` of the basis functions at the an evaluation point; `r` and `s` are indices for the corresponding entries in the element matrix and vector, and `x` is the global coordinate value corresponding to the current evaluation point.

Given a mesh represented by `vertices`, `cells`, and `dof_map` as explained before, we can write a pseudo Python code to list all the steps in the computational algorithm for finite element solution of a differential equation.

```
<Declare global matrix and rhs: A, b>

for e in range(len(cells)):

    # Compute element matrix and vector
    n = len(dof_map[e])  # no of dofs in this element
    h = vertices[cells[e][1]] - vertices[cells[e][1]]
    <Declare element matrix and vector: A_e, b_e>

    # Integrate over the reference cell
    points, weights = <numerical integration rule>
    for X, w in zip(points, weights):
        phi = <basis functions and derivatives at X>
        detJ = h/2
        x = <affine mapping from X>
        for r in range(n):
            for s in range(n):
                A_e[r,s] += integrand_lhs(phi, r, s, x)*detJ*w
            b_e[r] += integrand_rhs(phi, r, x)*detJ*w

    # Add boundary terms
    for r in range(n):
        for s in range(n):
            A_e[r,s] += boundary_lhs(phi, r, s, x)*detJ*w
        b_e[r] += boundary_rhs(phi, r, x)*detJ*w

    # Incorporate essential boundary conditions
    for r in range(n):
        global_dof = dof_map[e][r]
        if global_dof in essbc_dofs:
```

```
            # dof r is subject to an essential condition
            value = essbc_docs[global_dof]
            # Symmetric modification
            b_e -= value*A_e[:,r]
            A_e[r,:] = 0
            A_e[:,r] = 0
            A_e[r,r] = 1
            b_e[r] = value

    # Assemble
    for r in range(n):
        for s in range(n):
            A[dof_map[e][r], dof_map[e][r]] += A_e[r,s]
        b[dof_map[e][r] += b_e[r]

<solve linear system>
```

16 Variational formulations in 2D and 3D

The major difference between deriving variational formulations in 2D and 3D compared to 1D is the rule for integrating by parts. A typical second-order term in a PDE may be written in dimension-independent notation as

$$\nabla^2 u \quad \text{or} \quad \nabla\cdot(a(\boldsymbol{x})\nabla u)\ .$$

The explicit forms in a 2D problem become

$$\nabla^2 u = \nabla\cdot\nabla u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$

and

$$\nabla\cdot(a(\boldsymbol{x})\nabla u) = \frac{\partial}{\partial x}\left(a(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(a(x,y)\frac{\partial u}{\partial y}\right)\ .$$

The general rule for integrating by parts is

$$-\int_\Omega \nabla\cdot(a(\boldsymbol{x})\nabla u)v\,\mathrm{d}x = \int_\Omega a(\boldsymbol{x})\nabla u\cdot\nabla v\,\mathrm{d}x - \int_{\partial\Omega} a\frac{\partial u}{\partial n}v\,\mathrm{d}s, \tag{185}$$

where $\partial\Omega$ is the boundary of Ω and $\partial u/\partial n = \boldsymbol{n}\cdot\nabla u$ is the derivative of u in the outward normal direction, $\boldsymbol{n}$ being an outward unit normal to $\partial\Omega$.

Note that (185) obviously applies to a constant a, which is what we have if the PDE has a Laplace term $a\nabla^2 u$.

Let us divide the boundary into two parts:

- $\partial\Omega_N$, where we have Neumann conditions $-a\frac{\partial u}{\partial n} = g$, and
- $\partial\Omega_D$, where we have Dirichlet conditions $u = u_0$.

The test functions v are required to vanish on $\partial\Omega_D$.

Example. Here is a quite general linear PDE arising in many problems:

$$\boldsymbol{v} \cdot \nabla u + \alpha u = \nabla \cdot (a \nabla u) + f, \quad \boldsymbol{x} \in \Omega, \tag{186}$$

$$u = u_0, \quad \boldsymbol{x} \in \partial\Omega_D, \tag{187}$$

$$-a\frac{\partial u}{\partial n} = g, \quad \boldsymbol{x} \in \partial\Omega_N\,. \tag{188}$$

The vector field $\boldsymbol{v}$ and the scalar functions a, α, f, u_0, and g may vary with the spatial coordinate $\boldsymbol{x}$ and must be known.

Such a second-order PDE needs exactly one boundary condition at each point of the boundary, so $\partial\Omega_N \cup \partial\Omega_D$ must be the complete boundary $\partial\Omega$.

The unknown function can be expanded as

$$u = u_0 + \sum_{j=0}^{N} c_j \varphi_j\,.$$

The variational formula is obtained from Galerkin's method, which technically implies multiplying the PDE by a test function v and integrating over Ω:

$$\int_\Omega (\boldsymbol{v} \cdot \nabla u + \alpha u) v \,\mathrm{d}x = \int_\Omega \nabla \cdot (a \nabla u)\,\mathrm{d}x + fv\,\mathrm{d}x\,.$$

The second-order term is integrated by parts:

$$\int_\Omega \nabla \cdot (a \nabla u)\,\mathrm{d}x = -\int_\Omega a\nabla u \cdot \nabla v\,\mathrm{d}x + \int_{\partial\Omega} a\frac{\partial u}{\partial n} v\,\mathrm{d}s,$$

resulting in

$$\int_\Omega (\boldsymbol{v} \cdot \nabla u + \alpha u) v \,\mathrm{d}x = -\int_\Omega a\nabla u \cdot \nabla v\,\mathrm{d}x + \int_{\partial\Omega} a\frac{\partial u}{\partial n} v\,\mathrm{d}s + \int_\Omega fv\,\mathrm{d}x\,.$$

The boundary term can be developed further by noticing that $v \neq 0$ only on $\partial\Omega_N$,

$$\int_{\partial\Omega} a\frac{\partial u}{\partial n} v\,\mathrm{d}s = \int_{\partial\Omega_N} a\frac{\partial u}{\partial n} v\,\mathrm{d}s,$$

and that on $\partial\Omega_N$, we have the condition $a\frac{\partial u}{\partial n} = -g$, so the term becomes

$$-\int_{\partial\Omega_N} gv\,\mathrm{d}s\,.$$

The variational form is then

$$\int_\Omega (\boldsymbol{v} \cdot \nabla u + \alpha u) v \,\mathrm{d}x = -\int_\Omega a\nabla u \cdot \nabla v\,\mathrm{d}x - \int_{\partial\Omega} gv\,\mathrm{d}s + \int_\Omega fv\,\mathrm{d}x\,.$$

Instead of using the integral signs we may use the inner product notation $(\cdot,\cdot)$:

$$(\boldsymbol{v}\cdot\nabla u, v) + (\alpha u, v) = -(a\nabla u, \nabla v) - (g, v)_N + (f, v)\,.$$

The subscript $_N$ in $(g, v)_N$ is a notation for a line or surface integral over $\partial\Omega_N$.

Inserting the u expansion results in

$$\sum_{j=0}^{N}((\boldsymbol{v}\cdot\nabla\varphi_j, \varphi_i) + (\alpha\varphi_j, \varphi_i) + (a\nabla\varphi_j, \nabla\varphi_i))c_j = (g, \varphi_i)_N + (f, \varphi_i) - (\boldsymbol{v}\cdot\nabla u_0, \varphi_i) + (\alpha u_0, \varphi_i) + (a\nabla u_0, \nabla\varphi_i)\,.$$

This is a linear system with matrix entries

$$A_{i,j} = (\boldsymbol{v}\cdot\nabla\varphi_j, \varphi_i) + (\alpha\varphi_j, \varphi_i) + (a\nabla\varphi_j, \nabla\varphi_i)$$

and right-hand side entries

$$b_i = (g, \varphi_i)_N + (f, \varphi_i) - (\boldsymbol{v}\cdot\nabla u_0, \varphi_i) + (\alpha u_0, \varphi_i) + (a\nabla u_0, \nabla\varphi_i),$$

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

In the finite element method, we usually express u_0 in terms of basis functions and restrict i and j to run over the degrees of freedom that are not prescribed as Dirichlet conditions. However, we can also keep all the c_j, $j = 0, \ldots, N$, as unknowns drop the u_0 in the expansion for u, and incorporate all the known c_j values in the linear system. This has been explained in detail in the 1D case.

16.1 Transformation to a reference cell in 2D and 3D

We consider an integral of the type

$$\int_{\Omega^{(e)}} a(\boldsymbol{x})\nabla\varphi_i\cdot\nabla\varphi_j\,\mathrm{d}x \tag{189}$$

in the physical domain. Suppose we want to calculate this integral over a reference cell, denoted by $\tilde{\Omega}^r$, in a coordinate system with coordinates $\boldsymbol{X} = (X_0, X_1)$ (2D) or $\boldsymbol{X} = (X_0, X_1, X_2)$ (3D). The mapping between a point $\boldsymbol{X}$ in the reference coordinate system and the corresponding point $\boldsymbol{x}$ in the physical coordinate system is given by a vector relation $\boldsymbol{x}(\boldsymbol{X})$. The corresponding Jacobian, J, of this mapping has entries

$$J_{i,j} = \frac{\partial x_j}{\partial X_i}\,.$$

The change of variables requires $\mathrm{d}x$ to be replaced by $\det J\,\mathrm{d}X$. The derivatives in the ∇ operator in the variational form are with respect to $\boldsymbol{x}$, which we may denote by $\nabla_{\boldsymbol{x}}$. The $\varphi_i(\boldsymbol{x})$ functions in the integral are replaced by local basis functions $\tilde{\varphi}_r(\boldsymbol{X})$ so the integral features $\nabla_{\boldsymbol{x}}\tilde{\varphi}_r(\boldsymbol{X})$. We readily have $\nabla_{\boldsymbol{X}}\tilde{\varphi}_r(\boldsymbol{X})$

from formulas for the basis functions, but the desired quantity $\nabla_{\boldsymbol{x}}\tilde{\varphi}_r(\boldsymbol{X})$ requires some efforts to compute. All the details are now given.

Let $i = q(e,r)$ and consider two space dimensions. By the chain rule,

$$\frac{\partial\tilde{\varphi}_r}{\partial X} = \frac{\partial\varphi_i}{\partial X} = \frac{\partial\varphi_i}{\partial x}\frac{\partial x}{\partial X} + \frac{\partial\varphi_i}{\partial y}\frac{\partial y}{\partial X},$$

and

$$\frac{\partial\tilde{\varphi}_r}{\partial Y} = \frac{\partial\varphi_i}{\partial Y} = \frac{\partial\varphi_i}{\partial x}\frac{\partial x}{\partial Y} + \frac{\partial\varphi_i}{\partial y}\frac{\partial y}{\partial Y}\,.$$

We can write this as a vector equation

$$\begin{bmatrix} \frac{\partial\tilde{\varphi}_r}{\partial X} \\ \frac{\partial\tilde{\varphi}_r}{\partial Y} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial X} & \frac{\partial y}{\partial X} \\ \frac{\partial x}{\partial Y} & \frac{\partial y}{\partial Y} \end{bmatrix} \begin{bmatrix} \frac{\partial\varphi_i}{\partial x} \\ \frac{\partial\varphi_i}{\partial y} \end{bmatrix}$$

Identifying

$$\nabla_{\boldsymbol{X}}\tilde{\varphi}_r = \begin{bmatrix} \frac{\partial\tilde{\varphi}_r}{\partial X} \\ \frac{\partial\tilde{\varphi}_r}{\partial Y} \end{bmatrix}, \quad J = \begin{bmatrix} \frac{\partial x}{\partial X} & \frac{\partial y}{\partial X} \\ \frac{\partial x}{\partial Y} & \frac{\partial y}{\partial Y} \end{bmatrix}, \quad \nabla_{\boldsymbol{x}}\varphi_r = \begin{bmatrix} \frac{\partial\varphi_i}{\partial x} \\ \frac{\partial\varphi_i}{\partial y} \end{bmatrix},$$

we have the relation

$$\nabla_{\boldsymbol{X}}\tilde{\varphi}_r = J\cdot\nabla_{\boldsymbol{x}}\varphi_i,$$

which we can solve with respect to $\nabla_{\boldsymbol{x}}\varphi_i$:

$$\nabla_{\boldsymbol{x}}\varphi_i = J^{-1}\cdot\nabla_{\boldsymbol{X}}\tilde{\varphi}_r\,. \tag{190}$$

This means that we have the following transformation of the integral in the physical domain to its counterpart over the reference cell:

$$\int_\Omega^{(e)} a(\boldsymbol{x})\nabla_{\boldsymbol{x}}\varphi_i\cdot\nabla_{\boldsymbol{x}}\varphi_j\,\mathrm{d}x \int_{\tilde{\Omega}^r} a(\boldsymbol{x}(\boldsymbol{X}))(J^{-1}\cdot\nabla_{\boldsymbol{X}}\tilde{\varphi}_r)\cdot(J^{-1}\cdot\nabla\tilde{\varphi}_s)\det J\,\mathrm{d}X \tag{191}$$

16.2 Numerical integration

Integrals are normally computed by numerical integration rules. For multi-dimensional cells, various families of rules exist. All of them are similar to what is shown in 1D: $\int f dx \approx \sum_j w_i f(\boldsymbol{x}_j)$, where w_j are weights and $\boldsymbol{x}_j$ are corresponding points.

16.3 Convenient formulas for P1 elements in 2D

We shall now provide some formulas for piecewise linear φ_i functions and their integrals *in the physical coordinate system*. These formulas make it convenient to compute with P1 elements without the need to work in the reference coordinate system and deal with mappings and Jacobians. A lot of computational and algorithmic details are hidden by this approach.

Let $\Omega^{(e)}$ be cell number e, and let the three vertices have global vertex numbers I, j, and K. The corresponding coordinates are (x_I, y_I), (x_J, y_J), and (x_K, y_K). The basis function φ_I over $\Omega^{(e)}$ have the explicit formula

$$\varphi_I(x,y) = \frac{1}{2}\Delta\left(\alpha_I + \beta_I x + \gamma_I y\right), \tag{192}$$

where

$$\alpha_I = x_J y_K - x_K y_J, \tag{193}$$
$$\beta_I = y_J - y_K, \tag{194}$$
$$\gamma_I = x_K - x_J, \tag{195}$$
$$2\Delta = \det \begin{pmatrix} 1 & x_I & y_I \\ 1 & x_J & y_J \\ 1 & x_K & y_K \end{pmatrix}. \tag{196}$$

The quantity Δ is the area of the cell.

The following formula is often convenient when computing element matrices and vectors:

$$\int_{\Omega^{(e)}} \varphi_I^p \varphi_J^q \varphi_K^r dxdy = \frac{p!q!r!}{(p+q+r+2)!} 2\Delta\,. \tag{197}$$

(Note that the q in this formula is not to be mixed with the $q(e,r)$ mapping of degrees of freedom.)

As an example, the element matrix entry $\int_{\Omega^{(e)}} \varphi_I \varphi_J \,\mathrm{d}x$ can be computed by setting $p = q = 1$ and $r = 0$, when $I \neq J$, yielding $\Delta/12$, and $p = 2$ and $q = r = 0$, when $I = J$, resulting in $\Delta/6$. We collect these numbers in a local element matrix:

$$\frac{\Delta}{12}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

The common element matrix entry $\int_{\Omega^{(e)}} \nabla\varphi_I \cdot \nabla\varphi_J \,\mathrm{d}x$, arising from a Laplace term ∇^u, can also easily be computed by the formulas above. We have

$$\nabla\varphi_I \cdot \nabla\varphi_J = \frac{\Delta^2}{4}(\beta_I\beta_J + \gamma_I\gamma_J) = \text{const},$$

so that the element matrix entry becomes $\frac{1}{4}(\Delta^3\beta_I\beta_J + \gamma_I\gamma_J)$.

From an implementational point of view, one will work with local vertex numbers $r = 1, 2, 3$, parameterize the coefficients in the basis functions by r, and look up vertex coordinates through $q(e,r)$.

17 Summary

- When approximating f by $u = \sum_j c_j \varphi_j$, the least squares method and the Galerkin/projection method give the same result. The interpolation/collocation method is simpler and yields different (mostly inferior) results.
- Fourier series expansion can be viewed as a least squares or Galerkin approximation procedure with sine and cosine functions.
- Basis functions should optimally be orthogonal or almost orthogonal, because this gives little round-off errors when solving the linear system, and the coefficient matrix becomes diagonal or sparse.
- Finite element basis functions are *piecewise* polynomials, normally with discontinuous derivatives at the cell boundaries. The basis functions overlap very little, leading to stable numerics and sparse matrices.
- To use the finite element method for differential equations, we use the Galerkin method or the method of weighted residuals to arrive at a variational form. Technically, the differential equation is multiplied by a test function and integrated over the domain. Second-order derivatives are integrated by parts to allow for typical finite element basis functions that have discontinuous derivatives.
- The least squares method is not much used for finite element solution of differential equations of second order, because it then involves second-order derivatives which cause trouble for basis functions with discontinuous derivatives.
- We have worked with two common finite element terminologies and associated data structures (both are much used, especially the first one, while the other is more general):
  1. *elements*, *nodes*, and *mapping between local and global node numbers*
  2. an extended element concept consisting of *cell*, *vertices*, *degrees of freedom*, *local basis functions*, *geometry mapping*, and *mapping between local and global degrees of freedom*
- The meaning of the word "element" is multi-fold: the geometry of a finite element (also known as a cell), the geometry and its basis functions, or all information listed under point 2 above.
- One normally computes integrals in the finite element method element by element (cell by cell), either in a local reference coordinate system or directly in the physical domain.
- The advantage of working in the reference coordinate system is that the mathematical expressions for the basis functions depend on the element type only, not the geometry of that element in the physical domain. The

disadvantage is that a mapping must be used, and derivatives must be transformed from reference to physical coordinates.

- Element contributions to the global linear system are collected in an element matrix and vector, which must be assembled into the global system using the degree of freedom mapping (`dof_map`) or the node numbering mapping (`elements`), depending on which terminology that is used.
- Dirichlet conditions, involving prescribed values of u at the boundary, are implemented either via a boundary function that take on the right Dirichlet values, while the basis functions vanish at such boundaries. In the finite element method, one has a general expression for the boundary function, but one can also incorporate Dirichlet conditions in the element matrix and vector or in the global matrix system.
- Neumann conditions, involving prescribed values of the derivative (or flux) of u, are incorporated in boundary terms arising from integrating terms with second-order derivatives by part. Forgetting to account for the boundary terms implies the condition $\partial u/\partial n = 0$ at parts of the boundary where no Dirichlet condition is set.

18 Exercises

Exercise 17: Compute the deflection of a cable with sine functions

A hanging cable of length L with significant tension has a downward deflection $w(x)$ governed by

Solve

$$Tw''(x) = \ell(x),$$

where T is the tension in the cable and $\ell(x)$ the load per unit length. The cable is fixed at $x = 0$ and $x = L$ so the boundary conditions become $T(0) = T(L) = 0$. We assume a constant load $\ell(x) = \text{const}$.

The solution is expected to be symmetric around $x = L/2$. Formulating the problem for $x \in [0, L/2]$ and then scaling it, results in the scaled problem for the dimensionless vertical deflection u:

$$u'' = 1, \quad x \in (0,1), \quad u(0) = 0,\ u'(1) = 0$$

Introduce the function space spanned by $\varphi_i = \sin((i+1)\pi x/2)$, $i = 1, \ldots, N$. Use a Galerkin and a least squares method to find the coefficients c_j in $u(x) = \sum_j c_j\varphi_j$. Find how fast the coefficients decrease in magnitude by looking at c_j/c_{j-1}. Find the error in the maximum deflection at $x = 1$ when only one basis function is used ($N = 0$).

What happens if we choose basis functions $\varphi_i = \sin((i+1)\pi x)$? Filename: `cable_sin`.

Exercise 18: Check integration by parts

Consider the Galerkin method for the problem involving u in Exercise 17. Show that the formulas for c_j are independent of whether we perform integration by parts or not. Filename: `cable_integr_by_parts`.

Exercise 19: Compute the deflection of a cable with 2 P1 elements

Solve the problem for u in Exercise 17 using two P1 linear elements. Filename: `cable_2P1`.

Exercise 20: Compute the deflection of a cable with 1 P2 element

Solve the problem for u in Exercise 17 using one P2 element with quadratic basis functions. Filename: `cable_1P2`.

Exercise 21: Compute the deflection of a cable with a step load

We consider the deflection of a tension cable as described in Exercise 17. Now the load is

$$\ell(x) = \begin{cases} \ell_1, & x < L/2, \\ \ell_2, & x \geq L/2 \end{cases} \qquad x \in [0, L]\,.$$

This load is not symmetric with respect to the midpoint $x = L/2$ so the solution loses its symmetry and we must solve the scaled problem

$$u'' = \begin{cases} 1, & x < 1/2, \\ 0, & x \geq 1/2 \end{cases} \qquad x \in (0,1), \quad u(0) = 0,\ u(1) = 0\,.$$

a) Use $\varphi_i = \sin((i+1)\pi x)$, $i = 0, \ldots, N$ and the Galerkin method without integration by parts. Derive a formula for c_j in the solution expansion $u = \sum_j c_j \varphi_j$. Plot how fast the coefficients c_j tend to zero (on a log scale).

b) Solve the problem with P1 finite elements. Plot the solution for $n_e = 2, 4, 8$ elements.

Filename: `cable_discont_load`.

Exercise 22: Show equivalence between linear systems

Incorporation of Dirichlet conditions can either be done by introducing an expansion $u(x) = U_0\varphi_0 + U_N\varphi_N + \sum_{j=1}^{N-1} c_j\varphi_j$ and considering $c_1, \ldots, c_{N-1}$ as unknowns, *or* one can assemble the matrix system with $u(x) = \sum_{j=0}^{N} c_j\varphi_j$ and

afterwards replace the rows corresponding to known c_j values by the boundary conditions. The purpose of this exercise is to show the equivalence of these two approaches.

Consider the system (167) modified for the boundary value $u(L) = D$, as explained in Section 13.1, and the system (175), where all $c_0, \ldots, c_N$ are involved. Show that eliminating c_1 and c_N from (175) results in the other system (167).

Exercise 23: Compute with a non-uniform mesh

Derive the linear system for the problem $-u'' = 2$ on $[0, 1]$, with $u(0) = 0$ and $u(1) = 1$, using P1 elements and a *non-uniform* mesh. The vertices have coordinates $x_0 = 0 < x_1 < \cdots < x_N = 1$, and the length of cell number e is $h_e = x_{e+1} - x_e$.

It is of interest to compare the discrete equations for the finite element method in a non-uniform mesh with the corresponding discrete equations arising from a finite difference method. Repeat the reasoning for the finite difference formula $u''(x_i) \approx [D_x D_x u]_i$ and use it to find a natural discretization of $u''(x_i)$ on a non-uniform mesh. Filename: `nonuniform_P1.pdf`.

Exercise 24: Solve a 1D finite element problem by hand

The following scaled 1D problem is a very simple, yet relevant, model for convective transport in fluids:

$$u' = \epsilon u'', \quad u(0) = 0,\ u(1) = 1,\ x \in [0, 1]\,. \tag{198}$$

a) Find the analytical solution to this problem. (Introduce $w = u'$, solve the first-order differential equation for $w(x)$, and integrate once more.)

b) Derive the variational form of this problem.

c) Introduce a finite element mesh with uniform partitioning. Use P1 elements and compute the element matrix and vector for a general element.

d) Incorporate the boundary conditions and assemble the element contributions.

e) Identify the resulting linear system as a finite difference discretization of the differential equation using

$$[D_{2x} u = \epsilon D_x D_x u]_i\,.$$

f) Compute the numerical solution and plot it together with the exact solution for a mesh with 20 elements and $\epsilon = 0.1, 0.01$.

Filename: `convdiff1D_P1`.

Exercise 25: Compare finite elements and differences for a radially symmetric Poisson equation

We consider the Poisson problem in a disk with radius R with Dirichlet conditions at the boundary. Given that the solution is radially symmetric and hence dependent only on the radial coordinate ($r = \sqrt{x^2 + y^2}$), we can reduce the problem to a 1D Poisson equation

$$-\frac{1}{r}\frac{d}{dr}\left(r\frac{du}{dr}\right) = f(r), \quad r \in (0, R),\ u'(0) = 0,\ u(R) = U_R\,. \tag{199}$$

a) Use a uniform mesh partition with P1 elements and show what the resulting set of equations becomes. Integrate the matrix entries exact by hand, but use a Trapezoidal rule to integrate the f term.

b) Explain that a natural finite difference method applied to (199) gives

$$\frac{1}{r_i}\frac{1}{h^2}\left(r_{i+\frac{1}{2}}(u_{i+1} - u_i) - r_{i-\frac{1}{2}}(u_i - u_{i-1})\right) = f_i, \quad i = rh\,.$$

For $i = 0$ the factor $1/r_i$ seemingly becomes problematic. One must always have $u'(0) = 0$, because of the radial symmetry, which implies $u_{-1} = u_1$, if we allow introduction of a fictitious value u_{-1}. Using this u_{-1} in the difference equation for $i = 0$ gives

$$\frac{1}{r_0}\frac{1}{h^2}\left(r_{\frac{1}{2}}(u_1 - u_0) - r_{-\frac{1}{2}}(u_0 - u_1)\right) = \frac{1}{r_0}\frac{1}{2h^2}\left((r_0 + r_1)(u_1 - u_0) - (r_{-1} + r_0)(u_0 - u_1)\right) \approx 2(u_1 - u_0),$$

if we use $r_{-1} + r_1 \approx 2r_0$.

Set up the complete set of equations for the finite difference method and compare to the finite element method in case a Trapezoidal rule is used to integrate the f term in the latter method.

Filename: `radial_Poisson1D_P1.pdf`.

Exercise 26: Compute with variable coefficients and P1 elements by hand

Consider the problem

$$-\frac{d}{dx}\left(a(x)\frac{du}{dx}\right) + \gamma u = f(x), \quad x \in \Omega = [0, L], \quad u(0) = \alpha,\ u'(L) = \beta\,. \tag{200}$$

We choose $a(x) = 1 + x^2$. Then

$$u(x) = \alpha + \beta(1 + L^2)\tan^{-1}(x), \tag{201}$$

is an exact solution if $f(x) = \gamma u$.

Derive a variational formulation and compute general expressions for the element matrix and vector in an arbitrary element, using P1 elements and a uniform partitioning of $[0, L]$. The right-hand side integral is challenging and can be computed by a numerical integration rule. The Trapezoidal rule (102) gives particularly simple expressions. Filename: `atan1D_P1.pdf`.

Exercise 27: Solve a 2D Poisson equation using polynomials and sines

The classical problem of applying a torque to the ends of a rod can be modeled by a Poisson equation defined in the cross section Ω:

$$-\nabla^2 u = 2, \quad (x, y) \in \Omega,$$

with $u = 0$ on $\partial\Omega$. Exactly the same problem arises for the deflection of a membrane with shape Ω under a constant load.

For a circular cross section one can readily find an analytical solution. For a rectangular cross section the analytical approach ends up with a sine series. The idea in this exercise is to use a single basis function to obtain an approximate answer.

We assume for simplicity that the cross section is the unit square: $\Omega = [0, 1] \times [0, 1]$.

a) We consider the basis $\varphi_{p,q}(x, y) = \sin((p + 1)\pi x)\sin(q\pi y)$, $p, q = 0, \ldots, n$. These basis functions fulfill the Dirichlet condition. Use a Galerkin method and $n = 0$.

b) The basis function involving sine functions are orthogonal. Use this property in the Galerkin method to derive the coefficients $c_{p,q}$ in a formula $u = \sum_p \sum_q c_{p,q}\varphi_{p,q}(x, y)$.

c) Another possible basis is $\varphi_i(x, y) = (x(1 - x)y(1 - y))^{i+1}$, $i = 0, \ldots, N$. Use the Galerkin method to compute the solution for $N = 0$. Which choice of a single basis function is best, $u \sim x(1 - x)y(1 - y)$ or $u \sim \sin(\pi x)\sin(\pi y)$? In order to answer the question, it is necessary to search the web or the literature for an accurate estimate of the maximum u value at $x = y = 1/2$.

Filename: `torsion_sin_xy.pdf`.

Index