Basic finite element methods

Hans Petter Langtangen[1,2]

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

Oct 29, 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.

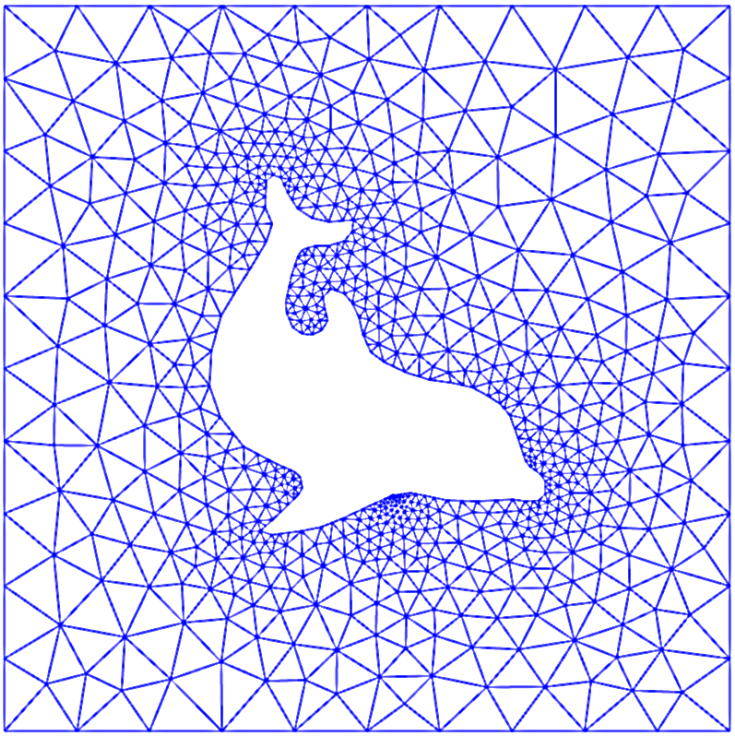

Figure 1: Domain for flow around a dolphin.

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) 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 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 a *Galerkin method* when the principle is used to solve differential equations, but it applies to the trivial equation $u = f$, i.e., approximation as well. 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*

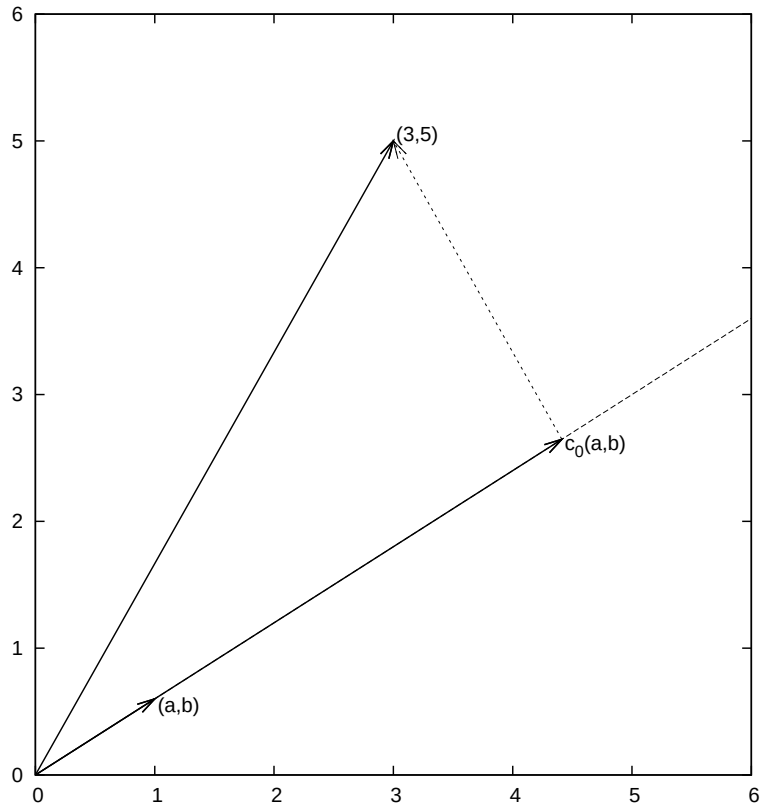

Figure 2: Approximation of a two-dimensional vector by a one-dimensional vector.

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

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

$$(\boldsymbol{u}, \boldsymbol{v}) = u_0 v_0 + u_1 v_1 \,. \tag{3}$$

Remark. 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.

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[1]:

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

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

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

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

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

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

The Galerkin 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 **??** (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

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

Therefore, instead of minimizing the square of the norm, we could demand that $\boldsymbol{e}$ is orthogonal to any vector in V. This is called Galerkin's method and stated mathematically as 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

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

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

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

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

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 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), has its name after the its inventor, the Russian mathematician Boris Galerkin[2], who used the approach to solve differential equations.

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 in Section 1.2. That is, we minimize the distance between u and f. However,

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

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 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 the 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 method are equivalent.

2.3 Example: linear approximation

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

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

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

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

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

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

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

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

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

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.

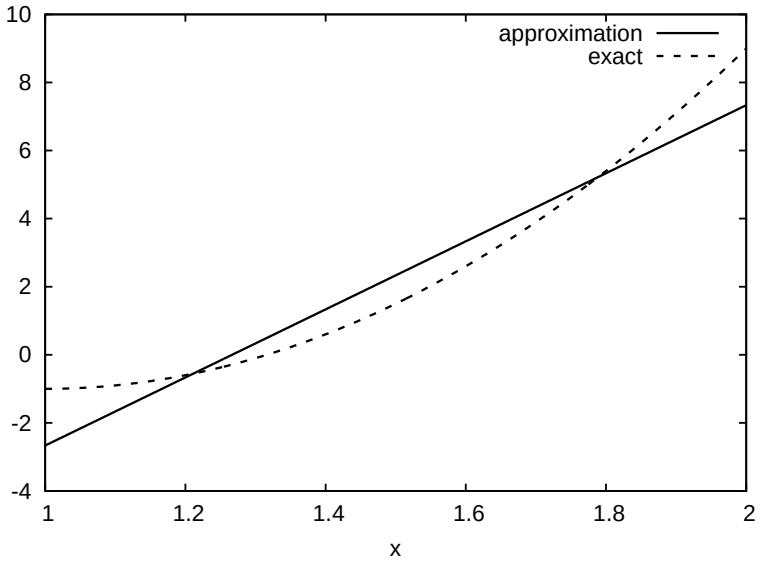

Figure 3: Best approximation of a parabola by a straight line.

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 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.

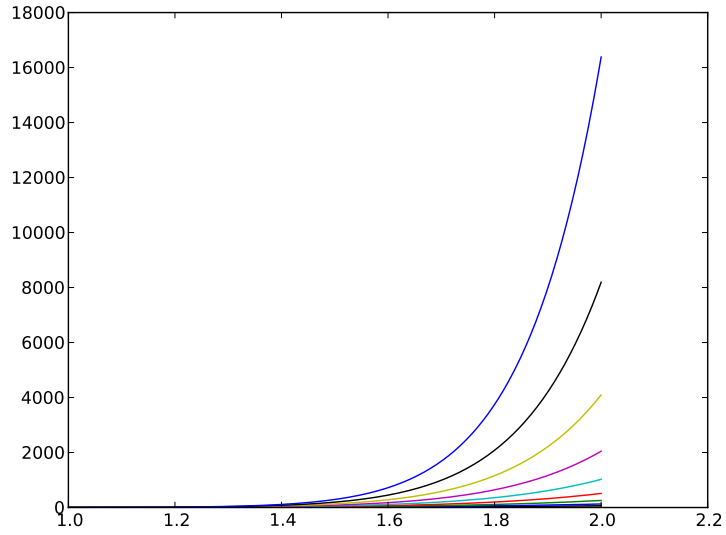

Figure 4: The 15 first basis functions x^i, $i = 0, \ldots, 14$.

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.

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
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).

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

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

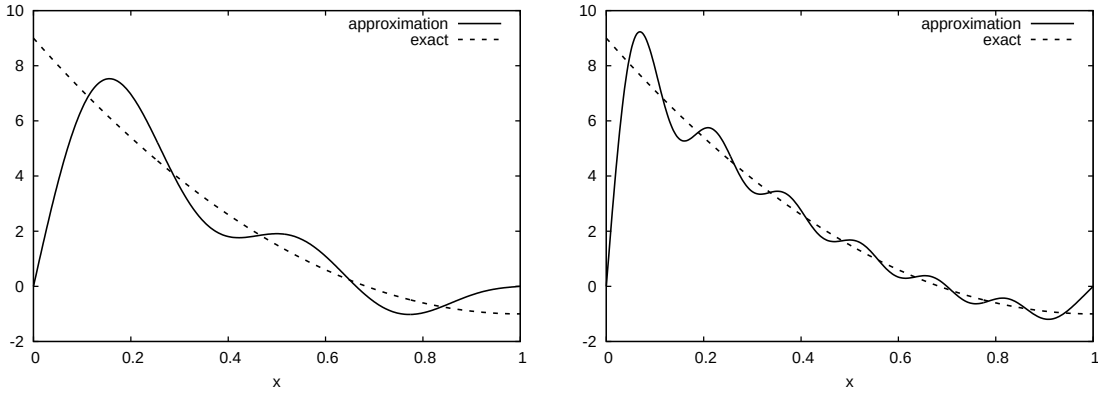

Figure 5: Best approximation of a parabola by a sum of 3 (left) and 11 (right) sine functions.

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.

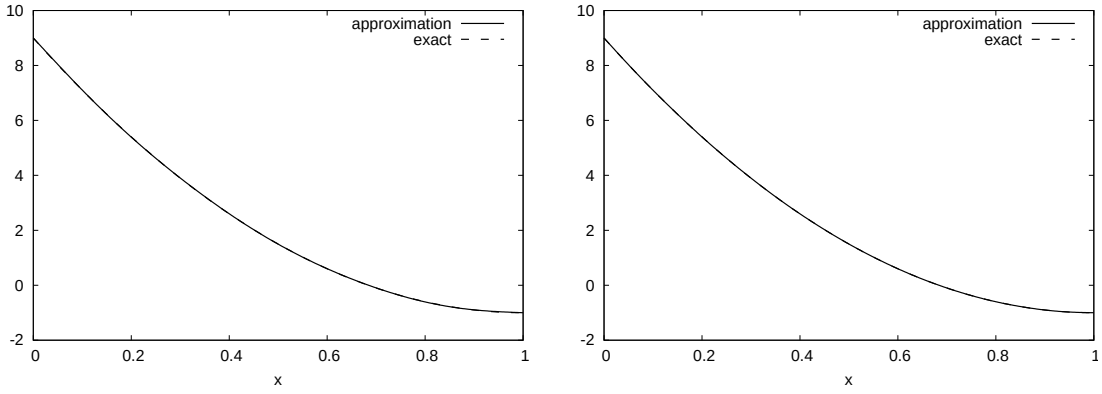

Figure 6: Best approximation of a parabola by a sum of 3 (left) and 11 (right) sine functions with a boundary term.

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[3] (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 or 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, $i = 0,\ldots,N$:

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

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

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).

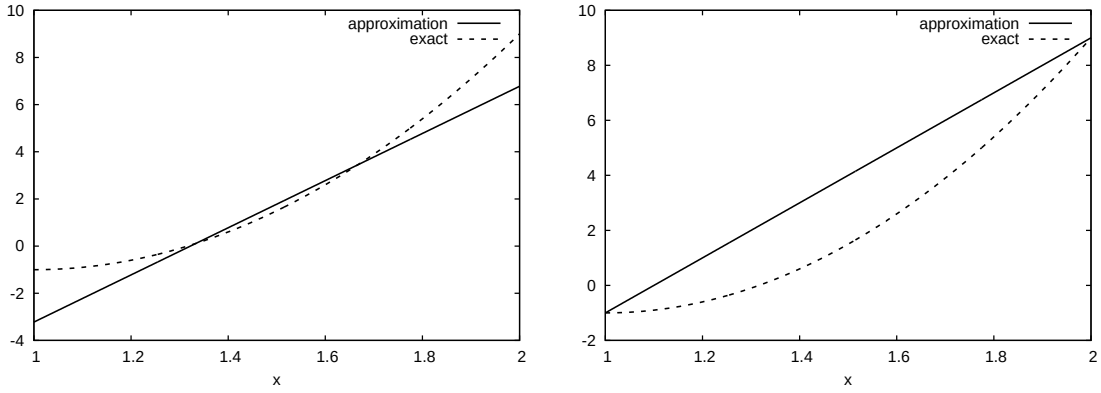

Figure 7: Approximation of a parabola by linear functions computed by two interpolation points: 4/3 and 5/3 (left) versus 1 and 2 (right).

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 a interpolation or collocation method a diagonal matrix implies that $\varphi_j(x_i) = 0$ if $i \neq j$. One set of basis functions $\varphi_i(x)$ with this property is the *Lagrange interpolating polynomials*, or just *Lagrange polynomials*. (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.

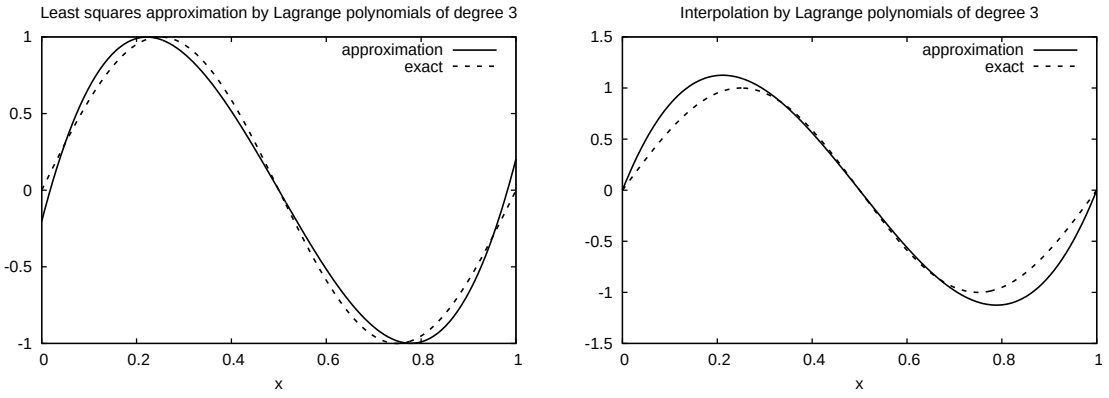

Figure 8: Approximation via least squares (left) and interpolation (right) of a sine function by Lagrange interpolating polynomials of degree 4.

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

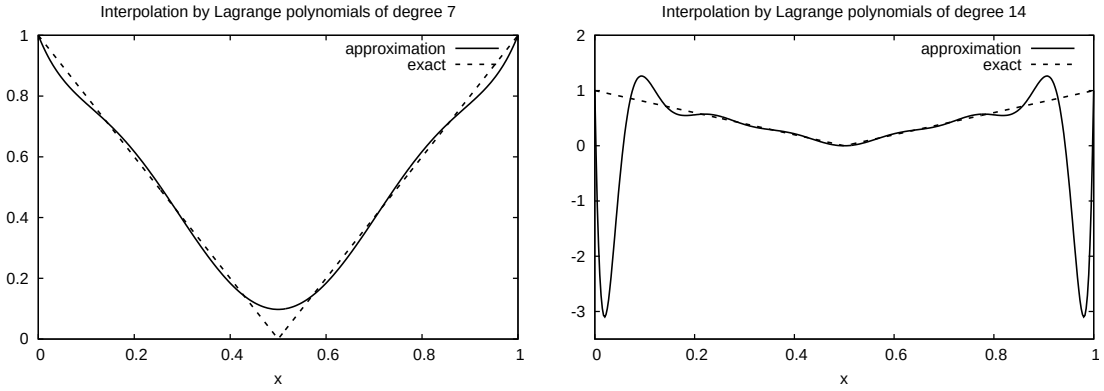

Figure 9: Interpolation of an absolute value function by Lagrange polynomials and uniformly distributed interpolation points: degree 7 (left) and 14 (right).

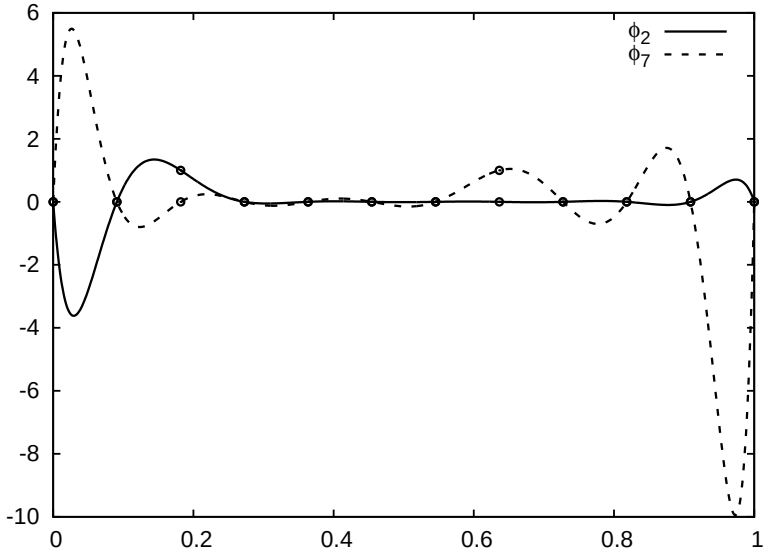

Figure 10: Illustration of the oscillatory behavior of two Lagrange polynomials for 12 uniformly spaced points (marked by circles).

an object of type `Integral`. We can test on this type and utilize the `mpmath` module in `sympy` to perform numerical integration of high precision. Here is the code:

```
def least_squares(f, phi, Omega):
    N = len(phi) - 1
    A = sm.zeros((N+1, N+1))
    b = sm.zeros((N+1, 1))
```

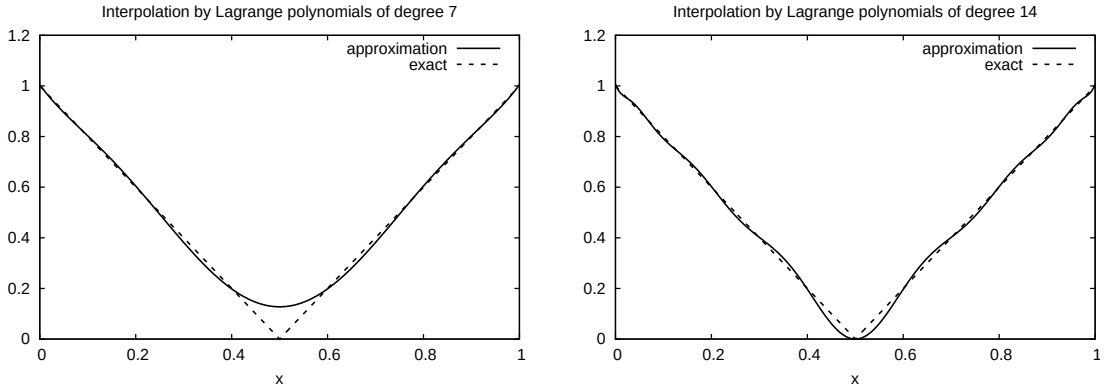

Figure 11: Interpolation of an absolute value function by Lagrange polynomials and Chebyshev nodes as interpolation points: degree 7 (left) and 14 (right).

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

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*.

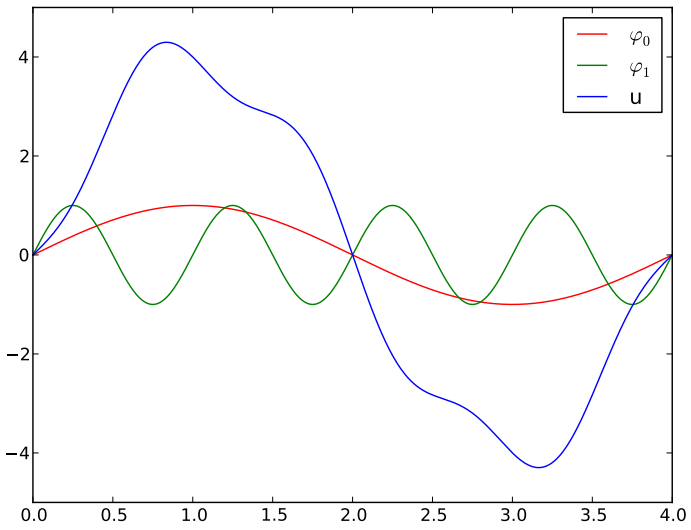

Figure 12: Approximation based on sine basis functions.

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 the family of finite element methods. The generalization is also compatible with the concepts used in the FEniCS[4] 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.

[4] `http://fenicsproject.org`

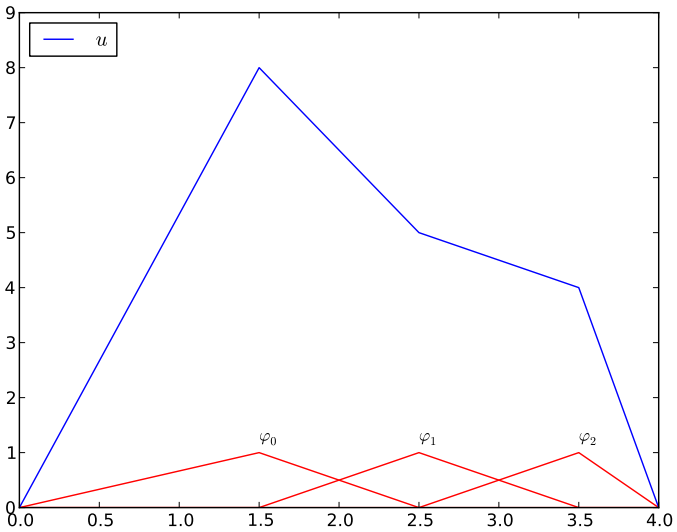

Figure 13: Approximation based on local piecewise linear (hat) functions.

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 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 $x \notin \Omega^{(e)}$.

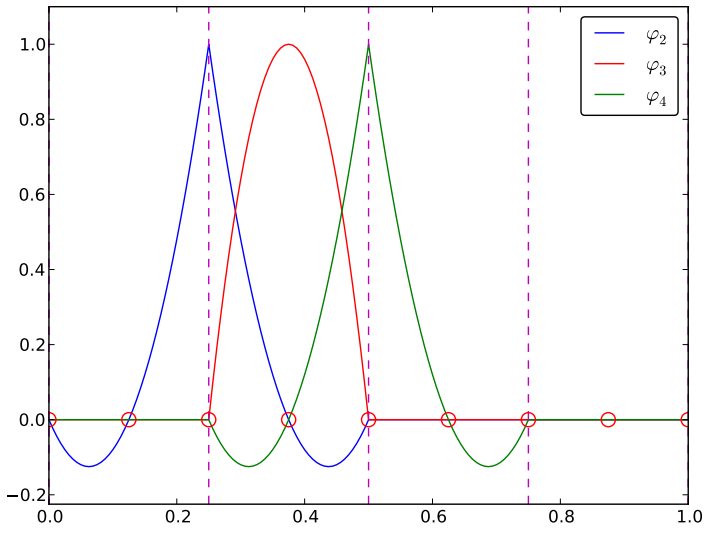

Figure 14: Illustration of the piecewise quadratic basis functions associated with nodes in element 1.

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 $\Omega^{(0)} = [0, 0.25]$.

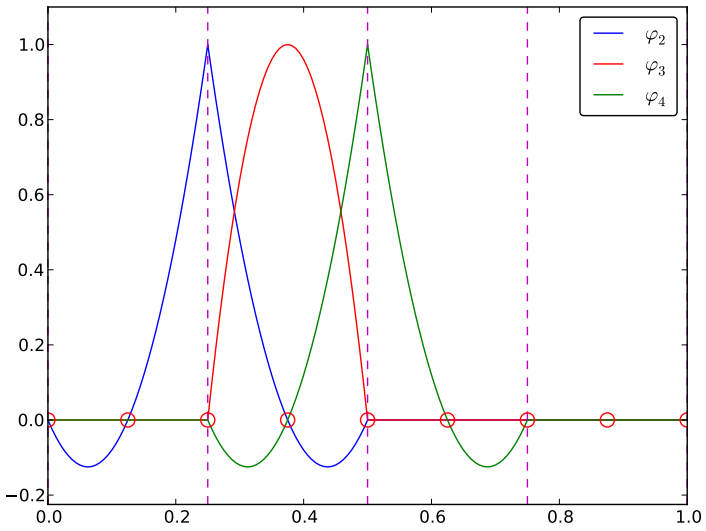

Figure 15: Illustration of the piecewise quadratic basis functions associated with nodes in element 1.

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 $\Omega^{(2)} = [0.5, 0.75]$.

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.

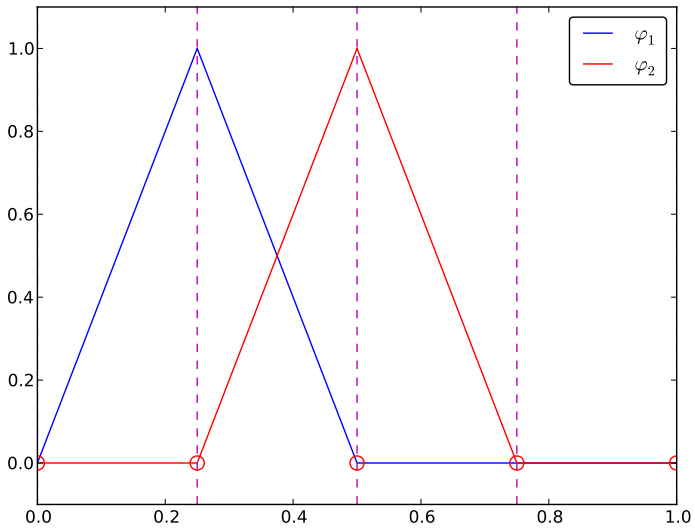

Figure 16: Illustration of the piecewise linear basis functions associated with nodes in element 1.

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

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

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 func-*

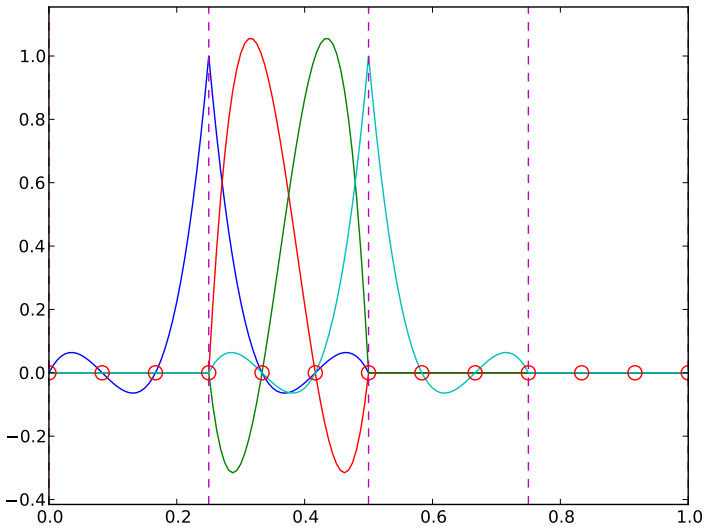

Figure 17: Illustration of the piecewise cubic basis functions associated with nodes in element 1.

tions. 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 "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.

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

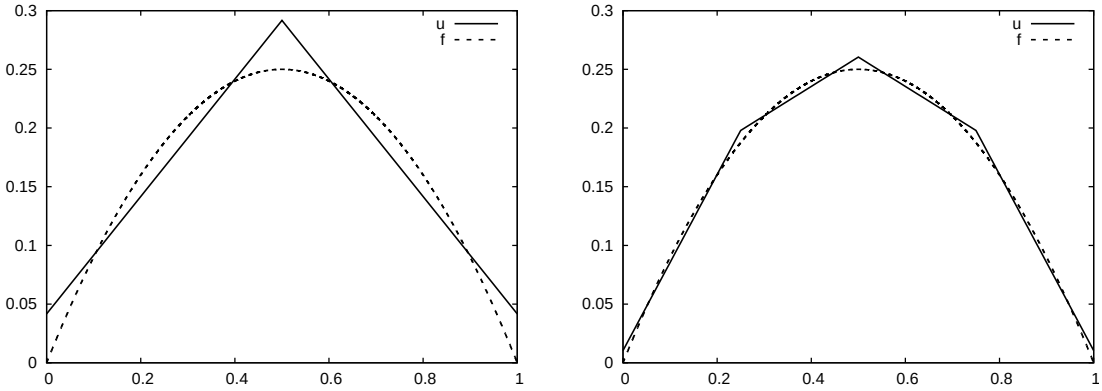

Figure 18: Least squares approximation using 2 (left) and 4 (right) P1 elements.

node numbers $i = q(e,r)$ and $j = q(e,s)$. With d nodes per element, all the nonzero elements in $A_{i,j}^{(e)}$ 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

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

for the element matrix. For the case $d = 2$ we have

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

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

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

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:

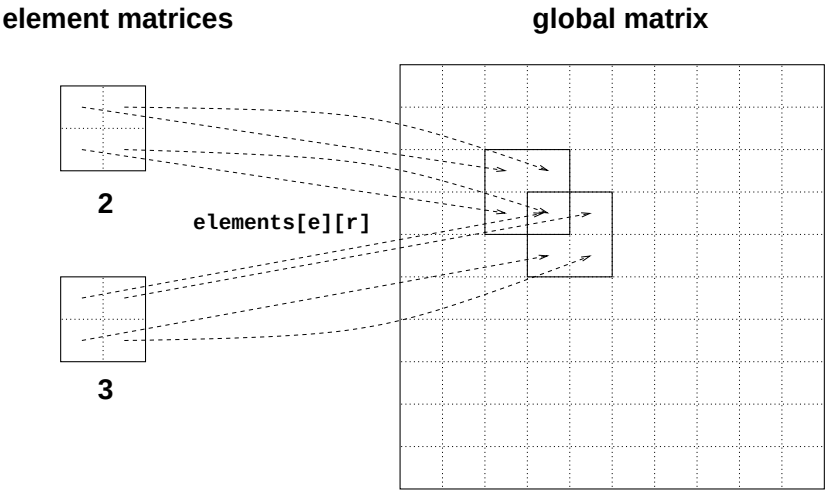

Figure 19: Illustration of matrix assembly.

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

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:

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

Successful numerical integration requires that the symbolic integrand is converted to a plain Python function (`integrand`) and that the element length `h` is 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)
    zeros = sm.zeros if symbolic else np.zeros
    A = zeros((n_n, n_n))
    b = zeros((n_n, 1))
    for e in range(n_e):
        Omega_e = [nodes[elements[e][0]], nodes[elements[e][-1]]]

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

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

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

Given the coefficient matrix `A` and the right-hand side `b`, we can compute the 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 or 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):

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

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

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.

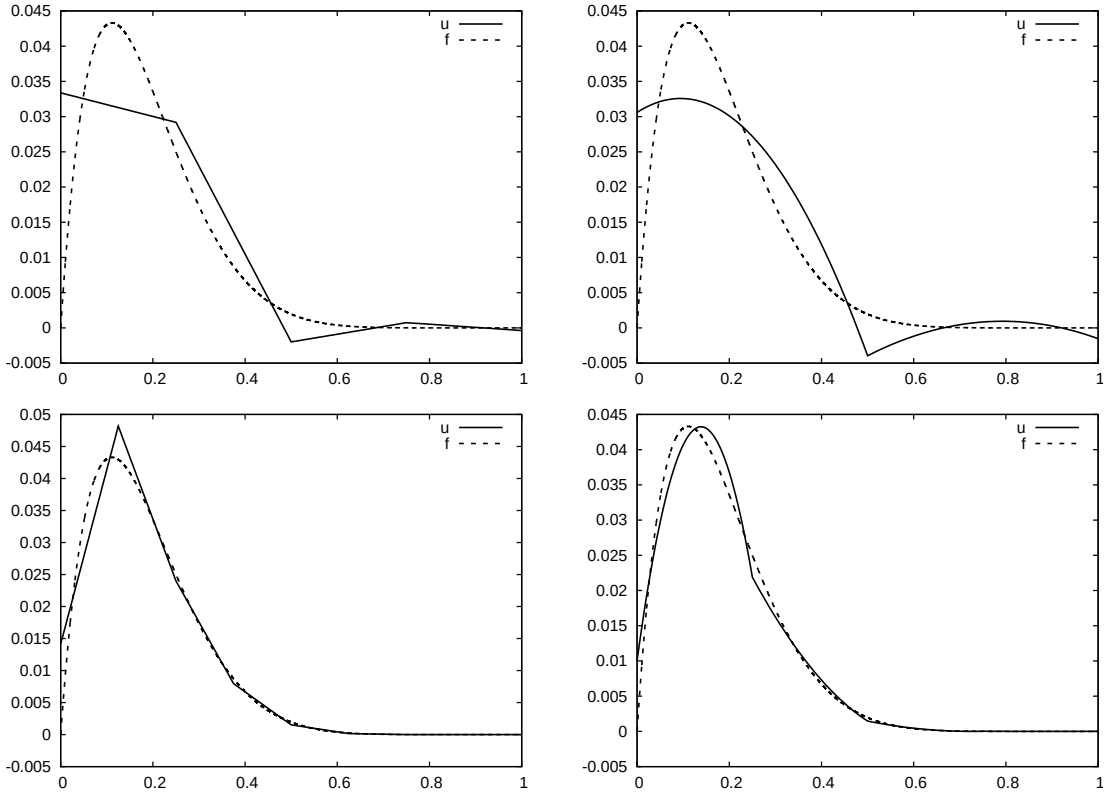

Figure 20: Comparison of the finite element approximations: 4 P1 elements with 5 nodes (upper left), 2 P2 elements with 5 nodes (upper right), 8 P1 elements with 9 nodes (lower left), and 4 P2 elements with 9 nodes (lower right).

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 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.

5.1 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.

5.2 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

$$u(x) = \sum_r c_r \tilde{\varphi}_r(X), \tag{84}$$

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.

5.3 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`.

5.4 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 $X = 1$

The four basis functions can be written in a general form

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

with four coefficients C_{ij}, $j = 0, 1, 2, 3$, 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_{ij} 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:

$$\tilde{\varphi}_0(X) = \tag{85}$$
$$\tilde{\varphi}_1(X) = \tag{86}$$
$$\tilde{\varphi}_2(X) = \tag{87}$$
$$\tilde{\varphi}_3(X) = \tag{88}$$
$$\tag{89}$$

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

6 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,

$$\int_{-1}^{1} g(X)dX \approx \sum_{j=0}^{M} w_j \bar{X}_j, \tag{90}$$

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.

6.1 Basic integration rules with uniform point distribution

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

$$\int_{-1}^{1} g(X)dX \approx 2g(0), \quad \bar{X}_0 = 0,\ w_0 = 2, \tag{91}$$

the *Trapezoidal rule*,

$$\int_{-1}^{1} g(X)dX \approx g(-1) + g(1), \quad \bar{X}_0 = -1,\ \bar{X}_1 = 1,\ w_0 = w_1 = 1, \tag{92}$$

and *Simpson's rule*,

$$\int_{-1}^{1} g(X)dX \approx \frac{1}{3}\left(g(-1) + 4g(0) + g(1)\right), \tag{93}$$

where

$$\bar{X}_0 = -1,\ \bar{X}_1 = 0,\ \bar{X}_2 = 1,\ w_0 = w_2 = \frac{1}{3},\ w_1 = \frac{4}{3}. \tag{94}$$

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 $\int_a^b f(x)dx$.

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

$$M = 1: \quad \bar{X}_0 = -\frac{1}{\sqrt{3}},\ \bar{X}_1 = \frac{1}{\sqrt{3}},\ w_0 = w_1 = 1 \tag{95}$$

$$M = 2: \quad \bar{X}_0 = -\sqrt{\frac{3}{5}},\ \bar{X}_0 = 0,\ \bar{X}_2 = \sqrt{\frac{3}{5}},\ w_0 = w_2 = \frac{5}{9},\ w_1 = \frac{8}{9}. \tag{96}$$

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.

7 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.

7.1 Global basis functions

An example will demonstrate the necessary extensions to use global basis functions and the least squares, Galerkin, 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 $f(x,y)$ is defined completely analogously to the 1D case (24):

$$(f,g) = \int_\Omega f(x,y)g(x,y)dxdy \tag{97}$$

Given $\Omega = [0, L_x] \times [0, L_y]$ and

$$f(x,y) = (1+x^2)(1+2y^2),$$

we want to find $u = \sum_{j=0}^{N} c_i\varphi_i(x,y)$.

Constructing 2D basis functions 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. Straightforward calculations give

$$
\begin{aligned}
A_{0,0} &= (\varphi_0, \varphi_0) = \int_0^{L_y}\int_0^{L_x} \varphi_0(x,y)^2 dxdy = \int_0^{L_y}\int_0^{L_x} dxdy = L_x L_y,\\
A_{1,0} &= (\varphi_1, \varphi_0) = \int_0^{L_y}\int_0^{L_x} x dxdy = \frac{1}{2}L_x^2 L_y,\\
A_{0,1} &= (\varphi_0, \varphi_1) = \int_0^{L_y}\int_0^{L_x} y dxdy = \frac{1}{2}L_y^2 L_x,\\
A_{0,1} &= (\varphi_0, \varphi_1) = \int_0^{L_y}\int_0^{L_x} xy dxdy = \int_0^{L_y} ydy \int_0^{L_x} xdx \frac{1}{4}L_y^2 L_x^2 .
\end{aligned}
$$

The right-hand side vector has the entries

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

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

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

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 and $i = qN_y + q$ and $j = sN_y + r$.

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

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

and

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

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 $L_x = L_y = 2$

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

Figure 23 illustrates the result.

7.2 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:

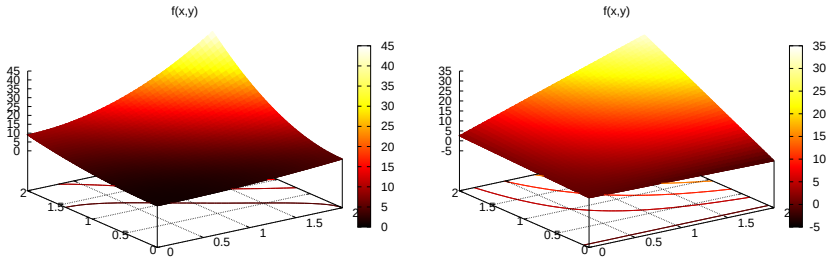

Figure 23: Approximation of a 2D quadratic function (left) by a 2D bilinear function (right) using the Galerkin or least squares method.

```
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
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
```

8 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. Two popular cell shapes are triangles and the quadrilaterals, see Figures 24, 25, and 26 provide examples.

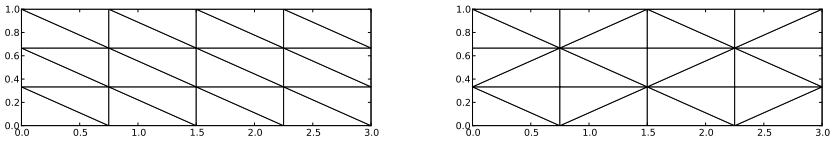

Figure 24: Examples on 2D P1 elements.

8.1 Basis functions over triangles in the physical domain

We start with triangles. A simple choice of approximation is to let u be linear over each triangle, as depicted in Figure 27.

We give the vertices global and local numbers and associate a basis function $\varphi_i(x, y)$ with vertex number i: $\varphi_i(x, y) = 1$ at this vertex and zero at all others. Moreover, $\varphi_i(x, y)$ is a linear function $a + bx + cy$ over each cell. The $\varphi_i(x, y)$ function is then a combination of planes over all 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 can only overlap φ_j (i.e., $\varphi_i\varphi_j \neq 0$) if 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 are, as in 1D, collected in an element matrix. The size of the element matrix becomes 3×3 since there are three vertex numbers for i and j. Again, as in 1D, we number the local vertices in a cell, starting at 0, and add the elements the element matrix into the global system matrix, exactly as in 1D. All details and code appear below.

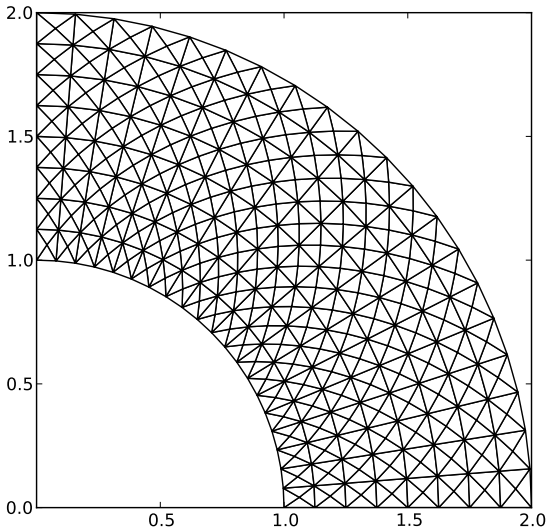

Figure 25: Examples on 2D P1 elements in a deformed geometry.

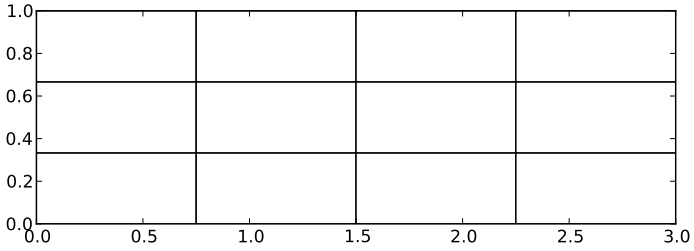

Figure 26: Examples on 2D Q1 elements.

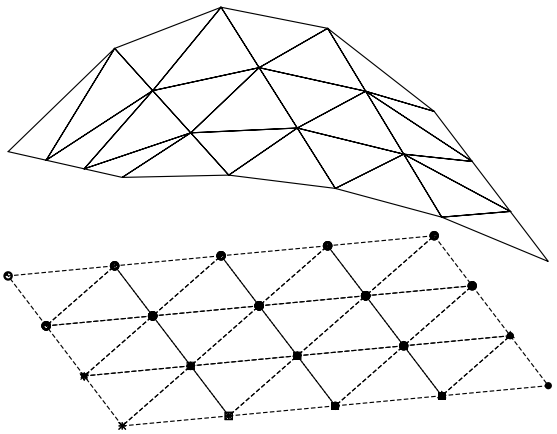

Figure 27: Example on piecewise linear 2D functions defined on triangles.

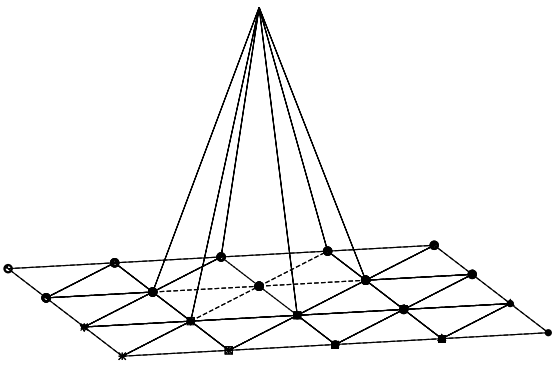

Figure 28: Example on a piecewise linear 2D basis function over a patch of triangles.

Explicit formulas for the basis functions and their integrals. We shall now provide some formulas for piecewise linear φ_i functions and their integrals

in the physical coordinate system. Let $\Omega^{(e)}$ be cell number e. The three vertices have global vertex numbers $I = q(e,0)$, $J = q(e,1)$, and $K = q(e,2)$. 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

$$\varphi_I(x,y) = \frac{1}{2}\Delta\left(\alpha_I + \beta_I x + \gamma_I y\right), \tag{98}$$

where

$$\alpha_I = x_J y_K - x_K y_J, \tag{99}$$

$$\beta_I = y_J - y_K, \tag{100}$$

$$\gamma_I = x_K - x_J, \tag{101}$$

$$2\Delta = \det \begin{pmatrix} 1 & x_I & y_I \\ 1 & x_J & y_J \\ 1 & x_K & y_K \end{pmatrix}, \tag{102}$$

The quantity Δ is the area of the cell.

The following formula can often be applied to easily compute the element matrix:

$$\int_{\Omega^{(e)}} \varphi_I^p \varphi_J^q \varphi_K^r dxdy = \frac{p!q!r!}{(p+q+r+2)!} 2\Delta\,. \tag{103}$$

In the present application with approximation, we need this integral for $p = q = 1$ and $r = 0$, when $I \neq J$, and $p = 2$ and $q = r = 0$, when $I = J$. The element matrix then becomes

$$\frac{\Delta}{12}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

8.2 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 to global degrees of freedom.

The reference cell in a (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 in 2D is on the form $C_{r,0} + C_{r,1}X + C_{r,2}Y$, and hence has three parameters $C_{r,0}$, $C_{r,1}X$, and $C_{r,2}Y$, we need three degrees of freedom, which for the P1 element are the function values at the vertices. Requiring $\tilde{\varphi}_r = 1$ at vertex number r and $\tilde{\varphi}_r = 0$ at the two other vertices, gives three linear equations to determine $C_{r,0}$, $C_{r,1}X$, and $C_{r,2}Y$. The result is

$$\tilde{\varphi}_0(X,Y) = 1 - X - Y, \tag{104}$$
$$\tilde{\varphi}_1(X,Y) = X, \tag{105}$$
$$\tilde{\varphi}_2(X,Y) = Y \tag{106}$$

The element is depicted in Figure 29.

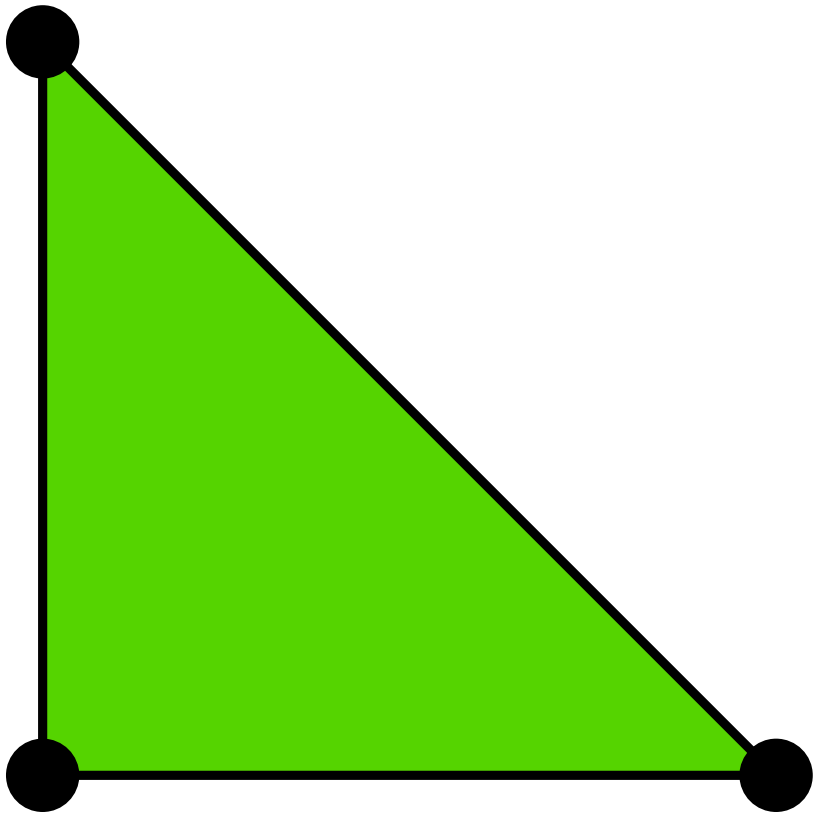

Figure 29: 2D P1 element.

Higher-order approximations are obtained by increasing the polynomial order, adding additional points (nodes), and letting the degrees of freedom be function values at the points. Figure 30 shows the six points (nodes) in the P2 element that are needed to uniquely determine quadratic basis functions.

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

The generalization to 3D is straightforward: the reference element is a tetra-

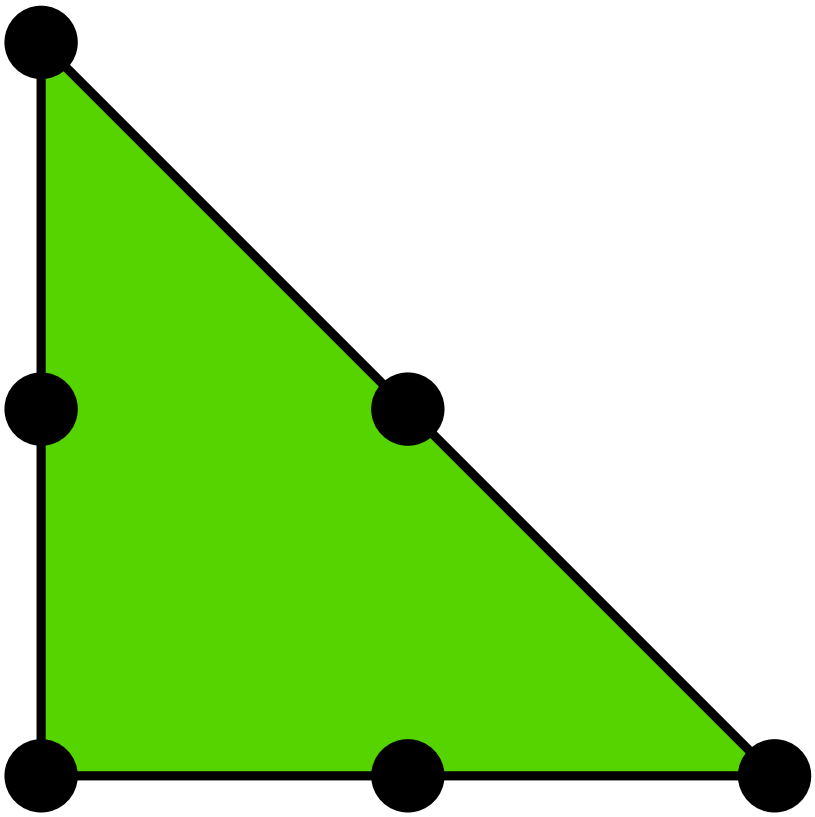

Figure 30: 2D P2 element.

hedron[5] 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 function values at the vertices, see Figure 32, while the P2 element adds additional points (nodes) along the edges and faces of the cell as depicted in Figure 33.

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 form simplices[6] or simplexes is also a much used term 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*

[5] `http://en.wikipedia.org/wiki/Tetrahedron`
[6] `http://en.wikipedia.org/wiki/Simplex`

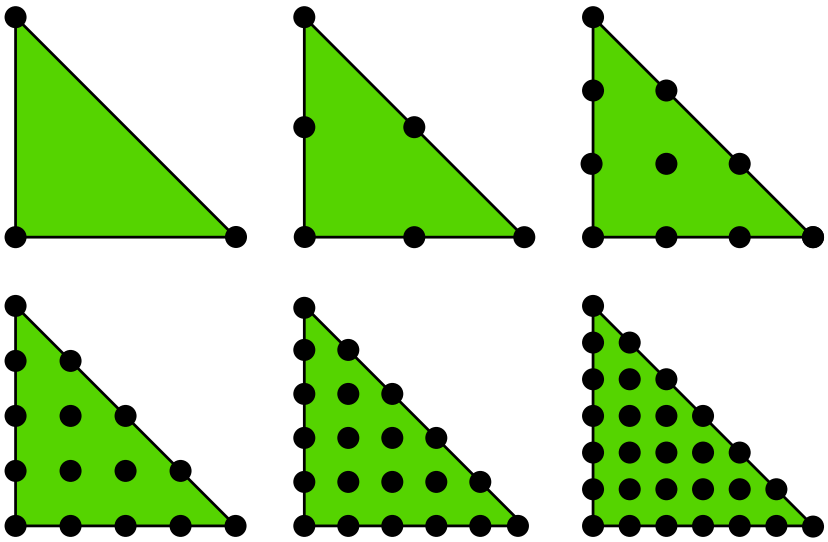

Figure 31: 2D P1, P2, P3, P4, P5, and P6 elements.

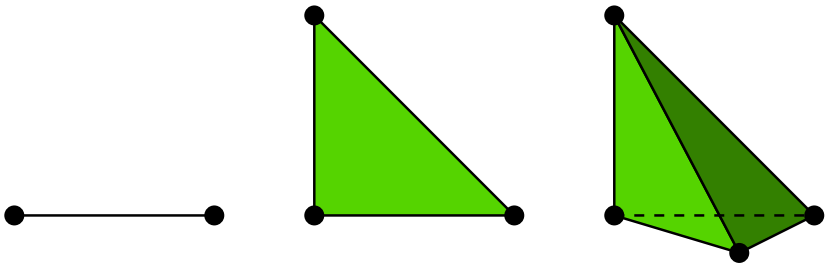

Figure 32: P1 elements in 1D, 2D, and 3D.

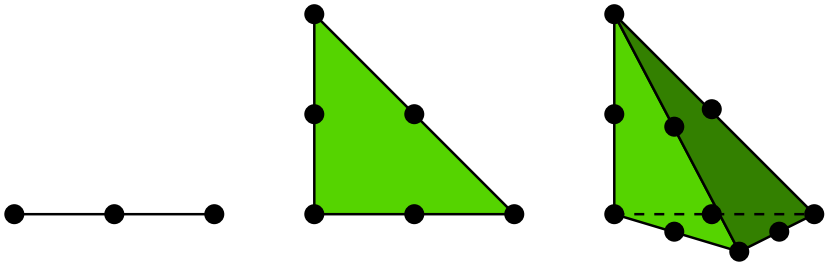

Figure 33: P2 elements in 1D, 2D, and 3D.

Finite Element Method[7], edited by A. Logg, K.-A. Mardal, and G. N. Wells,

[7] `https://launchpad.net/fenics-book`

published by Springer[8], 2012.

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

$$\boldsymbol{x} = \sum_r \tilde{\varphi}_r^{(1)}(\boldsymbol{X})\boldsymbol{x}_{q(e,r)}, \tag{107}$$

where r runs over the vertices in the cell.

8.4 Isoparametric mapping of the reference cell

Instead of using the P1 basis functions in the mapping 107, regardless of the degree of the basis functions on the element, we may use the basis functions themselves:

$$\boldsymbol{x} = \sum_r \tilde{\varphi}_r(\boldsymbol{X})\boldsymbol{x}_{q(e,r)}, \tag{108}$$

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 (107), but for higher-order elements the mapping of the straight 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 (108), the straight faces of the reference triangle are mapped onto curved sides of parabolic shape in the physical coordinate system.

From (107) or (108) 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 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.

8.5 Computing integrals

Let Ω^{ref} 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

[8] http://www.springer.com/mathematics/computational+science+%26+engineering/book/978-3-642-23098-1

$$\int_{\Omega^{(e)}} \varphi_i(\boldsymbol{x})\varphi_j(\boldsymbol{x})d\boldsymbol{x} = \int_{\Omega^{\text{ref}}} \tilde{\varphi}_i(\boldsymbol{X})\tilde{\varphi}_j(\boldsymbol{X}) \det J \, d\boldsymbol{X}, \tag{109}$$

$$\int_{\Omega^{(e)}} \varphi_i(\boldsymbol{x})f(\boldsymbol{x})d\boldsymbol{x} = \int_{\Omega^{\text{ref}}} \tilde{\varphi}_i(\boldsymbol{X})f(\boldsymbol{x}(\boldsymbol{X})) \det J \, d\boldsymbol{X}, \tag{110}$$

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,

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

With the affine mapping (107), $\det J = 2\Delta$, 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.

9 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. Filename: `vec111_approx1.py`.

Exercise 2: Linear algebra refresher II

As an extension of Exercise 1, check out the topic of *inner vector spaces*. Show that both examples of spaces in Exercise 1 can be equipped with an inner product and show that the choice of inner product satisfied the general requirements of an inner product in a vector space. Filename: `vec111_approx1.py`.

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.py`.

Exercise 4: Approximate the exponential function by power functions

Let V be a function space with basis functions x^k, $k = 0, 1, \ldots, N$. Find the best approximation to $f(x) = e^x$ among all functions in V, using $N = 8$ and the `least_squares` function from Section 2. Filename: `exp_by_powers.py`.

Exercise 5: Approximate a high frequency sine function by lower frequency sines

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

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

using the `least_squares_orth` function from Section 2.7. Plot $f(x)$ and its approximation. Filename: `hilow_sine_approx.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 a specific function $f(x)$, calculate the coefficients in the Fourier expansion by solving the linear system, arising from the least squares method, by hand. Plot some truncated versions of the series together with $f(x)$ to show how the series expansion converges. Filename: `tanh_approx.py`.

Exercise 7: Approximate a tanh function by Lagrange polynomials

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

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

by Lagrange polynomials for $s = 10, 100$ and $N = 3, 6, 9, 11$. Filename: `tanh_approx.py`.

Exercise 8: Define finite element meshes

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

Exercise 9: Perform symbolic finite element computations

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

Exercise 10: 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 11: Approximate a step function by finite elements

Approximate the step function

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

by 2, 4, and 8 P1 and P2 elements. Compare approximations visually. Filename: `'Heaviside_approx_P1P2.py.'`.

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.

Exercise 12: 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

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

Filename: `approx2D_lsorth_sin.py`.

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

Index