Study Guide: Approximation of functions with finite elements

Hans Petter Langtangen[1,2]

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

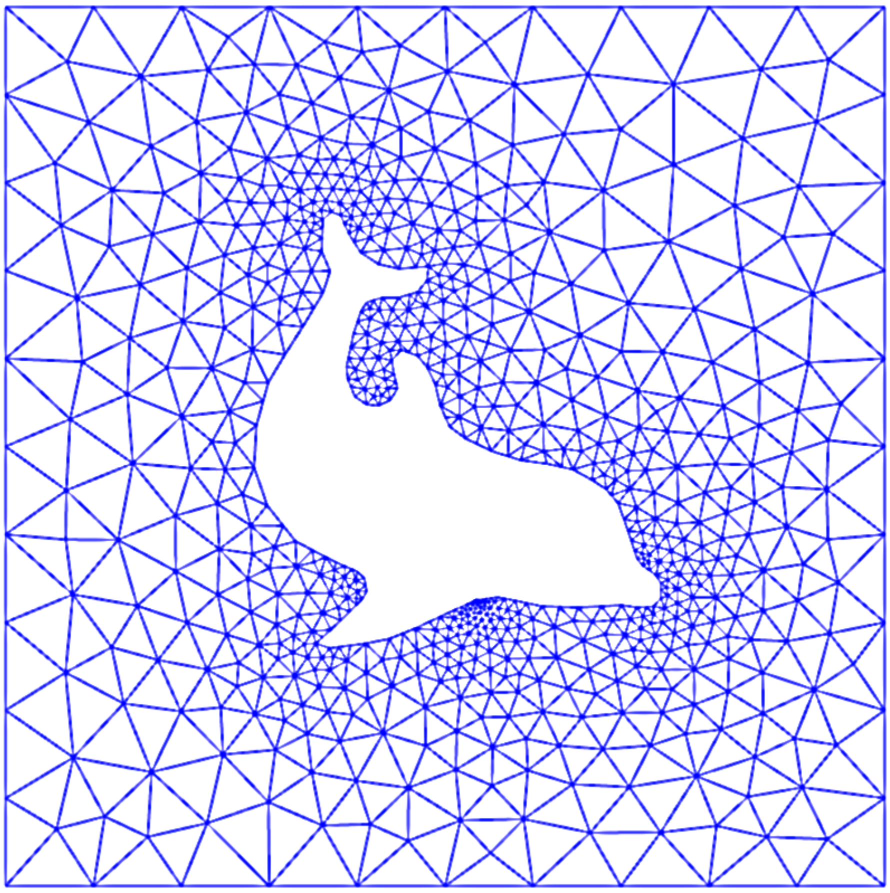
Oct 16, 2013

Contents

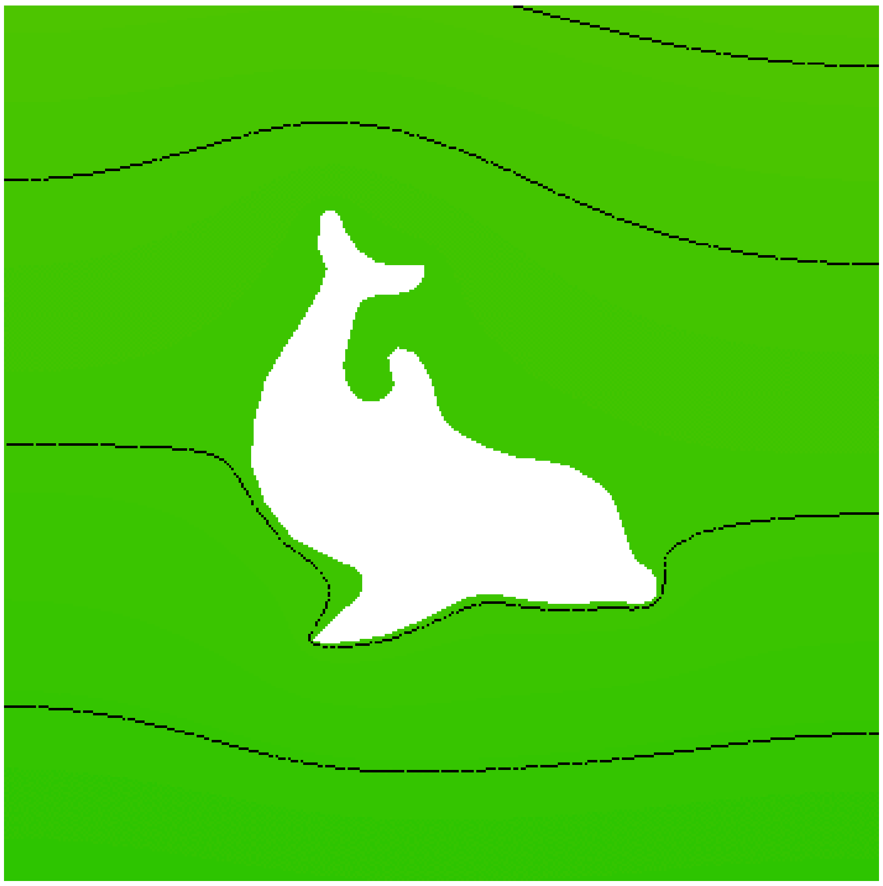
1 Why finite elements?

- Can with ease solve PDEs in domains with *complex geometry*
- Can with ease provide higher-order approximations
- Has (in simpler stationary problems) a rigorus mathematical analysis framework (not much considered here)

1.1 Domain for flow around a dolphin



1.2 The flow



1.3 Basic ingredients of the finite element method

- Transform the PDE problem to a *variational form*
- Define function approximation over *finite elements*
- Use a machinery to derive *linear systems*
- Solve linear systems

1.4 Our learning strategy

- Start with approximation of functions, not PDEs
- Introduce finite element *approximations*
- See later how this is applied to PDEs

Reason: the finite element method has many concepts and a jungle of details. This strategy minimizes the mixing of ideas, concepts, and technical details.

1.5 Approximation set-up

General idea:

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

where

- $\psi_i(x)$ are prescribed functions
- c_i, $i = 0, \ldots, N$ are unknown coefficients to be determined

1.6 How to determine the coefficients?

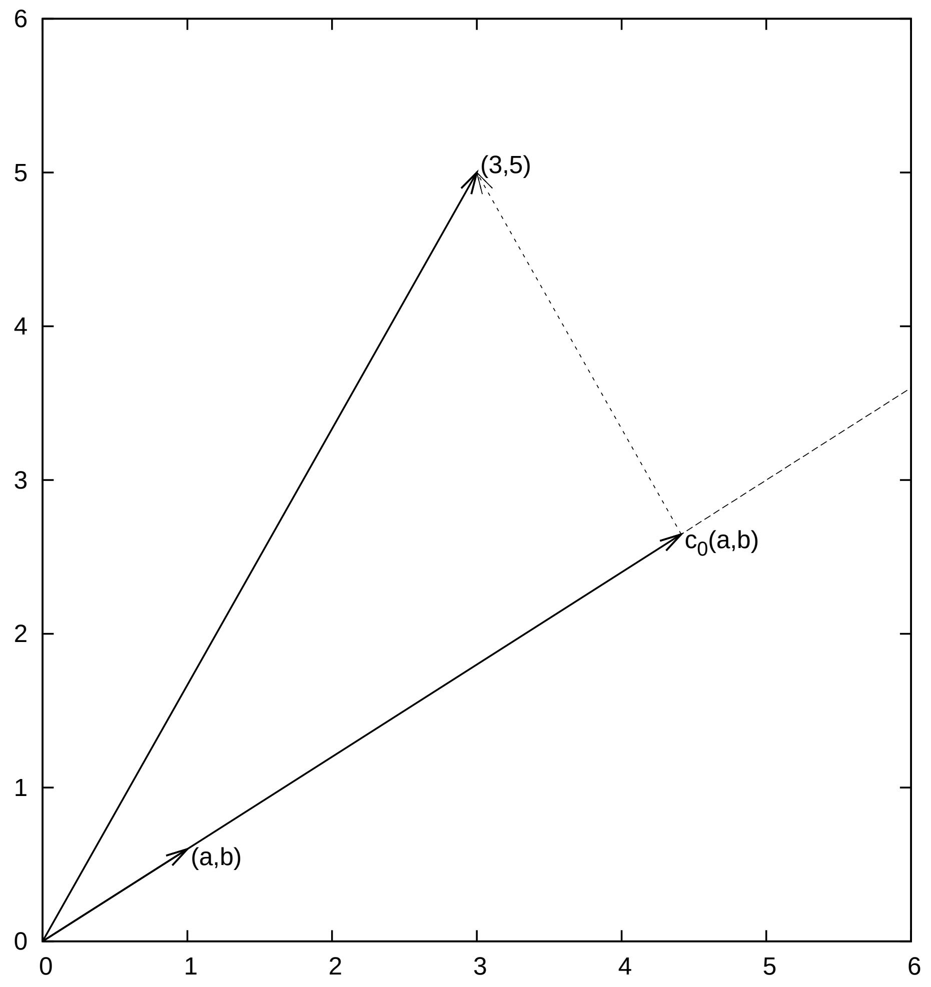
- least squares method
- projection or Galerkin method
- interpolation (or collocation) method

Underlying motivation for our notation.
Our mathematical framework for doing this is phrased in a way such that it becomes easy to understand and use the FEniCS[a] software package for finite element computing.

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

1.7 Approximation of planar vectors; problem

Given a vector $\boldsymbol{f} = (3, 5)$, find an approximation to $\boldsymbol{f}$ directed along a given line.



1.8 Approximation of planar vectors; vector space terminology

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

- $\boldsymbol{\psi}_0$ is a basis vector in the space V
- Seek $\boldsymbol{u} = c_0\boldsymbol{\psi}_0 \in V$
- Determine c_0 such that $\boldsymbol{u}$ is the "best" approximation to $\boldsymbol{f}$
- Visually, "best" is obvious

Define

- the error $\boldsymbol{e} = \boldsymbol{f} - \boldsymbol{u}$
- the (Eucledian) scalar product of two vectors: $(\boldsymbol{u}, \boldsymbol{v})$
- the norm of $\boldsymbol{e}$: $||\boldsymbol{e}|| = \sqrt{(\boldsymbol{e}, \boldsymbol{e})}$

1.9 The least squares method; principle

- Idea: find c_0 such that $||\boldsymbol{e}||$ is minimized
- Actually, we always minimize $E = ||\boldsymbol{e}||^2$

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

1.10 The least squares method; calculations

$$E(c_0) = (\boldsymbol{e}, \boldsymbol{e}) = (\boldsymbol{f}, \boldsymbol{f}) - 2c_0(\boldsymbol{f}, \boldsymbol{\psi}_0) + c_0^2(\boldsymbol{\psi}_0, \boldsymbol{\psi}_0) \tag{3}$$

$$\frac{\partial E}{\partial c_0} = -2(\boldsymbol{f}, \boldsymbol{\psi}_0) + 2c_0(\boldsymbol{\psi}_0, \boldsymbol{\psi}_0) = 0 \tag{4}$$

$$c_0 = \frac{(\boldsymbol{f}, \boldsymbol{\psi}_0)}{(\boldsymbol{\psi}_0, \boldsymbol{\psi}_0)} \tag{5}$$

$$c_0 = \frac{3a + 5b}{a^2 + b^2} \tag{6}$$

Observation for later: the vanishing derivative (4) can be alternatively written as

$$(\boldsymbol{e}, \boldsymbol{\psi}_0) = 0\,. \tag{7}$$

1.11 The projection (or Galerkin) method

- Backgrund: minimizing $||\boldsymbol{e}||^2$ implies that $\boldsymbol{e}$ is orthogonal to *any* vector $\boldsymbol{v}$ in the space V (visually clear, but can easily be computed too)
- Alternative idea: demand $(\boldsymbol{e}, \boldsymbol{v}) = 0, \quad \forall \boldsymbol{v} \in V$
- Equivalent statement: $(\boldsymbol{e}, \boldsymbol{\psi}_0) = 0$ (see notes for why)
- Insert $\boldsymbol{e} = \boldsymbol{f} - c_0\boldsymbol{\psi}_0$ and solve for c_0
- Same equation for c_0 and hence same solution as in the least squares method

1.12 Approximation of general vectors

Given a vector $\boldsymbol{f}$, find an approximation $\boldsymbol{u} \in V$:

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

- We have a set of linearly independent basis vectors $\boldsymbol{\psi}_0, \ldots, \boldsymbol{\psi}_N$
- Any $\boldsymbol{u} \in V$ can then be written as $\boldsymbol{u} = \sum_{j=0}^{N} c_j\boldsymbol{\psi}_j$

1.13 The least squares method

Idea: find $c_0, \ldots, c_N$ such that $E = ||\boldsymbol{e}||^2$ is minimized, $\boldsymbol{e} = \boldsymbol{f} - \boldsymbol{u}$.

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

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

After some work we end up with a *linear system*

$$\sum_{j=0}^{N} A_{i,j} c_j = b_i, \quad i = 0, \ldots, N \tag{8}$$

$$A_{i,j} = (\boldsymbol{\psi}_i, \boldsymbol{\psi}_j) \tag{9}$$

$$b_i = (\boldsymbol{\psi}_i, \boldsymbol{f}) \tag{10}$$

1.14 The projection (or Galerkin) method

Can be shown that minimizing $||\boldsymbol{e}||$ implies that $\boldsymbol{e}$ is orthogonal to all $\boldsymbol{v} \in V$:

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

which implies that $\boldsymbol{e}$ most be orthogonal to each basis vector:

$$(\boldsymbol{e}, \boldsymbol{\psi}_i) = 0, \quad i = 0, \ldots, N \,. \tag{11}$$

This orthogonality condition is the principle of the projection (or Galerkin) method. Leads to the same linear system as in the least squares method.

2 Approximation of functions

Let V be a *function space* spanned by a set of *basis functions* $\psi_0, \ldots, \psi_N$,

$$V = \text{span}\,\{\psi_0, \ldots, \psi_N\},$$

Find $u \in V$ as a linear combination of the basis functions:

$$u = \sum_{j \in I} c_j \psi_j, \quad I = \{0, 1, \ldots, N\} \tag{12}$$

2.1 The least squares method

- Extend the ideas from the vector case: minimize the (square) norm of the error.
- What norm? $(f, g) = \int_\Omega f(x) g(x)\, dx$

$$E = (e, e) = (f - u, f - u) = (f(x) - \sum_{j\in I} c_j\psi_j(x), f(x) - \sum_{j\in I} c_j\psi_j(x)) \tag{13}$$

$$E(c_0, \ldots, c_N) = (f, f) - 2\sum_{j\in I} c_j(f, \psi_i) + \sum_{p\in I}\sum_{q\in I} c_p c_q(\psi_p, \psi_q) \tag{14}$$

$$\frac{\partial E}{\partial c_i} = 0, \quad i =\in I$$

After computations *identical to the vector case*, we get a linear system

$$\sum_{j\in I}^{N} A_{i,j}c_j = b_i, \quad i \in I \tag{15}$$

$$A_{i,j} = (\psi_i, \psi_j) \tag{16}$$

$$b_i = (f, \psi_i) \tag{17}$$

2.2 The projection (or Galerkin) method

As before, minimizing (e, e) is equivalent to the projection (or Galerkin) method

$$(e, v) = 0, \quad \forall v \in V, \tag{18}$$

which means, as before,

$$(e, \psi_i) = 0, \quad i \in I\,. \tag{19}$$

With the same algebra as in the multi-dimensional vector case, we get the same linear system as arose from the least squares method.

2.3 Example: linear approximation; problem

Problem.
Approximate a parabola $f(x) = 10(x - 1)^2 - 1$ by a straight line.

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

That is, $\psi_0(x) = 1$, $\psi_1(x) = x$, and $N = 1$. We seek

$$u = c_0\psi_0(x) + c_1\psi_1(x) = c_0 + c_1x,$$

2.4 Example: linear approximation; solution

$$A_{0,0} = (\psi_0, \psi_0) = \int_1^2 1 \cdot 1\,dx = 1 \tag{20}$$

$$A_{0,1} = (\psi_0, \psi_1) = \int_1^2 1 \cdot x\,dx = 3/2 \tag{21}$$

$$A_{1,0} = A_{0,1} = 3/2, \tag{22}$$

$$A_{1,1} = (\psi_1, \psi_1) = \int_1^2 x \cdot x\,dx = 7/3 \tag{23}$$

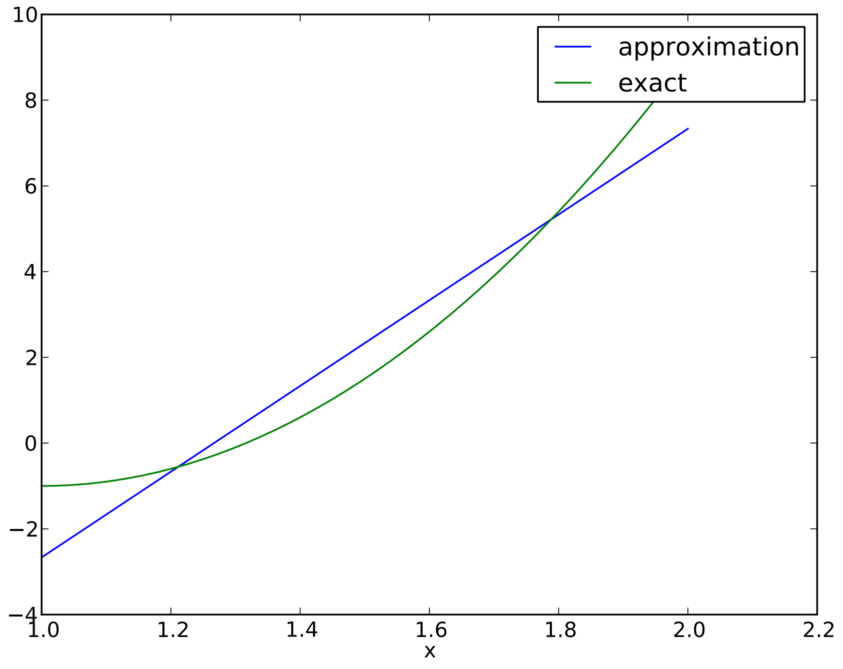
$$b_1 = (f, \psi_0) = \int_1^2 (10(x-1)^2 - 1) \cdot 1 \, dx = 7/3 \tag{24}$$

$$b_2 = (f, \psi_1) = \int_1^2 (10(x-1)^2 - 1) \cdot x \, dx = 13/3 \tag{25}$$

Solution of 2x2 linear system:

$$c_0 = -38/3, \quad c_1 = 10, \quad u(x) = 10x - \frac{38}{3} \tag{26}$$

2.5 Example: linear approximation; plot



2.6 Implementation of the least squares method; ideas

Consider 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 $\{\psi_i\}_{i\in I}$,
- `Omega` is a 2-tuple/list holding the domain Ω

Carry out the integrations, solve the linear system, and return $u(x) = \sum_j c_j \psi_j(x)$

2.7 Implementation of the least squares method; code

```
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: symmetric coefficient matrix so we can halve the integrations.

2.8 Implementation of the least squares method; plotting

Compare f and u visually:

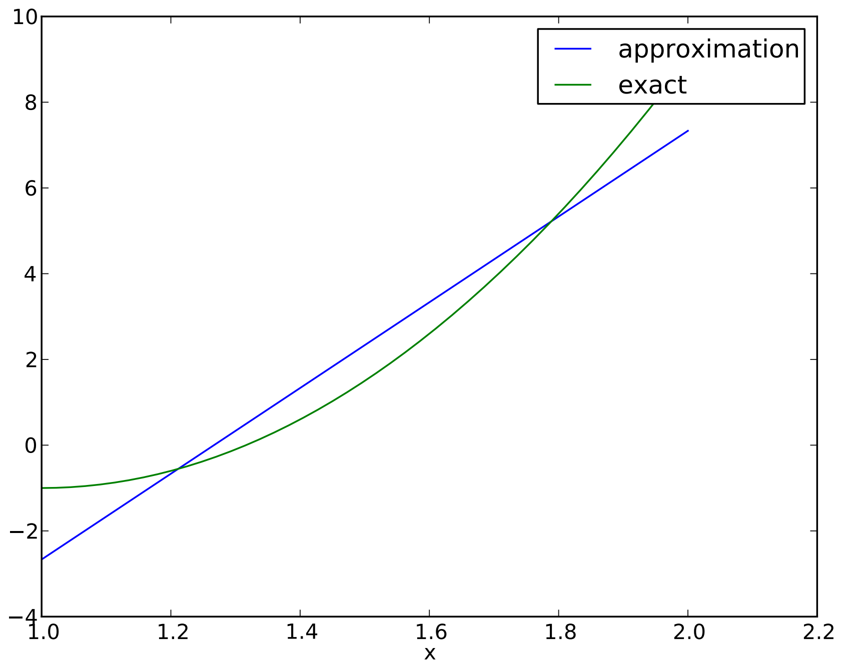
```
def comparison_plot(f, u, Omega, filename='tmp.pdf'):
    x = sm.Symbol('x')
    # Turn f and u to ordinary Python functions
    f = sm.lambdify([x], f, modules="numpy")
    u = sm.lambdify([x], u, modules="numpy")
    resolution = 401  # no of points in plot
    xcoor  = linspace(Omega[0], Omega[1],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

All code in module `approx1D.py`[1]

2.9 Implementation of the least squares method; application

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

[1] `http://tinyurl.com/jvzzcfn/fem/approx1D.py`



2.10 Perfect approximation; parabola approximating parabola

- What if we add $\psi_2 = x^2$ to the space V?
- That is, approximating a parabola by any parabola?
- (Hopefully we get the exact parabola!)

```
>>> 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
```

2.11 Perfect approximation; the general result

- What if we use $\phi_i(x) = x^i$ for $i = 0, \ldots, N = 40$?
- The output from `least_squares` is $c_i = 0$ for $i > 2$

General result.
If $f \in V$, least squares and projection/Galerkin give $u = f$.

2.12 Perfect approximation; proof of the general result

If $f \in V$, $f = \sum_{j\in I} d_j\psi_j$, for some $\{d_i\}_{i\in I}$. Then

$$b_i = (f, \psi_i) = \sum_{j\in I} d_j(\psi_j, \psi_i) = \sum_{j\in I} d_j A_{i,j}\,.$$

The linear system $\sum_j A_{i,j}c_j = b_i$, $i \in I$, is then

$$\sum_{j\in I} c_j A_{i,j} = \sum_{j\in I} d_j A_{i,j}, \quad i \in I,$$

which implies that $c_i = d_i$ for $i \in I$ and u is identical to f.

2.13 Finite-precision/numerical computations

The previous computations were symbolic. What if we solve the linear system numerically with standard arrays?

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

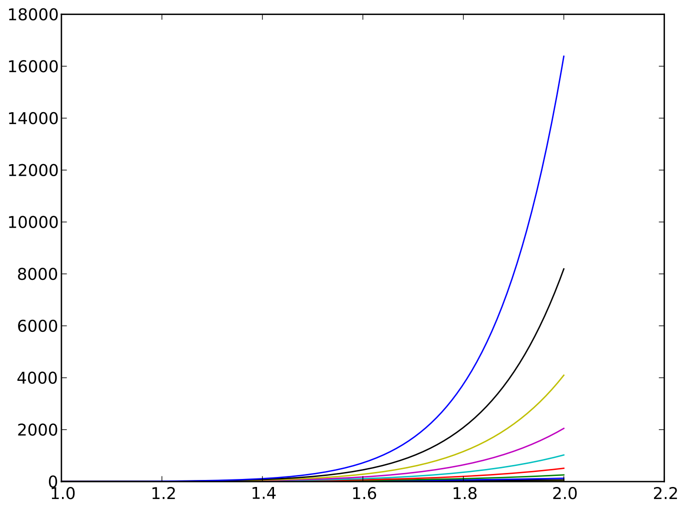
- Column 2: `sympy.mpmath.fp.matrix` and `sympy.mpmath.fp.lu_solve`
- Column 3: `numpy` arrays with `numpy.float32` entries
- Column 4: `numpy` arrays with `numpy.float64` entries

2.14 Ill-conditioning (1)

Observations:

- Significant round-off errors in the numerical computations (!)
- But if we plot the approximations they look good (!)

Problem: The basis functions x^i become almost linearly dependent for large N.



2.15 Ill-conditioning (2)

- Almost linearly dependent basis functions give almost singular matrices
- Such matrices are said to be *ill conditioned*, and Gaussian elimination is severely affected by round-off errors
- The basis $1, x, x^2, x^3, x^4, \ldots$ is a bad basis
- Polynomials are fine as basis, but the more orthogonal they are, $(\psi_i, \psi_j) \approx 0$, the better

2.16 Fourier series approximation; problem and code

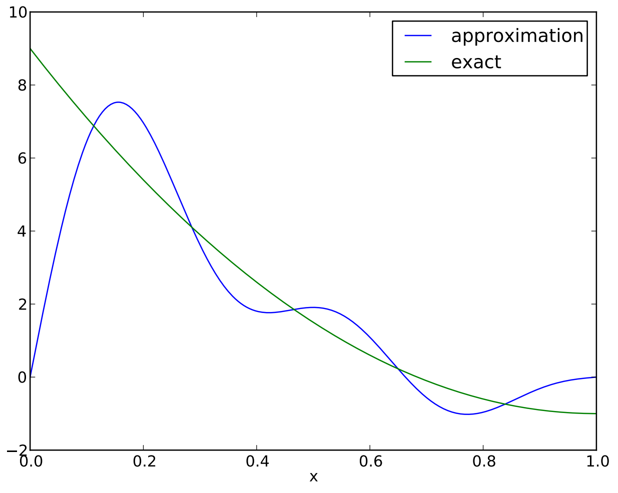
Consider

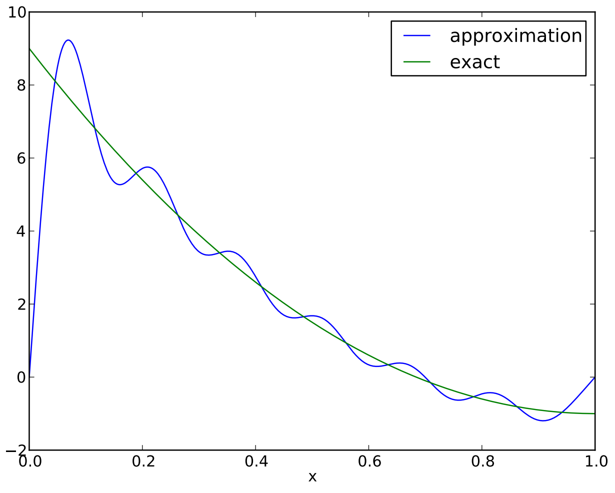
$$V = \text{span}\{\sin \pi x, \sin 2\pi x, \ldots, \sin(N+1)\pi x\}.$$

```
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)
```

2.17 Fourier series approximation; plot

$N = 3$ vs $N = 11$:





2.18 Fourier series approximation; improvements

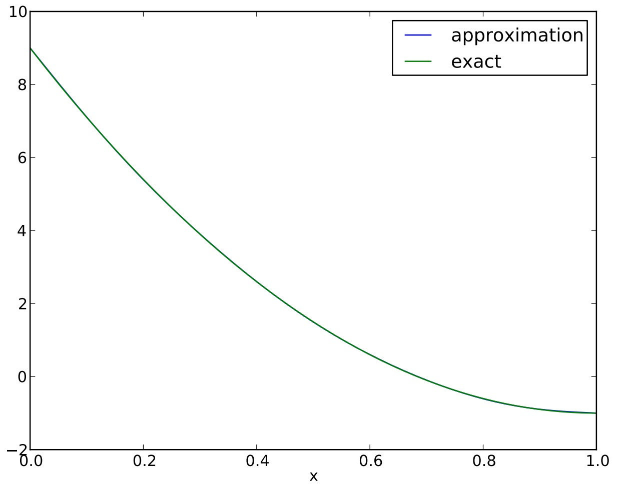
- Considerably improvement by $N = 11$
- But always discrepancy of $f(0) - u(0) = 9$ at $x = 0$, because all the $\psi_i(0) = 0$ and hence $u(0) = 0$
- Possible remedy: add a term that leads to correct boundary values

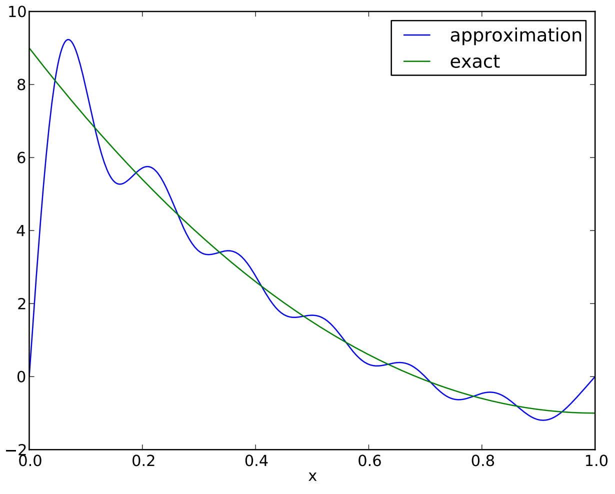
$$u(x) = f(0)(1-x) + xf(1) + \sum_{j\in I} c_j\psi_j(x)\,. \tag{27}$$

The extra term ensures $u(0) = f(0)$ and $u(1) = f(1)$ and is a strikingly good help to get a good approximation!

2.19 Fourier series approximation; final results

$N = 3$ vs $N = 11$:





2.20 Orthogonal basis functions

This choice of sine functions as basis functions is popular because

- the basis functions are orthogonal: $(\psi_i, \psi_j) = 0$
- implying that $A_{i,j}$ is a diagonal matrix
- implying that we can solve for $c_i = 2\int_0^1 f(x)\sin((i+1)\pi x)dx$

In general for an orthogonal basis, $A_{i,j}$ is diagonal and we can easily solve for c_i:

$$c_i = \frac{b_i}{A_{i,i}} = \frac{(f, \psi_i)}{(\psi_i, \psi_i)}\,.$$

2.21 The collocation or interpolation method; ideas and math

Here is another idea for approximating $f(x)$ by $u(x) = \sum_j c_j\psi_j$:

- Force $u(x_i) = f(x_i)$ at some selected *collocation* points $\{x_i\}_{i\in I}$
- Then u interpolates f
- The method is known as *interpolation* or *collocation*

$$u(x_i) = \sum_{j\in I} c_j\psi_j(x_i) = f(x_i), \quad i \in I, N\,. \tag{28}$$

This is a linear system with no need for integration:

$$\sum_{j\in I} A_{i,j}c_j = b_i, \quad i \in I \tag{29}$$

$$A_{i,j} = \psi_j(x_i) \tag{30}$$

$$b_i = f(x_i) \tag{31}$$

No symmetric matrix: $\psi_j(x_i) \neq \psi_i(x_j)$ in general

2.22 The collocation or interpolation method; implementation

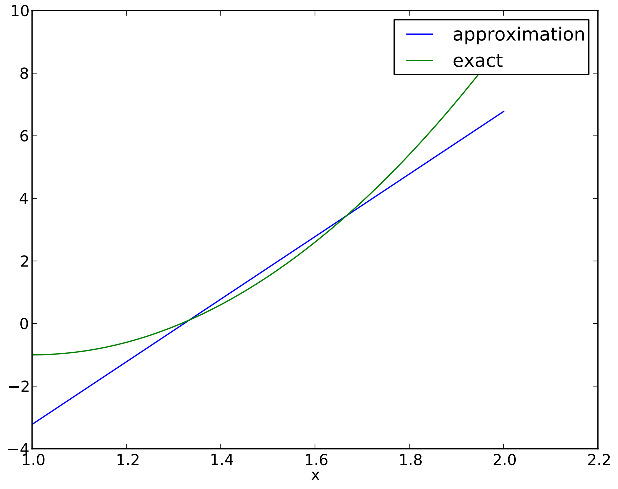
`points` holds the interpolation/collocation points

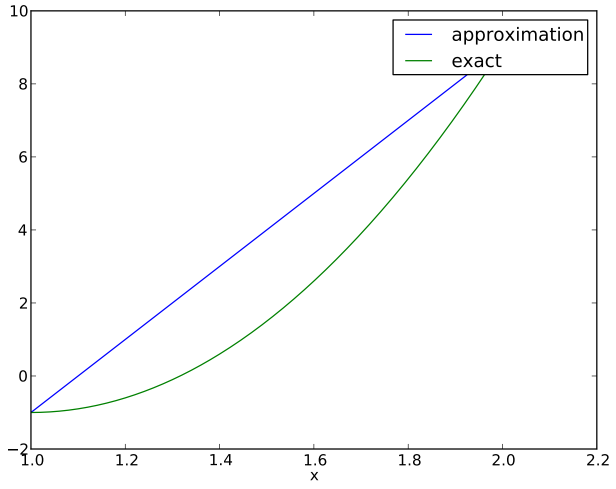
```
def interpolation(f, phi, points):
    N = len(phi) - 1
    A = sm.zeros((N+1, N+1))
    b = sm.zeros((N+1, 1))
    x = sm.Symbol('x')
    # Turn phi and f into Python functions
    phi = [sm.lambdify([x], phi[i]) for i in range(N+1)]
    f = sm.lambdify([x], f)
    for i in range(N+1):
        for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

2.23 The collocation or interpolation method; approximating a parabola by linear functions

- Potential difficulty: how to choose x_i?
- The results are sensitive to the points!

$(4/3, 5/3)$ vs $(1, 2)$:





2.24 Lagrange polynomials; motivation and ideas

Motivation:

- The interpolation/collocation method avoids integration
- With a diagonal matrix $A_{i,j} = \psi_j(x_i)$ we can solve the linear system by hand

The *Lagrange interpolating polynomials* ψ_j have the property that

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

Hence, $c_i = f(x_i)$ and

$$u(x) = \sum_{j \in I} f(x_i)\psi_i(x) \qquad (32)$$

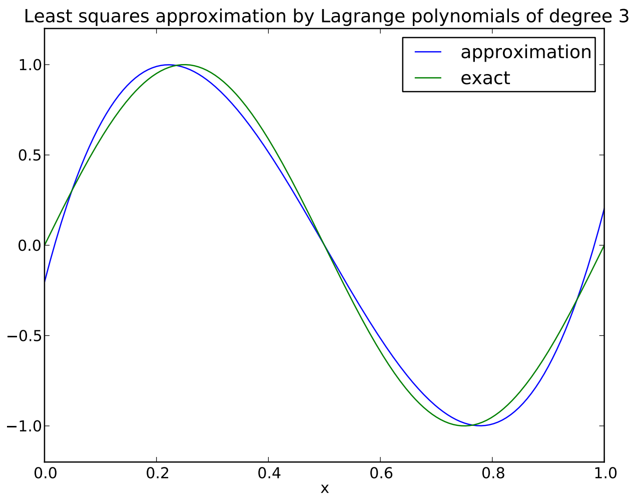
- Lagrange polynomials and interpolation/collocation look convenient
- Lagrange polynomials are very much used in the finite element method

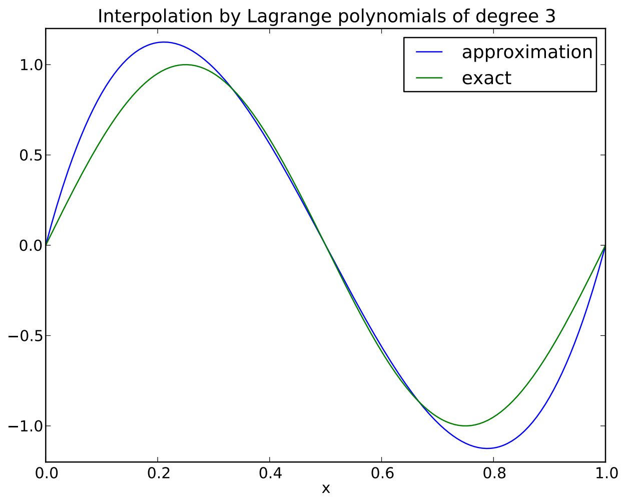
2.25 Lagrange polynomials; formula and code

$$\psi_i(x) = \prod_{j=0, j\neq i}^{N} \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_i - x_0} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_N}{x_i - x_N}, \qquad (33)$$

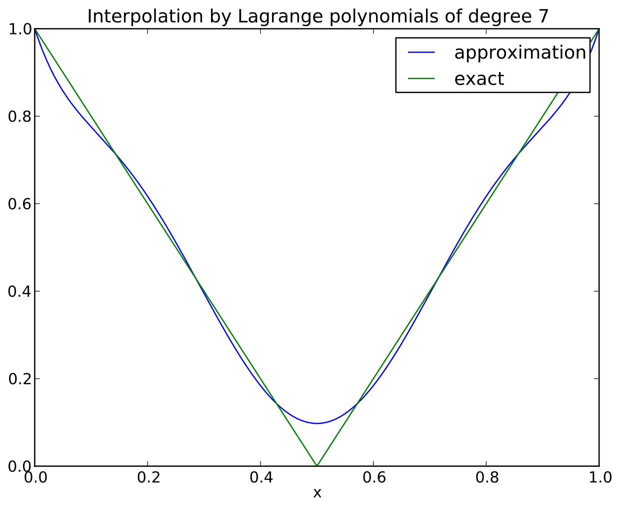
```
def Lagrange_polynomial(x, i, points):
    p = 1
    for k in range(len(points)):
        if k != i:
            p *= (x - points[k])/(points[i] - points[k])
    return p
```

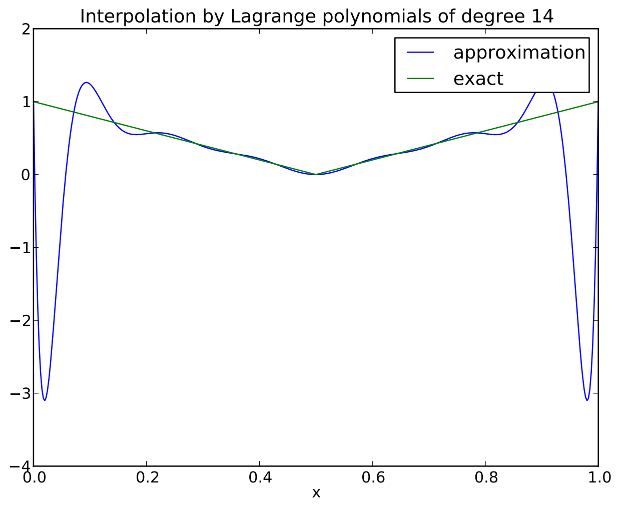
2.26 Lagrange polynomials; successful example





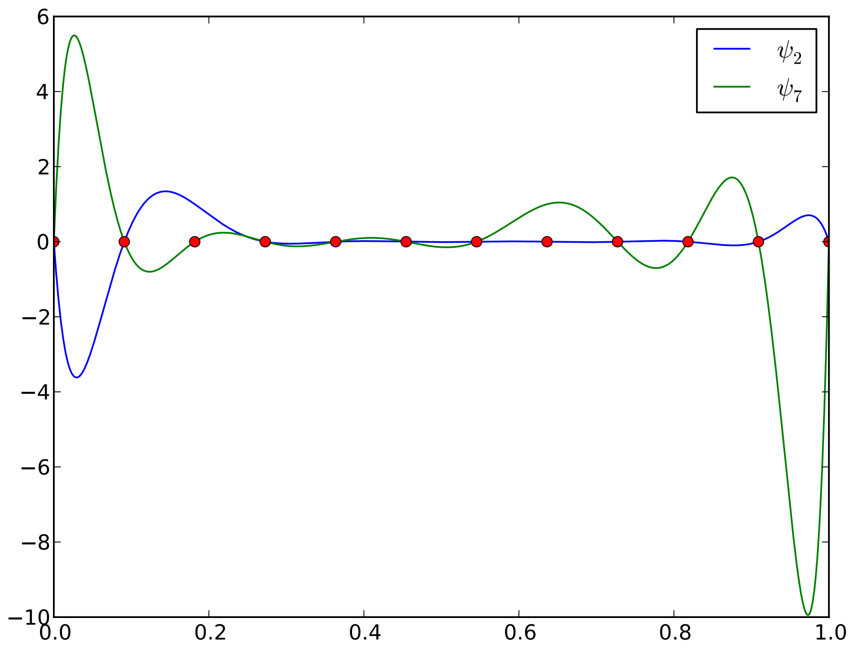
2.27 Lagrange polynomials; a less successful example





2.28 Lagrange polynomials; oscillatory behavior

12 points, degree 11, plot of two of the Lagrange polynomials - note that they are zero at all points except one.



Problem: strong oscillations near the boundaries for larger N values.

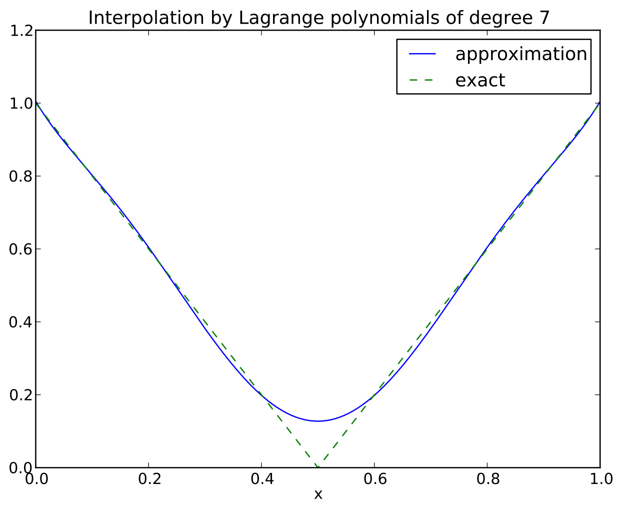
2.29 Lagrange polynomials; remedy for strong oscillations

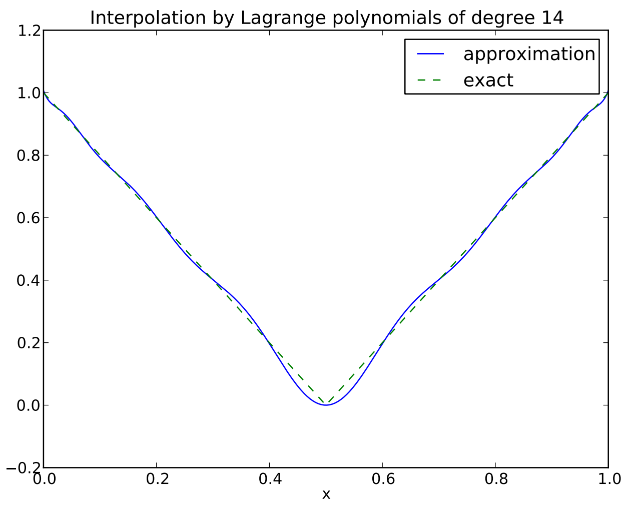
The oscillations can be reduced by a more clever choice of interpolation points, called the *Chebyshev nodes*:

$$x_i = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos\left(\frac{2i+1}{2(N+1)}pi\right), \quad i = 0\ldots,N, \tag{34}$$

on an interval $[a,b]$.

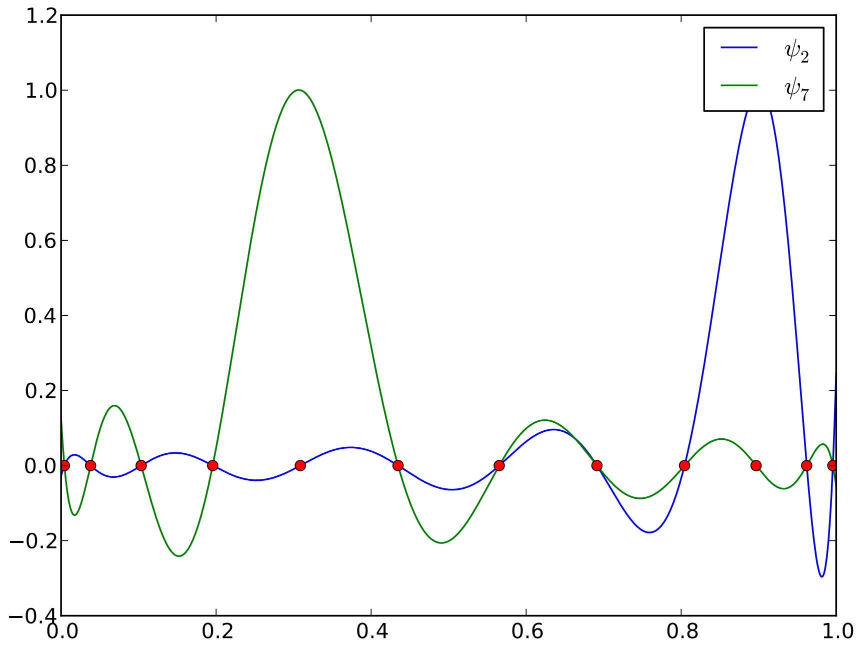
2.30 Lagrange polynomials; recalculation with Chebyshev nodes





2.31 Lagrange polynomials; less oscillations with Chebyshev nodes

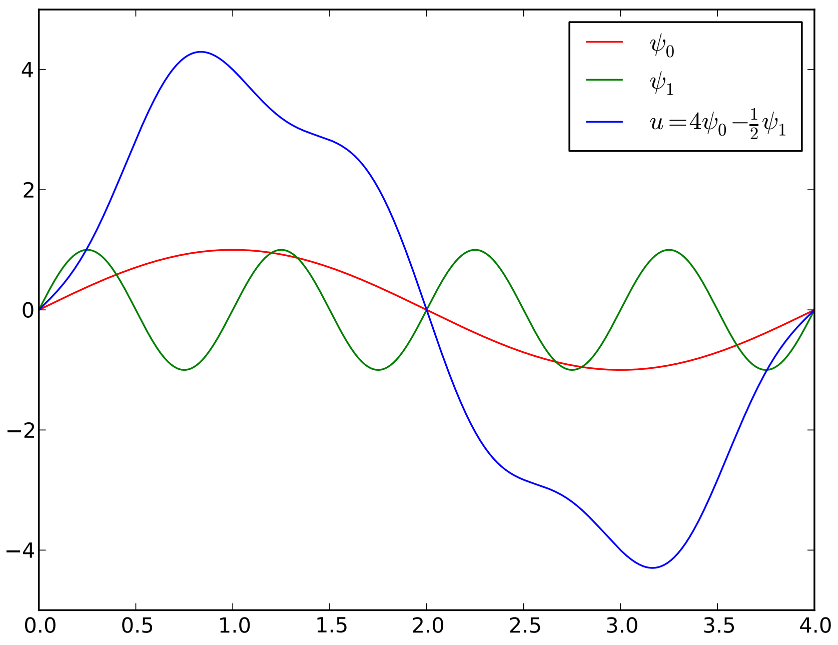
12 points, degree 11, plot of two of the Lagrange polynomials - note that they are zero at all points except one.



3 Finite element basis functions

3.1 So far: basis functions have been global

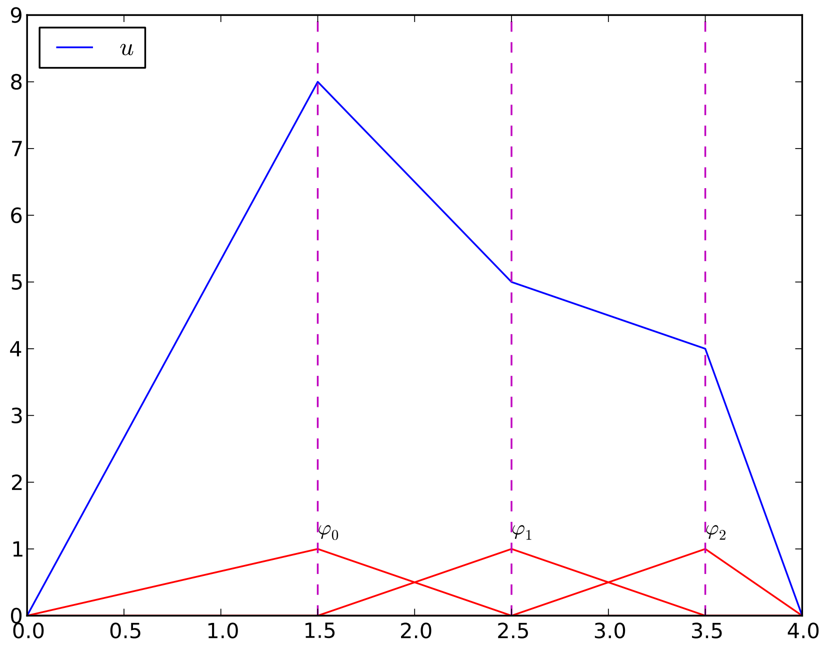
$\psi_i(x) \neq 0$ for most $x \in \Omega$



3.2 In the finite element method we use basis functions with local support

- *Local support*: $\psi_i(x) \neq 0$ for x in a small subdomain of Ω
- Typically hat-shaped
- $u(x)$ based on these ψ_i is a piecewise polynomial defined over many (small) subdomains

3.3 The linear combination of hat functions is a piecewise linear function



3.4 Elements and nodes

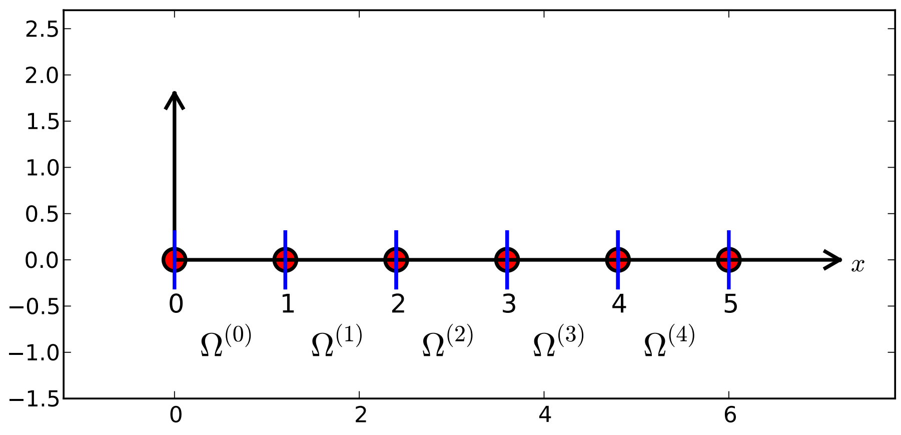
Split Ω into non-overlapping subdomains called *elements*:

$$\Omega = \Omega^{(0)} \cup \cdots \cup \Omega^{(N_e)}\,. \tag{35}$$

On each element, introduce points called *nodes*: $x_0, \ldots, x_{N_n}$

- The finite element basis functions are named $\varphi_i(x)$
- $\varphi_i = 1$ at node i and 0 at all other nodes
- φ_i is a Lagrange polynomial on each element
- For nodes at the boundary between two elements, φ_i is made up of a Lagrange polynomial over each element

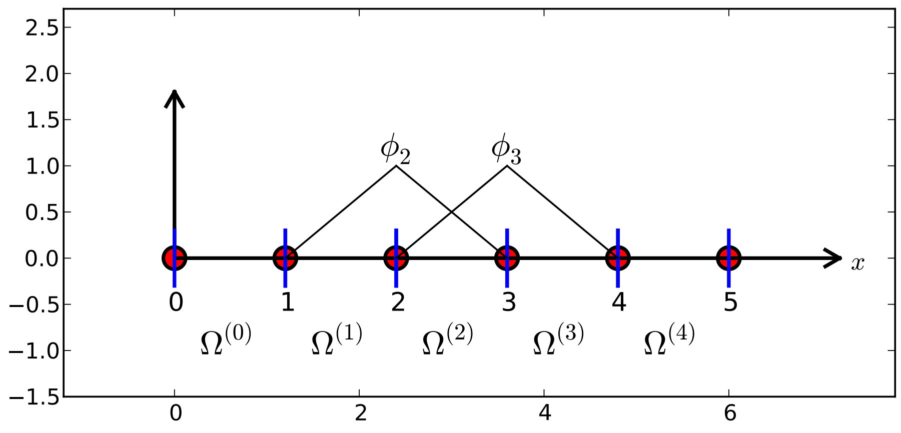
3.5 Example on elements with two nodes (P1 elements)



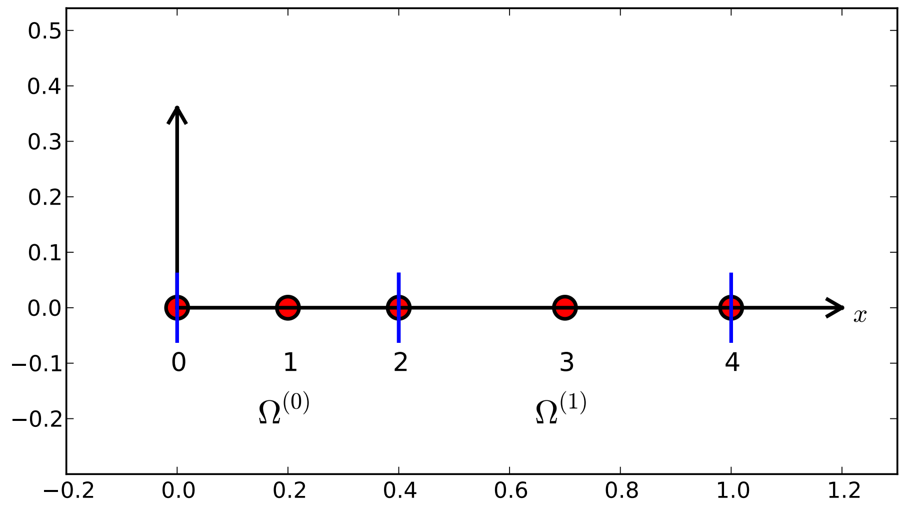
Data structure: `nodes` holds coordinates or nodes, `elements` holds the node numbers in each element

```
nodes = [0, 1.2, 2.4, 3.6, 4.8, 5]
elements = [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5]]
```

3.6 Illustration of two basis functions on the mesh

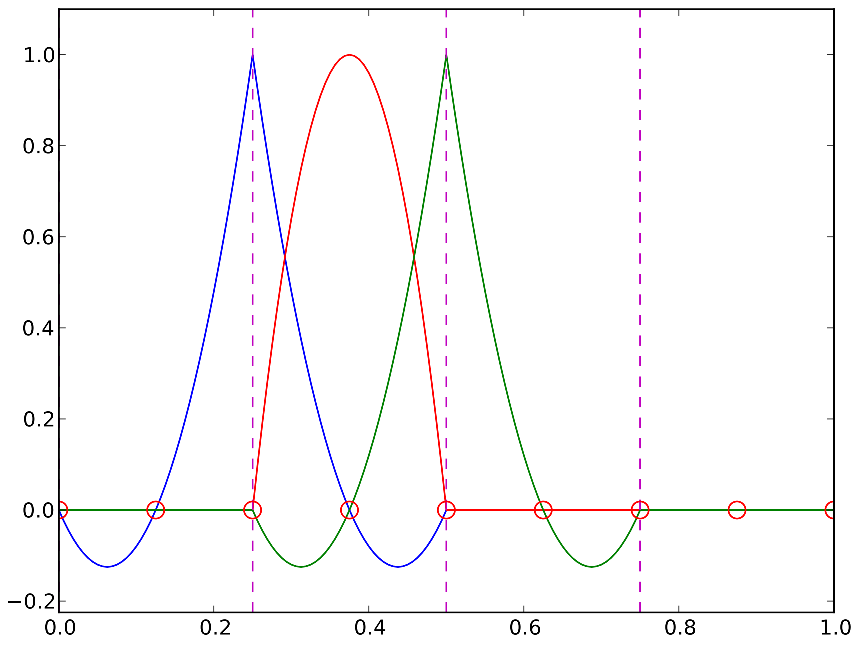


3.7 Example on elements with three nodes (P2 elements)

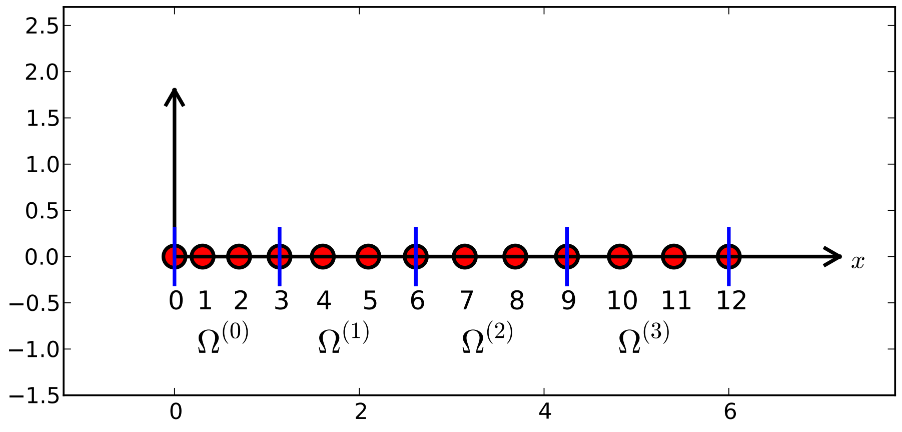


```
nodes = [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.0]
elements = [[0, 1, 2], [2, 3, 4], [4, 5, 6], [6, 7, 8]]
```

3.8 Some corresponding basis functions (P2 elements)

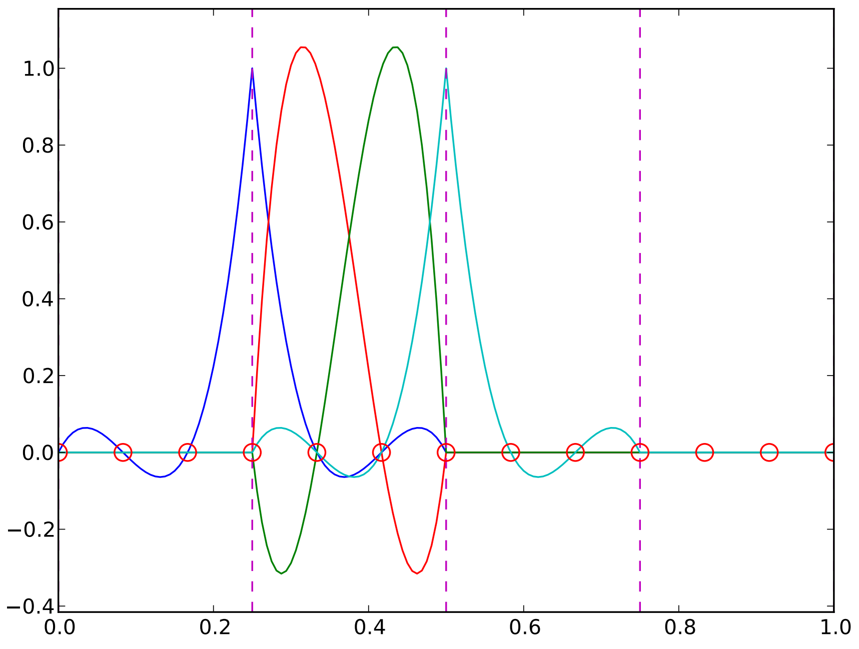


3.9 Examples on elements with four nodes per element (P3 elements)

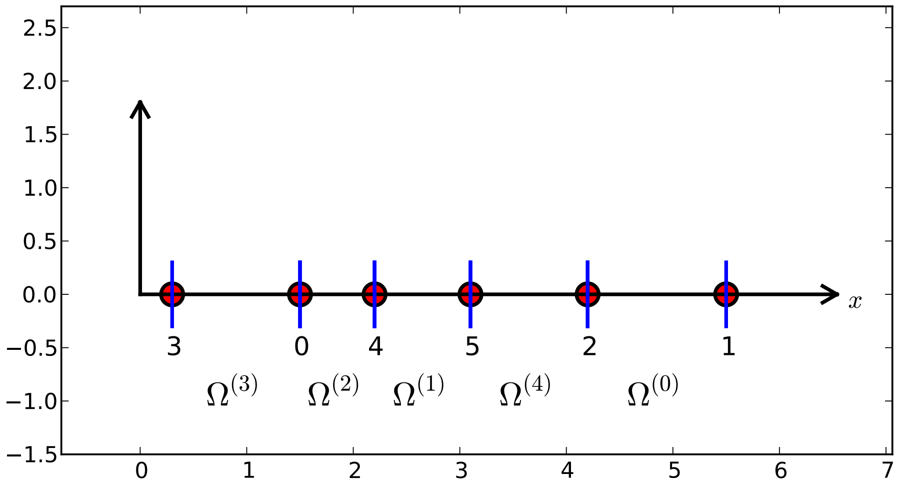


```
d = 3  # d+1 nodes per element
num_elements = 4
num_nodes = num_elements*d + 1
nodes = [i*0.5 for i in range(num_nodes)]
elements = [[i*d+j for j in range(d+1)] for i in range(num_elements)]
```

3.10 Some corresponding basis functions (P3 elements)



3.11 The numbering does not need to be regular from left to right



```
nodes = [1.5, 5.5, 4.2, 0.3, 2.2, 3.1]
elements = [[2, 1], [4, 5], [0, 4], [3, 0], [5, 2]]
```

3.12 Interpretation of the coefficients c_i

Important property: c_i is the value of u at node i, x_i:

$$u(x_i) = \sum_{j\in I} c_j\varphi_j(x_i) = c_i\varphi_i(x_i) = c_i \tag{36}$$

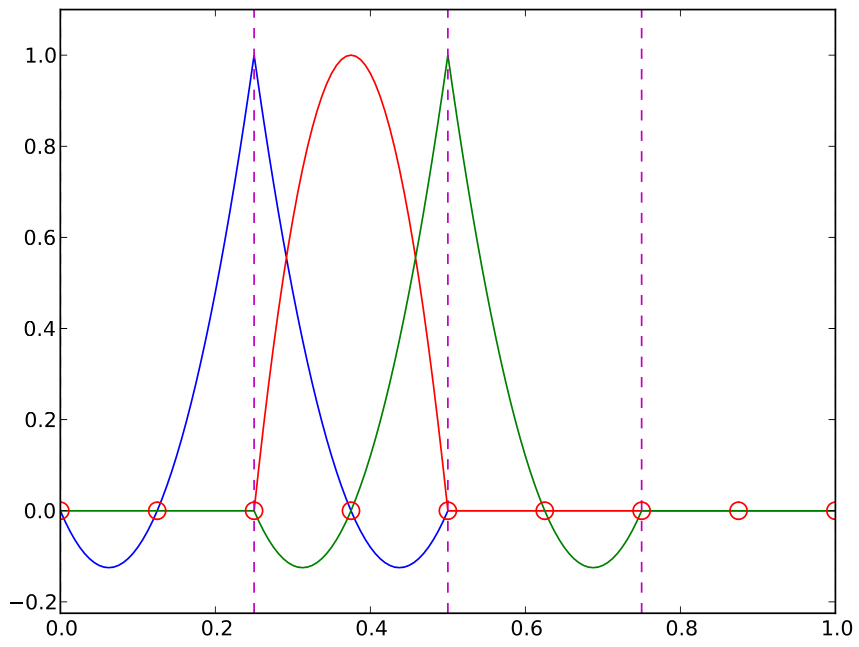
3.13 Properties of the basis functions

$\varphi_i(x)$ 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* (important for implementation!).

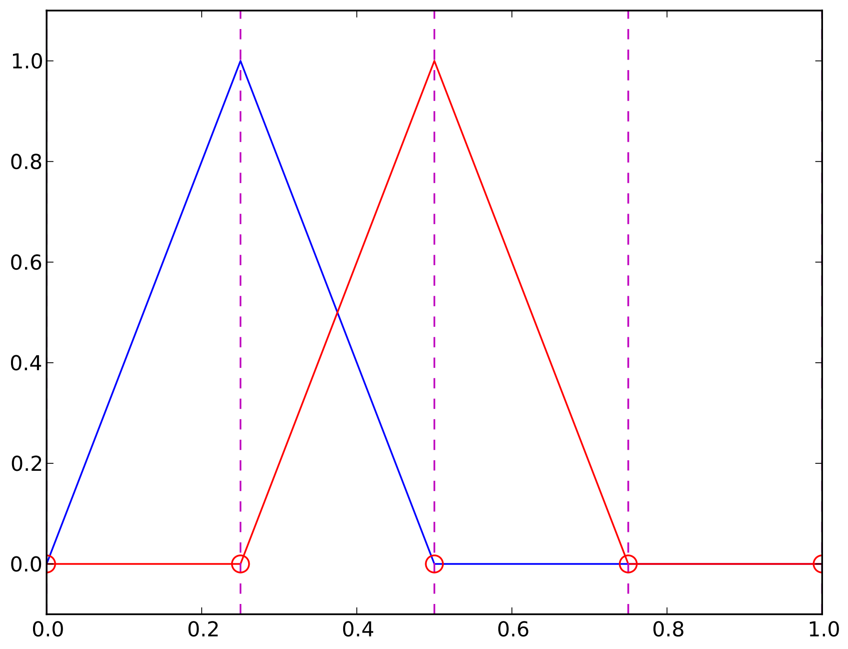
3.14 How to construct quadratic φ_i (P2 elements)



1. Associate Lagrange polynomials with the nodes in an element

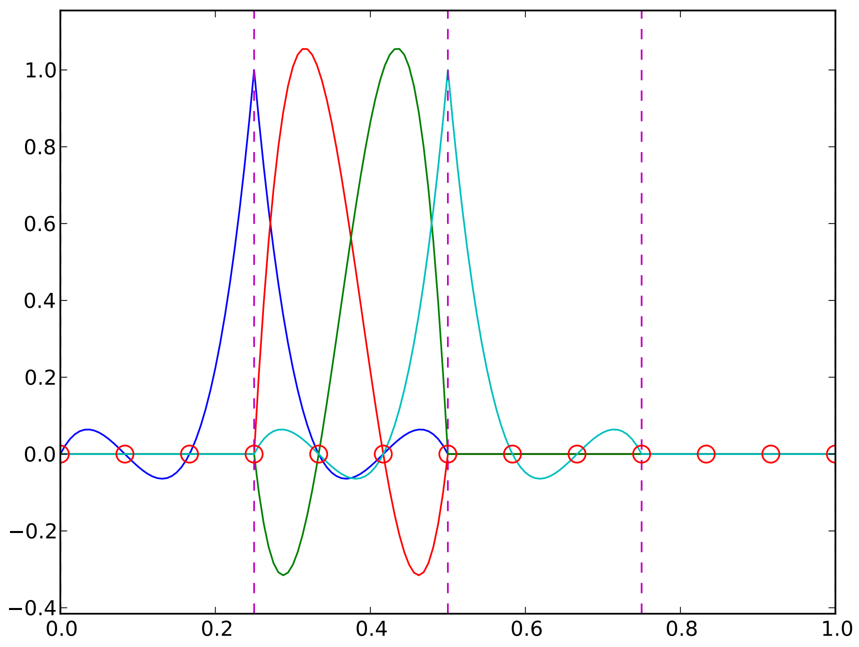
2. When the polynomial is 1 on the element boundary, combine it with the polynomial in the neighboring element

3.15 Example on linear φ_i (P1 elements)



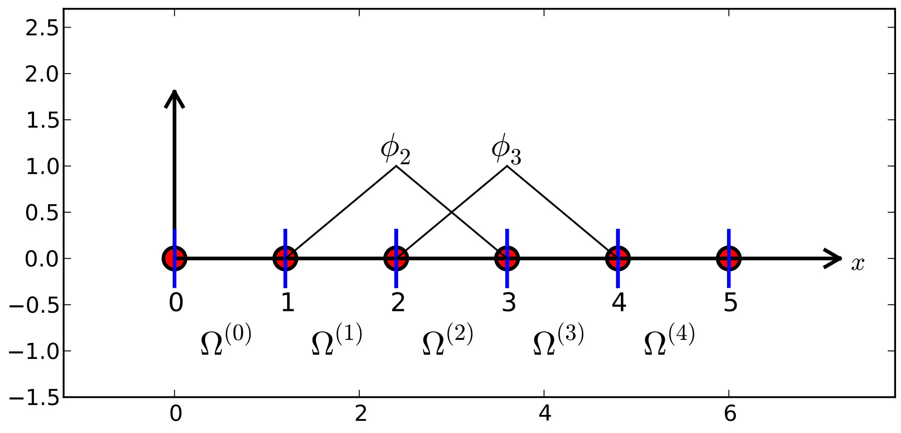
$$\varphi_i(x) = \begin{cases} 0, & x < x_{i-1}, \\ (x - x_{i-1})/h, & x_{i-1} \le x < x_i, \\ 1 - (x - x_i)/h, & x_i \le x < x_{i+1}, \\ 0, & x \ge x_{i+1} \end{cases} \tag{37}$$

3.16 Example on cubic φ_i (P3 elements)



4 Calculating the linear system for c_i

4.1 Computing a specific matrix entry (1)

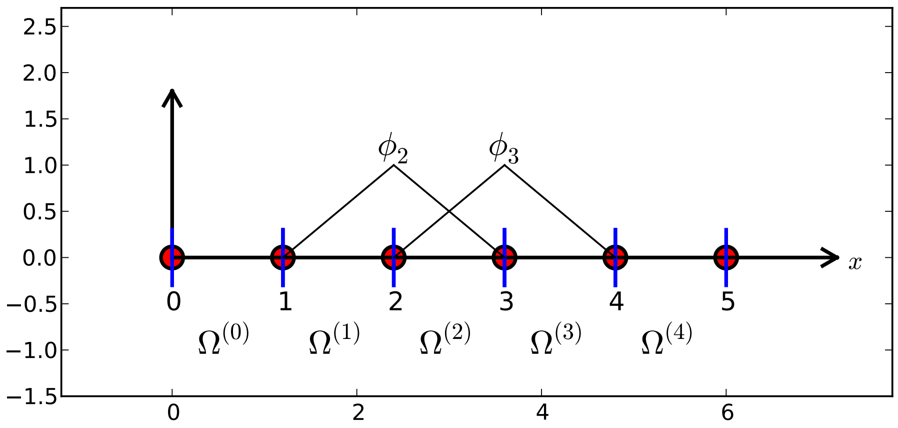


$A_{2,3} = \int_\Omega \varphi_2\varphi_3 dx$: $\varphi_2\varphi_3 \neq 0$ only over element 2. There,

$$\varphi_3(x) = (x - x_2)/h, \quad \varphi_2(x) = 1 - (x - x_2)/h$$

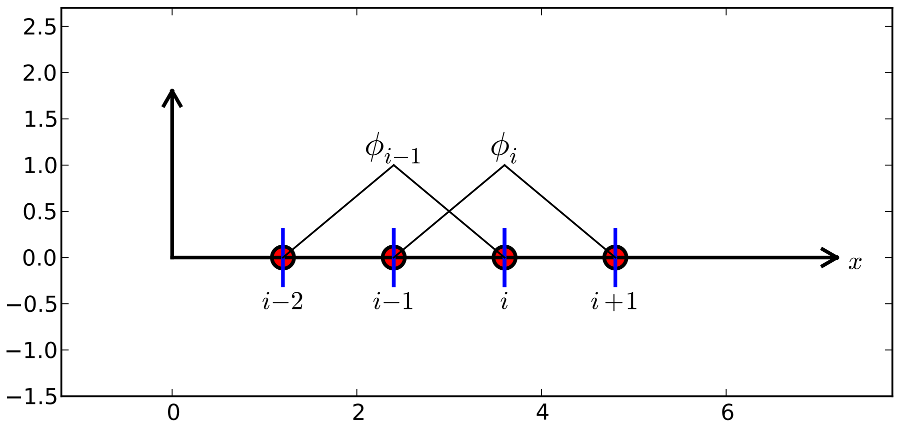
$$A_{2,3} = \int_\Omega \varphi_2\varphi_3 \,\mathrm{d}x = \int_{x_2}^{x_3} \left(1 - \frac{x-x_2}{h}\right)\frac{x-x_2}{h}\,\mathrm{d}x = \frac{h}{6}\,.$$

4.2 Computing a specific matrix entry (2)



$$A_{2,2} = \int_{x_1}^{x_2} \left(\frac{x-x_1}{h}\right)^2 \mathrm{d}x + \int_{x_2}^{x_3} \left(1 - \frac{x-x_2}{h}\right)^2 \mathrm{d}x = \frac{h}{3}\,.$$

4.3 Calculating a general row in the matrix; figure



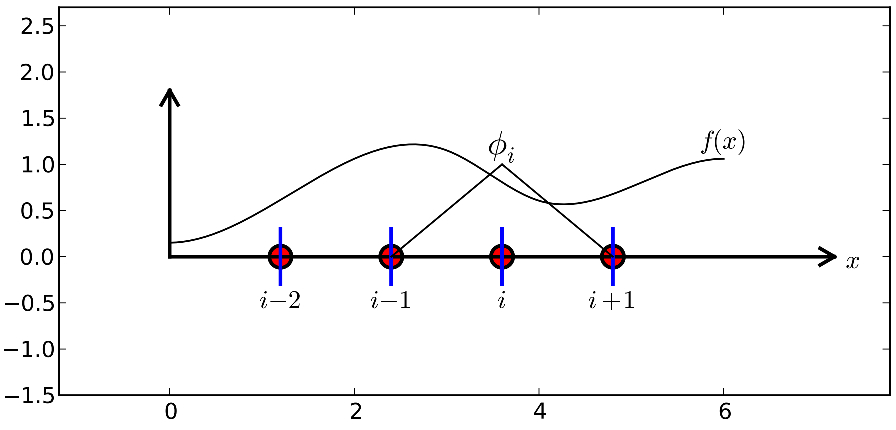
$$A_{i,i-1} = \int_\Omega \varphi_i\varphi_{i-1}\,\mathrm{d}x = ?$$

4.4 Calculating a general row in the matrix; details

$$
\begin{aligned}
A_{i,i-1} &= \int_\Omega \varphi_i\varphi_{i-1}\,\mathrm{d}x \\
&= \underbrace{\int_{x_{i-2}}^{x_{i-1}} \varphi_i\varphi_{i-1}\,\mathrm{d}x}_{\varphi_i=0} + \int_{x_{i-1}}^{x_i} \varphi_i\varphi_{i-1}\,\mathrm{d}x + \underbrace{\int_{x_i}^{x_{i+1}} \varphi_i\varphi_{i-1}\,\mathrm{d}x}_{\varphi_{i-1}=0} \\
&= \int_{x_{i-1}}^{x_i} \underbrace{\frac{x-x_i}{h}}_{\varphi_i(x)} \underbrace{\left(1-\frac{x-x_{i-1}}{h}\right)}_{\varphi_{i-1}(x)} \mathrm{d}x = \frac{h}{6}\,.
\end{aligned}
$$

- $A_{i,i+1} = A_{i,i-1}$ due to symmetry
- $A_{i,i} = h/3$ (same calculation as for $A_{2,2}$)
- $A_{0,0} = A_{N,N} = h/3$ (only one element)

4.5 Calculation of the right-hand side



$$
b_i = \int_\Omega \varphi_i(x) f(x)\,\mathrm{d}x = \int_{x_{i-1}}^{x_i} \frac{x-x_{i-1}}{h} f(x)\,\mathrm{d}x + \int_{x_i}^{x_{i+1}} \left(1-\frac{x-x_i}{h}\right) f(x)\,\mathrm{d}x\,. \qquad (38)
$$

Need a specific $f(x)$ to do more...

4.6 Specific example: two elements; linear system and solution

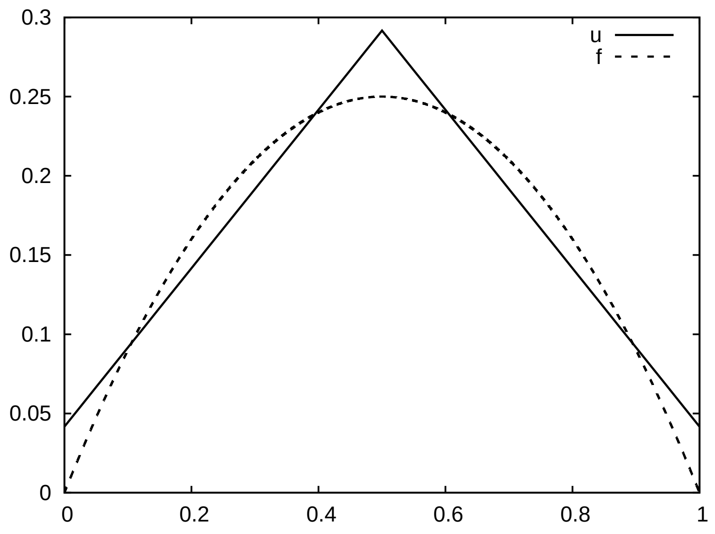
- $f(x) = x(1-x)$ on $\Omega = [0,1]$
- Two equal-sized elements $[0, 0.5]$ and $[0.5, 1]$

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

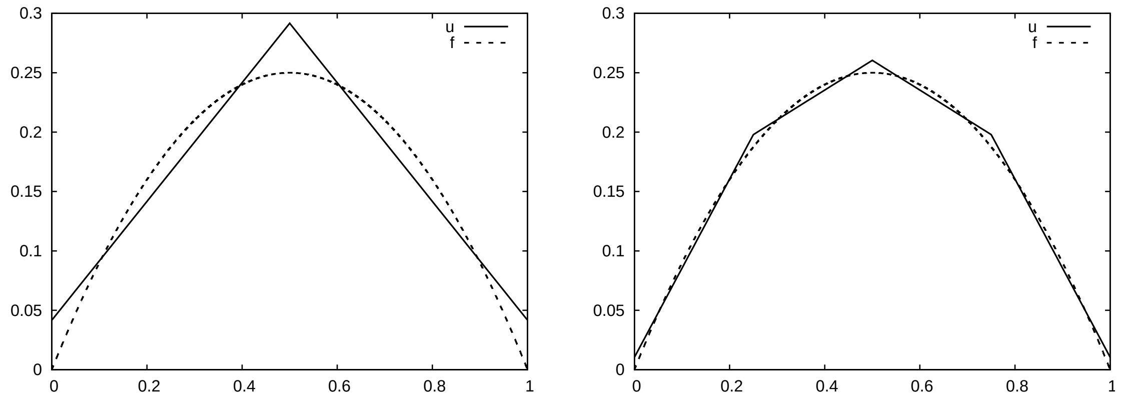
$$
c_0 = \frac{h^2}{6}, \quad c_1 = h - \frac{5}{6}h^2, \quad c_2 = 2h - \frac{23}{6}h^2\,.
$$

4.7 Specific example: two elements; plot

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



4.8 Specific example: what about four elements?



5 Assembly of elementwise computations

5.1 Split the integrals into elementwise integrals

$$A_{i,j} = \int_\Omega \varphi_i\varphi_j dx = \sum_e A^{(e)}_{i,j}, \quad A^{(e)}_{i,j} = \int_{\Omega^{(e)}} \varphi_i\varphi_j dx\,. \tag{39}$$

Important:

- $A^{(e)}_{i,j} \neq 0$ if and only if i and j are nodes in element e (otherwise no overlap between the basis functions)
- all the nonzero elements in $A^{(e)}_{i,j}$ are collected in an *element matrix*

5.2 The element matrix

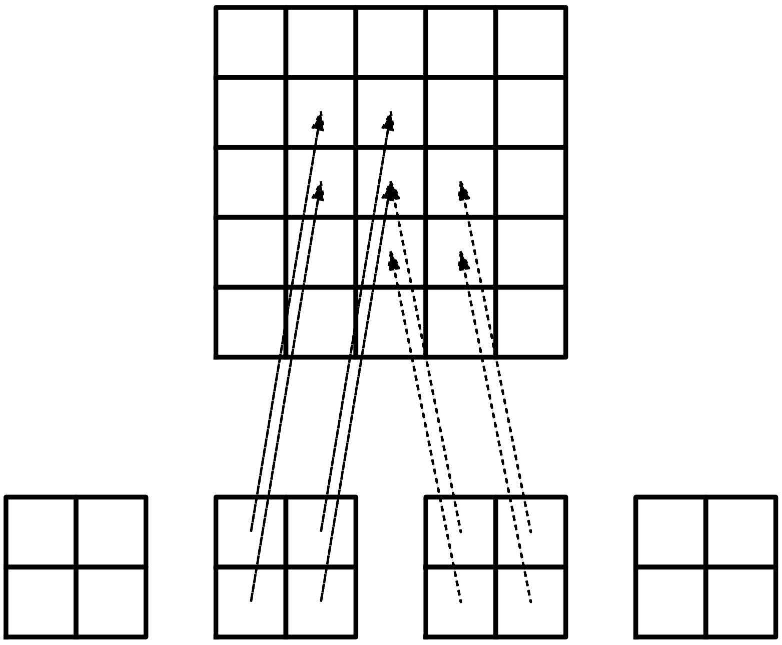
$$\tilde{A}^{(e)} = \{\tilde{A}^{(e)}_{r,s}\},\quad r,s \in I_d = \{0,\ldots,d\},$$

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

- r,s run over *local node numbers* within an element, while i,j run over *global node numbers*.
- $i=q(e,r)$: mapping of local node number r in element e to the global node number i. Math equivalent to `i=elements[e][r]`.
- Add contribution from an element into the global coefficient matrix (*assembly*)

$$A_{q(e,r),q(e,s)} := A_{q(e,r),q(e,s)} + \tilde{A}^{(e)}_{r,s},\quad r,s\in I_d\,. \tag{40}$$

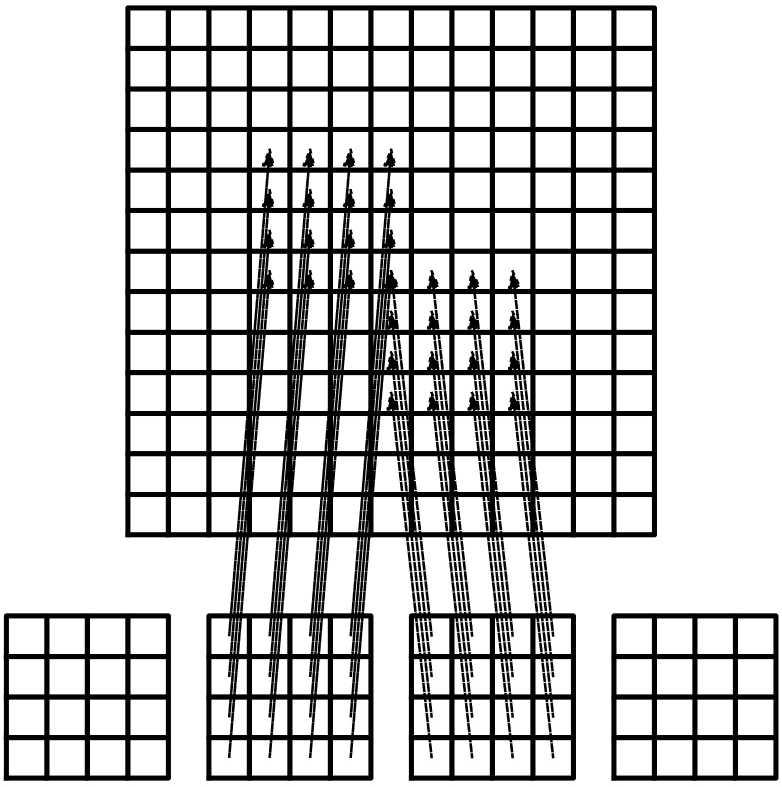
5.3 Illustration of the matrix assembly: regularly numbered P1 elements



Animation[2]

[2]http://tinyurl.com/k3sdbuv/pub/mov-fem/fe_assembly.html

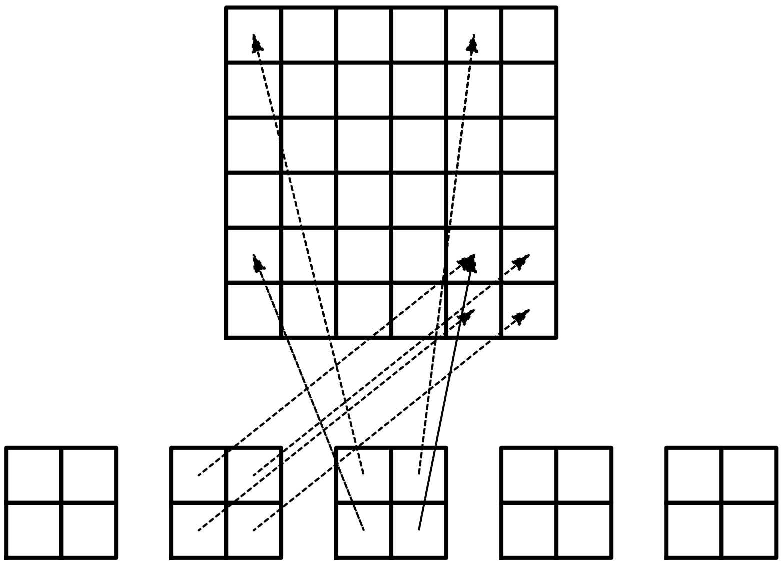
5.4 Illustration of the matrix assembly: regularly numbered P3 elements



Animation[3]

[3]http://tinyurl.com/k3sdbuv/pub/mov-fem/fe_assembly.html

5.5 Illustration of the matrix assembly: irregularly numbered P1 elements



Animation[4]

5.6 Assembly of the right-hand side

Split in elementwise contributions:

$$b_i = \int_\Omega \varphi_i \varphi_j dx = \sum_e b_i^{(e)}, \quad b_i^{(e)} = \int_{\Omega^{(e)}} f(x)\varphi_i(x) dx\,. \tag{41}$$

Important:

- $b_i^{(e)} \neq 0$ if and only if global node i is a node in element e (otherwise $\varphi_i = 0$)
- The $d+1$ nonzero $b_i^{(e)}$ can be collected in an *element vector*

$$\tilde{b}_r^{(e)} = \{\tilde{b}_r^{(e)}\}, \quad r \in I_d\,.$$

Assembly:

$$b_{q(e,r)} := b_{q(e,r)} + \tilde{b}_r^{(e)}, \quad r, s \in I_d\,. \tag{42}$$

6 Mapping to a reference element

Instead of computing

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

over some element $\Omega^{(e)} = [x_L, x_R]$, we now map $[x_L, x_R]$ to a standardized reference element domain $[-1, 1]$ with local coordinate X.

[4]`http://tinyurl.com/k3sdbuv/pub/mov-fem/fe_assembly.html`

6.1 Affine mapping

$$x = \frac{1}{2}(x_L + x_R) + \frac{1}{2}(x_R - x_L)X\,. \tag{43}$$

or rewritten as

$$x = x_m + \frac{1}{2}hX, \qquad x_m = (x_L + x_R)/2 \tag{44}$$

6.2 Integral transformation

Integrating on the reference element is a matter of just changing the integration variable from x to X. Introduce local basis function

$$\tilde{\varphi}_r(X) = \varphi_{q(e,r)}(x(X)) \tag{45}$$

The integral transformation reads

$$\tilde{A}^{(e)}_{r,s} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}(x)\varphi_{q(e,s)}(x)dx = \int_{-1}^{1} \tilde{\varphi}_r(X)\tilde{\varphi}_s(X)\frac{dx}{dX}dX\,. \tag{46}$$

Introduce the notation $\det J = dx/dX = h/2$ (2D/3D must use $\det J$)

$$\tilde{A}^{(e)}_{r,s} = \int_{-1}^{1} \tilde{\varphi}_r(X)\tilde{\varphi}_s(X)\det J\,dX\,. \tag{47}$$

$$\tilde{b}^{(e)}_{r} = \int_{\Omega^{(e)}} f(x)\varphi_{q(e,r)}(x)dx = \int_{-1}^{1} f(x(X))\tilde{\varphi}_r(X)\det J\,dX\,. \tag{48}$$

6.3 Advantages of the reference element

- Always the same domain for integration: $[-1, 1]$
- We only need formulas for $\tilde{\varphi}_r(X)$ on the reference elements (no need for piecewise polynomial definition)
- All geometric information (length and location) is "factored out" in the mapping and $\det J$

6.4 Standardized basis functions for P1 elements

$$\tilde{\varphi}_0(X) = \frac{1}{2}(1 - X) \tag{49}$$

$$\tilde{\varphi}_1(X) = \frac{1}{2}(1 + X) \tag{50}$$

6.5 Standardized basis functions for P2 elements

P2 elements:

$$\tilde{\varphi}_0(X) = \frac{1}{2}(X - 1)X \tag{51}$$

$$\tilde{\varphi}_1(X) = 1 - X^2 \tag{52}$$

$$\tilde{\varphi}_2(X) = \frac{1}{2}(X + 1)X \tag{53}$$

Easy to generalize to arbitrary order!

6.6 Integration over a reference element; element matrix

P1 elements and $f(x) = x(1-x)$.

$$\begin{aligned}
\tilde{A}^{(e)}_{0,0} &= \int_{-1}^{1} \tilde{\varphi}_0(X)\tilde{\varphi}_0(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} \frac{1}{2}(1-X)\frac{1}{2}(1-X)\frac{h}{2}dX = \frac{h}{8}\int_{-1}^{1}(1-X)^2 dX = \frac{h}{3}, \quad (54)\\
\tilde{A}^{(e)}_{1,0} &= \int_{-1}^{1} \tilde{\varphi}_1(X)\tilde{\varphi}_0(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} \frac{1}{2}(1+X)\frac{1}{2}(1-X)\frac{h}{2}dX = \frac{h}{8}\int_{-1}^{1}(1-X^2) dX = \frac{h}{6}, \quad (55)\\
\tilde{A}^{(e)}_{0,1} &= \tilde{A}^{(e)}_{1,0}, \quad (56)\\
\tilde{A}^{(e)}_{1,1} &= \int_{-1}^{1} \tilde{\varphi}_1(X)\tilde{\varphi}_1(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} \frac{1}{2}(1+X)\frac{1}{2}(1+X)\frac{h}{2}dX = \frac{h}{8}\int_{-1}^{1}(1+X)^2 dX = \frac{h}{3}\,. \quad (57)
\end{aligned}$$

6.7 Integration over a reference element; element vector

$$\begin{aligned}
\tilde{b}^{(e)}_0 &= \int_{-1}^{1} f(x(X))\tilde{\varphi}_0(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} (x_m + \frac{1}{2}hX)(1-(x_m+\frac{1}{2}hX))\frac{1}{2}(1-X)\frac{h}{2}dX \\
&= -\frac{1}{24}h^3 + \frac{1}{6}h^2 x_m - \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m \quad (58)\\
\tilde{b}^{(e)}_1 &= \int_{-1}^{1} f(x(X))\tilde{\varphi}_1(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} (x_m + \frac{1}{2}hX)(1-(x_m+\frac{1}{2}hX))\frac{1}{2}(1+X)\frac{h}{2}dX \\
&= -\frac{1}{24}h^3 - \frac{1}{6}h^2 x_m + \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m\,. \quad (59)
\end{aligned}$$

x_m: element midpoint.

6.8 Tedious calculations! Let's use symbolic software

```
>>> 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
```

Can printe out in LaTeX too (convenient for copying into reports):

```
>>> 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}
```

7 Implementation

- Coming functions appear in `fe_approx1D.py`[5]
- Functions can operate in symbolic or numeric mode
- The code documents all steps in finite element calculations!

7.1 Compute finite element basis functions

Let $\tilde{\varphi}_r(X)$ be 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

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

7.2 Compute 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):
```

[5]http://tinyurl.com/jvzzcfn/fem/fe_approx1D.py

```
            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
```

7.3 Example on symbolic and numeric element matrix

```
>>> 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]
```

7.4 Compute the element vector

```
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
```

Note `f.subs('x', x)`: replace `x` by $x(X)$ such that `f` contains `X`

7.5 Fallback on numerical integration if symbolic integration fails

- Element matrix: only polynomials and `sympy` always succeeds
- Element vector: $\int f\tilde{\varphi}\,dx$ can fail (`sympy` then returns an `Integral` object instead of a number)

```
def element_vector(f, phi, Omega_e, symbolic=True):
        ...
        I = sm.integrate(f*phi[r]*detJ, (X, -1, 1))  # try...
        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
        ...
```

7.6 Linear system assembly and solution

```
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
```

7.7 Linear system solution

```
if symbolic:
    c = A.LUsolve(b)           # sympy arrays, symbolic Gaussian elim.
else:
    c = np.linalg.solve(A, b)  # numpy arrays, numerical solve
```

Note: the symbolic computation of `A` and `b` and the symbolic solution can be very tedious.

7.8 Example on computing approximations

```
>>> 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)]
```

Numerical computations:

```
>>> 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]
```

7.9 The structure of the coefficient matrix

```
>>> 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]
```

Note: do this by hand to understand what is going on!

7.10 General result: the coefficient matrix is sparse

- Sparse = most of the entries are zeros
- Below: P1 elements

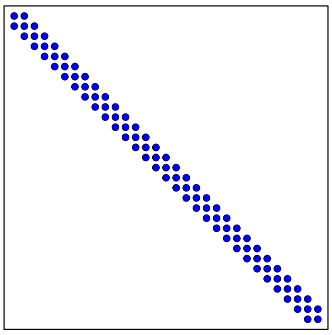
$$A = \frac{h}{6}\left(\begin{array}{cccccccccc}
2 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
1 & 4 & 1 & \ddots & & & & & \vdots \\
0 & 1 & 4 & 1 & \ddots & & & & \vdots \\
\vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\
\vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\
\vdots & & & 0 & 1 & 4 & 1 & \ddots & \vdots \\
\vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\
\vdots & & & & & \ddots & 1 & 4 & 1 \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 & 2
\end{array}\right) \tag{60}$$

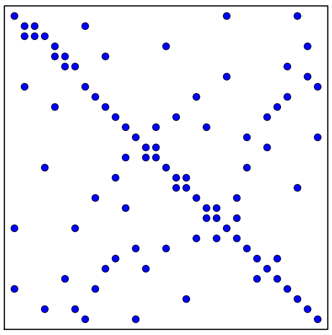
7.11 Exemplifying the sparsity for P2 elements

$$A = \frac{h}{30}\left(\begin{array}{ccccccccc}
4 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0\\
2 & 16 & 2 & 0 & 0 & 0 & 0 & 0 & 0\\
-1 & 2 & 8 & 2 & -1 & 0 & 0 & 0 & 0\\
0 & 0 & 2 & 16 & 2 & 0 & 0 & 0 & 0\\
0 & 0 & -1 & 2 & 8 & 2 & -1 & 0 & 0\\
0 & 0 & 0 & 0 & 2 & 16 & 2 & 0 & 0\\
0 & 0 & 0 & 0 & -1 & 2 & 8 & 2 & -1\\
0 & 0 & 0 & 0 & 0 & 0 & 2 & 16 & 2\\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 & 4
\end{array}\right) \tag{61}$$

7.12 Matrix sparsity pattern for regular/random numbering of P1 elements

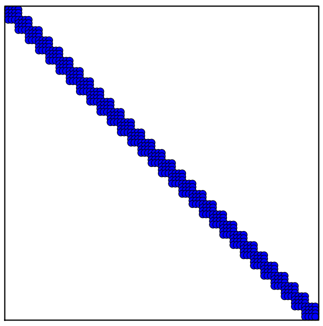
- Left: number nodes and elements from left to right
- Right: number nodes and elements arbitrarily

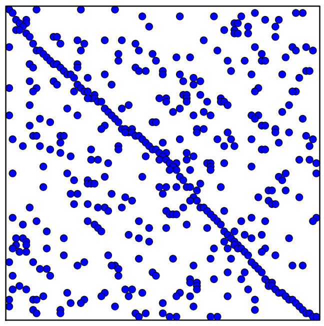




7.13 Matrix sparsity pattern for regular/random numbering of P3 elements

- Left: number nodes and elements from left to right
- Right: number nodes and elements arbitrarily





7.14 Sparse matrix storage and solution

The minimum storage requirements for the coefficient matrix $A_{i,j}$:

- P1 elements: only 3 nonzero entires per row
- P2 elements: only 5 nonzero entires per row
- P2 elements: only 7 nonzero entires per row
- It is important to utilize sparse storage and sparse solvers
- In Python: `scipy.sparse` package

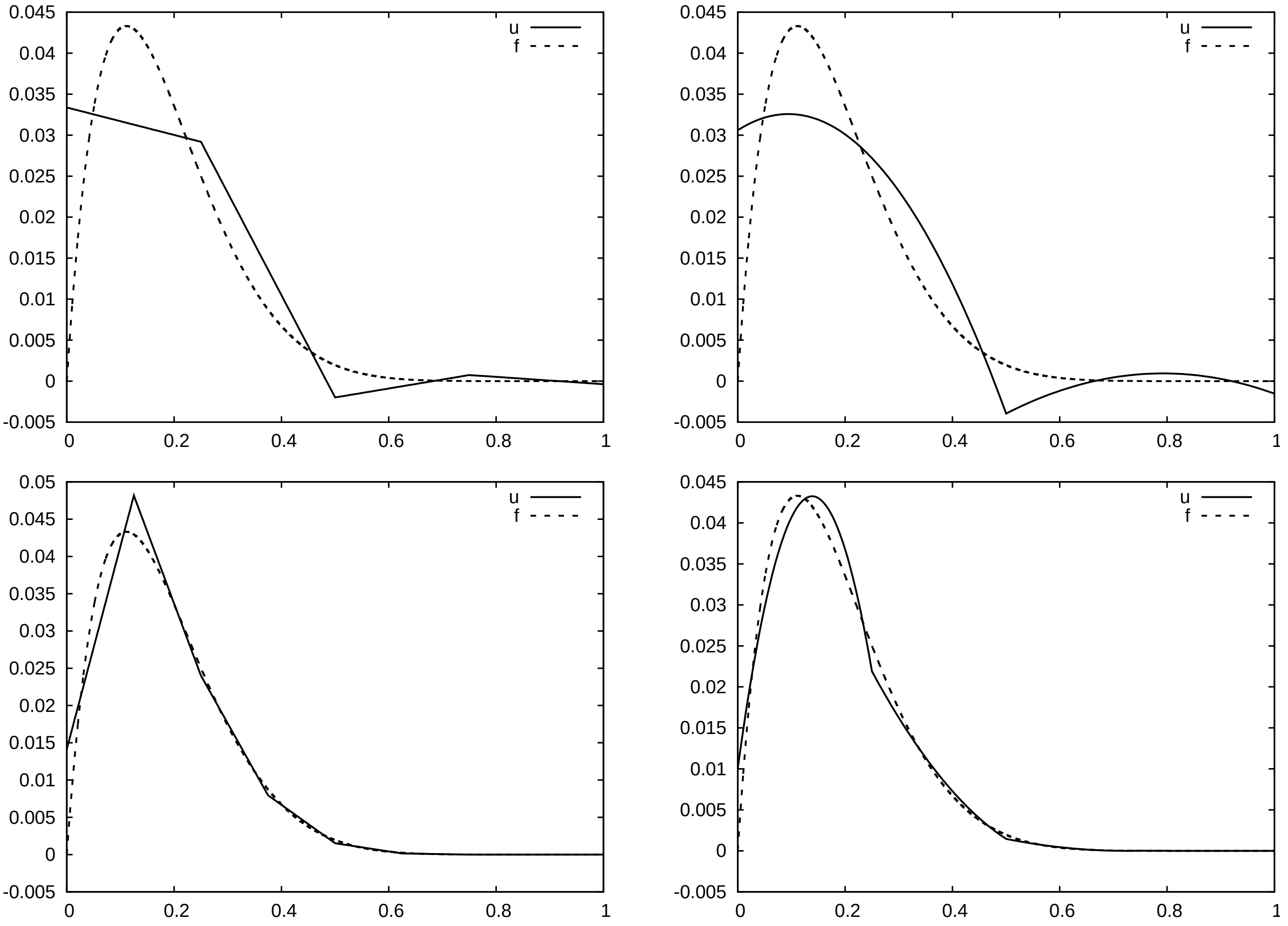
7.15 Approximate $f \sim x^9$ by various elements; code

Compute a mesh with `N_e` elements, basis functions of degree `d`, and approximate a given symbolic expression `f` by a finite element expansion $u(x) = \sum_j c_j \varphi_j(x)$:

```
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)
```

7.16 Approximate $f \sim x^9$ by various elements; plot



8 Comparison of finite element and finite difference approximation

- Finite difference approximation of a function $f(x)$: simply choose $u_i = f(x_i)$ (interpolation)
- Galerkin/projection and least squares method: must derive and solve a linear system
- What is really the difference?

8.1 Interpolation/collocation with finite elements

Let x_i, $i \in I$, be the nodes in the mesh. Collocation means

$$u(x_i) = f(x_i), \quad i \in I, \qquad (62)$$

which translates to

$$\sum_{j\in I} c_j\varphi_j(x_i) = f(x_i),$$

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

$$c_i = f(x_i)\,. \tag{63}$$

- Same result as the standard finite difference approach
- u *interpolates* f at the node points
- u has a variation between the node points dictated by the φ_i functions

Index