
Introduction to Numerical Methods for Variational Problems

Hans Petter Langtangen^{1,2}

Kent-Andre Mardal^{3,1}

¹Center for Biomedical Computing, Simula Research Laboratory

²Department of Informatics, University of Oslo

³Department of Mathematics, University of Oslo

This easy-to-read book introduces the basic ideas and technicalities of least squares, Galerkin, and weighted residual methods for solving partial differential equations. Special emphasis is put on finite element methods.

Aug 8, 2019

Preface

The present book is essentially a book on the finite element method, although we discuss many other choices of basis functions and other applications than partial differential equations. The literature on finite elements contains books of many different flavors, ranging from an abstract view of the method [5, 6, 9, 13, 26] to a more practical, algorithmic treatment of the subject [18, 33]. The present book has a very strong algorithmic focus (“how to compute”), but formulate the method in abstract form with variational forms and function spaces.

One highly valued feature of the finite element method is the prospect of rigorous analysis provided by the functional analysis framework. In particular, for elliptic problems (or, in general, symmetric problems) the theory provides error control via sharp estimates and efficient computations via multiscale algorithms. In fact, the development of this mathematical theory of finite element methods is one of the highlights of numerical analysis. However, within scientific computing the computational engine of the finite element method is being used far outside what is theoretically understood. Obviously, this is a trend that will continue in the future. Hence, it is not our aim to present the established mathematical theory here, but rather provide the reader with the gory details of the implementation in an explicit, user-friendly and simplistic manner to lower the threshold of usage. At the same time, we want to present tools for verification and debugging that are applicable in general situations such that the method can be used safely beyond what is theoretically proven.

An important motivation for writing this book was to provide an intuitive approach to the finite element method, using mathematical language frameworks such as the FEniCS software [23]. FEniCS is a modern, very powerful tool for solving partial differential equations by the finite element method, and it was designed to make implementations very compact, which is attractive for those who are used to the abstract formulation of the method. Our aim here is different. Through explicit, detailed and sometimes lengthy derivations, the reader will be able to get direct exposition of all components in a finite element engine.

A standard reference for FEniCS users has been the excellent text by Brenner and Scott [6], but for many students with weak formal mathematical background the learning curve still becomes too steep. The present book grew out of the need to explain variational formulations in the most intuitive way so FEniCS users can transform their PDE problem into the proper formulation for FEniCS programming. We then added material such that also the details of the most fundamental finite element algorithms could easily be understood.

The learning outcomes of this book are five-fold:

1. understanding various types of variational formulations of PDE problems,
2. understanding the machinery of finite element algorithms, with an emphasis on one-dimensional problems,
3. understanding potential artifacts in simulation results,
4. understanding how variational formulations can be used in other contexts (generalized boundary conditions, solving linear systems)
5. understanding how variational methods may be used for complicated PDEs (systems of non-linear and time-dependent PDEs)

The exposition is recognized by very explicit mathematics, i.e., we have tried to write out all details of the finite element “engine” such that a reader can calculate a finite element problem by hand. Explaining all details and carrying them out by hand are formidable tasks in two- and three-dimensional PDE problems, so we restrict the attention to one space dimension when it comes to detailed calculations. Although we imagine that the reader will use FEniCS or other similar software to actually solve finite element problems, we strongly believe that successful application of such complex software requires a thorough understanding of the underlying method, which is best gained by hand calculations of the steps in the algorithms. Also, hand calculations are indispensable for debugging finite element programs: one can run a one-dimensional

problem, print out intermediate results, and compare with separate hand calculations. When the program is fully verified in 1D, ideally the program should be turned into a 2D/3D simulation simply by switching from a 1D mesh to the relevant 2D/3D mesh.

When working with algorithms and hand calculations in the present book, we emphasize the usefulness of symbolic computing. Our choice is the free SymPy package, which is very easy to use for students and which gives a seamless transition from symbolic to numerical computing. Most of the numerical algorithms in this book are summarized as compact SymPy programs. However, symbolic computing certainly has its limitations, especially when it comes to speed, so their numerical counterparts are usually also developed in the text. The reader should be able to write her own finite element program for one-dimensional problems, but otherwise we have no aim to educate the reader to write fast, state-of-the-art finite element programs!

Another learning outcome (although not needed to be a successful FEniCS user) is to understand how the finite element method is a special case of more general variational approaches to solving equations. We consider approximation in general, solution of PDEs, as well as solving linear systems in a way that hopefully gives the reader an understanding of how seemingly very different numerical methods actually are just variants of a common way of reasoning.

Many common topics found in finite element books are not present in this book. A striking feature is perhaps the presence of the abstract formulation of the finite element method, but without any classical error analysis. The reason is that we have taken a very practical approach to the contents: what does a user need to know to safely apply finite element software? A thorough understanding of the errors is obviously essential, but the classical error analysis of elliptic problems is of limited practical interest for the practitioner, except for the final results regarding convergence rates of different types of finite elements.

In time-dependent problems, on the other hand, a lot of things can go wrong with finite element solutions, but the extensions of the classical finite element error analysis to time dependency quickly meets limitations with respect to explaining typical numerical artifacts. We therefore follow a completely different type of analysis, namely the one often used for finite difference methods: insight through numerical dispersion relations and similar results based on exact discrete solutions via Fourier wave components. Actually, all the analysis of the quality of finite element solutions are in this book done with the aid of techniques for analyzing

finite difference methods, so a knowledge of finite differences is needed. This approach also makes it very easy to compare the two methods, which is frequently done throughout the text.

The mathematical notation in this text makes deviations from the literature trends. Especially books on the abstract formulation of the finite element method often denote the numerical solution by u_h . Our mathematical notation is dictated by the natural notation in a computer code, so if \mathbf{u} is the unknown in the code, we let u be the corresponding quantity in the mathematical description as well. When also the exact solution of the PDE problem is needed, it is usually denoted by u_e . Especially in the chapter on nonlinear problems we introduce notations that are handy in a program and use the same notation in the mathematics such that we achieve as close correspondence as possible between the mathematics and the code.

Contents. The very first chapter starts with a quick overview of how PDE problems are solved by the finite element method. The next four chapters go into deep detail of the algorithms. We employ a successful idea, pursued by Larson and Bengzon [22] in particular, of first treating finite element approximation *before* attacking PDE problems. Chapter 3 explains approximation of functions in function spaces from a general point of view, where finite element basis functions constitute one example to be explored in Chapter 4. The principles of variational formulations constitute the subject of Chapter 5. A lot of details of the finite element machinery are met already in the approximation problem in Chapter 4, so when these topics are familiar together with the variational formulations of Chapter 5, it is easier to put the entire finite element method together in Chapter 6 with variational formulations of PDE problems, boundary conditions, and the necessary finite element computing algorithms. Our experience is that this pedagogical approach greatly simplifies the learning process for students. Chapter 7 explains how time-dependent problems are attacked, primarily by using finite difference discretizations in time. Here we develop the corresponding finite difference schemes and analyze them via Fourier components and numerical dispersion relations. How to set up variational formulations of systems of PDEs, and in particular the way systems are treated in FEniCS, is the topic of Chapter 8. Nonlinear ODE and PDE problems are treated quite comprehensively in Chapter 10. Finally, the applicability of variational thinking appears in a different context in Chapter 11 where we construct iterative methods for linear systems and derive methods in the Conjugate gradient family.

Supplementary materials. All program and data files referred to in this book are available from the book's primary web site: <http://hplgit.github.io/fem-book/doc/web/>.

Oslo, September 2016 Hans Petter Langtangen, Kent-Andre Mardal

It is now a little more than three years since Hans Petter asked me to help him finalize this book. Obviously, I was honored, but also aware that this would be a significant commitment. That said, it did not take long for me to decide. A main reason was that the first book I read about finite element methods, in the late nineties, was Hans Petter's first book about finite elements and the software library Diffpack. His book was different from other finite element books and it was precious to me for many years. In particular, it combined interesting mathematics and skillful programming with detailed examples that one could play with and learn from. The Diffpack library and the book have had widespread influence and many of today's finite element packages are organized around the same principles, FEniCS being one example.

This book is in many ways similar to Hans Petter's first book. It is an introductory text focused on demonstrating the broad, versatile, and rich approach of finite element methods rather than the examples where a rigorous analysis is available. However, a lot has changed since the nineties: There are now many powerful and successful open source packages for finite elements in particular and scientific computing in general that are available. Hence, students today are not expected to develop their codes from scratch, but rather to use and combine the tools out there. The programming environment is also different from what it was in the 90s. Python was an obscure language then, but is now the dominating language in scientific computing which almost all packages provide an interface to. Hans Petter, with his many excellent books on scientific programming in Python, has been important in this transition.

That said, it is our opinion that it still is important to develop codes from scratch in order to learn all the gory details. That is; "programming is understanding" as Kristen Nygaard put it – a favorite quote of both Hans Petter and me. As such, there is a need for teaching material that exposes the internals of a finite element engine and allow for scrutinous investigation in a clean environment. In particular, Hans Petter always wanted to lower the bar for introducing finite elements both by avoiding technical details of implementation as well as avoiding the theoretical issues with Sobolev spaces and functional analysis. This is the purpose of this book.

Acknowledgement Many people have helped with this book over the years as it evolved. Unfortunately, I did not discuss this process with Hans Petter and neither did we discuss the acknowledgement on his behalf.

Johannes Ring and Kristian Hustad have helped with many technical issues regarding DocOnce. I have received comments, advice and corrections from Yapi Achou, Svein Linge, Matthew Moelter, and Murilo Moreira.

Finally, I would like to thank Nancy, Natalie and Niklas for their support and for providing excellent working conditions during the writing of this book.

Oslo, March 2019

Kent-Andre Mardal

Contents

Preface	v
Second Preface	xi
2 Quick overview of the finite element method	1
3 Function approximation by global functions	9
3.1 Approximation of vectors	10
3.1.1 Approximation of planar vectors.....	11
3.1.2 Approximation of general vectors	14
3.2 Approximation principles	18
3.2.1 The least squares method	18
3.2.2 The projection (or Galerkin) method.....	20
3.2.3 Example of linear approximation	20
3.2.4 Implementation of the least squares method.....	21
3.2.5 Perfect approximation	25
3.2.6 The regression method	26
3.3 Orthogonal basis functions	31
3.3.1 Ill-conditioning.....	31
3.3.2 Fourier series	34
3.3.3 Orthogonal basis functions.....	36
3.3.4 Numerical computations.....	38
3.4 Interpolation	39

3.4.1	The interpolation (or collocation) principle	39
3.4.2	Lagrange polynomials	42
3.4.3	Bernstein polynomials	49
3.5	Approximation properties and convergence rates	52
3.6	Approximation of functions in higher dimensions	59
3.6.1	2D basis functions as tensor products of 1D functions	59
3.6.2	Example on polynomial basis in 2D	61
3.6.3	Implementation	65
3.6.4	Extension to 3D	66
3.7	Exercises	67
4	Function approximation by finite elements	99
4.1	Finite element basis functions	99
4.1.1	Elements and nodes	101
4.1.2	The basis functions	103
4.1.3	Example on quadratic finite element functions	105
4.1.4	Example on linear finite element functions	106
4.1.5	Example on cubic finite element functions	108
4.1.6	Calculating the linear system	109
4.1.7	Assembly of elementwise computations	112
4.1.8	Mapping to a reference element	116
4.1.9	Example on integration over a reference element	118
4.2	Implementation	120
4.2.1	Integration	120
4.2.2	Linear system assembly and solution	123
4.2.3	Example on computing symbolic approximations	124
4.2.4	Using interpolation instead of least squares	124
4.2.5	Example on computing numerical approximations	125
4.2.6	The structure of the coefficient matrix	126
4.2.7	Applications	128
4.2.8	Sparse matrix storage and solution	128
4.3	Comparison of finite elements and finite differences	131
4.3.1	Finite difference approximation of given functions	132
4.3.2	Interpretation of a finite element approximation in terms of finite difference operators	132
4.3.3	Making finite elements behave as finite differences	135
4.4	A generalized element concept	136

4.4.1	Cells, vertices, and degrees of freedom	137
4.4.2	Extended finite element concept	137
4.4.3	Implementation	139
4.4.4	Computing the error of the approximation	141
4.4.5	Example on cubic Hermite polynomials	142
4.5	Numerical integration	143
4.5.1	Newton-Cotes rules	144
4.5.2	Gauss-Legendre rules with optimized points	145
4.6	Finite elements in 2D and 3D	145
4.6.1	Basis functions over triangles in the physical domain .	147
4.6.2	Basis functions over triangles in the reference cell . .	148
4.6.3	Affine mapping of the reference cell	151
4.6.4	Isoparametric mapping of the reference cell	151
4.6.5	Computing integrals	153
4.7	Implementation	153
4.7.1	Example of approximation in 2D using FEniCS	154
4.7.2	Refined code with curve plotting	156
4.8	Exercises	159
5	Variational formulations with global basis functions .	189
5.1	Basic principles for approximating differential equations	189
5.1.1	Differential equation models	190
5.1.2	Simple model problems and their solutions	191
5.1.3	Forming the residual	194
5.1.4	The least squares method	195
5.1.5	The Galerkin method	195
5.1.6	The method of weighted residuals	196
5.1.7	The method of weighted residual and the truncation error	197
5.1.8	Test and trial functions	198
5.1.9	The collocation method	198
5.1.10	Examples on using the principles	200
5.1.11	Integration by parts	204
5.1.12	Boundary function	206
5.2	Computing with global polynomials	207
5.2.1	Computing with Dirichlet and Neumann conditions .	208
5.2.2	When the numerical method is exact	210

5.2.3	Abstract notation for variational formulations	211
5.2.4	Variational problems and minimization of functionals.	212
5.3	Examples on variational formulations	215
5.3.1	Variable coefficient	215
5.3.2	First-order derivative in the equation and boundary condition	217
5.3.3	Nonlinear coefficient	219
5.4	Implementation of the algorithms	220
5.4.1	Extensions of the code for approximation	220
5.4.2	Fallback to numerical methods	221
5.4.3	Example with constant right-hand side	222
5.5	Approximations may fail: convection-diffusion	224
5.6	Exercises	230
6	Variational formulations with finite elements	249
6.1	Computing with finite elements	249
6.1.1	Finite element mesh and basis functions	249
6.1.2	Computation in the global physical domain	251
6.1.3	Comparison with a finite difference discretization	253
6.1.4	Cellwise computations	254
6.2	Boundary conditions: specified nonzero value	257
6.2.1	General construction of a boundary function	257
6.2.2	Example on computing with a finite element-based boundary function	259
6.2.3	Modification of the linear system	262
6.2.4	Symmetric modification of the linear system	265
6.2.5	Modification of the element matrix and vector	266
6.3	Boundary conditions: specified derivative	267
6.3.1	The variational formulation	267
6.3.2	Boundary term vanishes because of the test functions	267
6.3.3	Boundary term vanishes because of linear system modifications	268
6.3.4	Direct computation of the global linear system	269
6.3.5	Cellwise computations	270
6.4	Implementation of finite element algorithms	271
6.4.1	Extensions of the code for approximation	271

6.4.2 Utilizing a sparse matrix	275
6.4.3 Application to our model problem	276
6.5 Variational formulations in 2D and 3D	278
6.5.1 Integration by parts	278
6.5.2 Example on a multi-dimensional variational problem	279
6.5.3 Transformation to a reference cell in 2D and 3D	281
6.5.4 Numerical integration	283
6.5.5 Convenient formulas for P1 elements in 2D	284
6.5.6 A glimpse of the mathematical theory of the finite element method	285
6.6 Implementation in 2D and 3D via FEniCS	290
6.6.1 Mathematical problem	290
6.6.2 Variational formulation	291
6.6.3 The FEniCS solver	293
6.6.4 Making the mesh	295
6.6.5 Solving a problem	298
6.7 Convection-diffusion and Petrov-Galerkin methods	299
6.8 Summary	305
6.9 Exercises	306
7 Time-dependent variational forms	331
7.1 Discretization in time by a Forward Euler scheme	332
7.1.1 Time discretization	333
7.1.2 Space discretization	334
7.1.3 Variational forms	334
7.1.4 Notation for the solution at recent time levels	335
7.1.5 Deriving the linear systems	336
7.1.6 Computational algorithm	337
7.1.7 Example using cosinusoidal basis functions	338
7.1.8 Comparing P1 elements with the finite difference method	340
7.2 Discretization in time by a Backward Euler scheme	341
7.2.1 Time discretization	341
7.2.2 Variational forms	342
7.2.3 Linear systems	342
7.3 Dirichlet boundary conditions	343

7.3.1	Boundary function	344
7.3.2	Finite element basis functions	344
7.3.3	Modification of the linear system	345
7.3.4	Example: Oscillating Dirichlet boundary condition	346
7.4	Accuracy of the finite element solution	348
7.4.1	Methods of analysis	348
7.4.2	Fourier components and dispersion relations	350
7.4.3	Forward Euler discretization	351
7.4.4	Backward Euler discretization	352
7.4.5	Comparing amplification factors	353
7.5	Exercises	359
8	Variational forms for systems of PDEs	361
8.1	Variational forms	361
8.1.1	Sequence of scalar PDEs formulation	362
8.1.2	Vector PDE formulation	362
8.2	A worked example	363
8.3	Identical function spaces for the unknowns	364
8.3.1	Variational form of each individual PDE	364
8.3.2	Compound scalar variational form	365
8.3.3	Decoupled linear systems	366
8.3.4	Coupled linear systems	367
8.4	Different function spaces for the unknowns	369
8.5	Computations in 1D	371
8.5.1	Another example in 1D	375
8.6	Exercises	384
9	Flexible implementations of boundary conditions	385
9.1	Optimization with constraint	385
9.1.1	Elimination of variables	386
9.1.2	Lagrange multiplier method	386
9.1.3	Penalty method	387
9.2	Optimization of functionals	388
9.2.1	Classical calculus of variations	389
9.2.2	Penalty and Nitsche's methods for optimization with constraints	391

9.2.3 Lagrange multiplier method for optimization with constraints	394
9.2.4 Example: 1D problem	396
9.2.5 Example: adding a constraint in a Neumann problem ..	398
10 Nonlinear problems	405
10.1 Introduction of basic concepts	405
10.1.1 Linear versus nonlinear equations	405
10.1.2 A simple model problem	407
10.1.3 Linearization by explicit time discretization	408
10.1.4 Exact solution of nonlinear algebraic equations	409
10.1.5 Linearization	410
10.1.6 Picard iteration	411
10.1.7 Linearization by a geometric mean	413
10.1.8 Newton's method	415
10.1.9 Relaxation	416
10.1.10 Implementation and experiments	417
10.1.11 Generalization to a general nonlinear ODE	419
10.1.12 Systems of ODEs	422
10.2 Systems of nonlinear algebraic equations	425
10.2.1 Picard iteration	425
10.2.2 Newton's method	426
10.2.3 Stopping criteria	428
10.2.4 Example: A nonlinear ODE model from epidemiology	429
10.3 Linearization at the differential equation level	431
10.3.1 Explicit time integration	432
10.3.2 Backward Euler scheme and Picard iteration	432
10.3.3 Backward Euler scheme and Newton's method	433
10.3.4 Crank-Nicolson discretization	436
10.4 1D stationary nonlinear differential equations	437
10.4.1 Finite difference discretization	438
10.4.2 Solution of algebraic equations	440
10.4.3 Galerkin-type discretization	445
10.4.4 Picard iteration defined from the variational form	446
10.4.5 Newton's method defined from the variational form ..	447
10.5 Multi-dimensional PDE problems	449
10.5.1 Finite element discretization	450
10.5.2 Finite difference discretization	453

10.5.3 Continuation methods	456
10.6 Symbolic nonlinear finite element equations	457
10.6.1 Finite element basis functions	457
10.6.2 The group finite element method	458
10.6.3 Numerical integration of nonlinear terms by hand	461
10.6.4 Discretization of a variable coefficient Laplace term	462
10.7 Exercises	465
11 Variational methods for linear systems	485
11.1 Conjugate gradient-like iterative methods	486
11.1.1 The Galerkin method	486
11.1.2 The least squares method	487
11.1.3 Krylov subspaces	487
11.1.4 Computation of the basis vectors	487
11.1.5 Computation of a new solution vector	489
11.1.6 Summary of the least squares method	490
11.1.7 Truncation and restart	490
11.1.8 Summary of the Galerkin method	491
11.1.9 A framework based on the error	492
11.2 Preconditioning	493
11.2.1 Motivation and Basic Principles	493
11.2.2 Use of the preconditioning matrix in the iterative methods	494
11.2.3 Classical iterative methods as preconditioners	495
11.2.4 Incomplete factorization preconditioners	496
11.2.5 Preconditioners developed for solving PDE problems	497
A Useful formulas	499
A.1 Finite difference operator notation	499
A.2 Truncation errors of finite difference approximations	500
A.3 Finite differences of exponential functions	501
A.4 Finite differences of t^n	502
A.4.1 Software	503
References	505
Index	509

List of Exercises and Problems

Problem 3.1: Linear algebra refresher	67
Problem 3.2: Approximate a three-dimensional vector in a plane .	71
Problem 3.3: Approximate a parabola by a sine	72
Problem 3.4: Approximate the exponential function by power functions	75
Problem 3.5: Approximate the sine function by power functions .	76
Problem 3.6: Approximate a steep function by sines.....	80
Problem 3.7: Approximate a steep function by sines with boundary adjustment	83
Exercise 3.8: Fourier series as a least squares approximation.....	86
Problem 3.9: Approximate a steep function by Lagrange polynomials	91
Problem 3.10: Approximate a steep function by Lagrange polynomials and regression	95
Problem 4.1: Define nodes and elements	159
Problem 4.2: Define vertices, cells, and dof maps	160
Problem 4.3: Construct matrix sparsity patterns.....	162
Problem 4.4: Perform symbolic finite element computations	163
Problem 4.5: Approximate a steep function by P1 and P2 elements	165
Problem 4.6: Approximate a steep function by P3 and P4 elements	166
Exercise 4.7: Investigate the approximation error in finite elements	168
Problem 4.8: Approximate a step function by finite elements	171
Exercise 4.9: 2D approximation with orthogonal functions	178
Exercise 4.10: Use the Trapezoidal rule and P1 elements	182
Exercise 4.11: Compare P1 elements and interpolation	183

Exercise 4.12: Implement 3D computations with global basis functions	184
Exercise 4.13: Use Simpson's rule and P2 elements	186
Exercise 4.14: Make a 3D code for Lagrange elements of arbitrary order	187
Exercise 5.1: Refactor functions into a more general class	230
Exercise 5.2: Compute the deflection of a cable with sine functions	232
Exercise 5.3: Compute the deflection of a cable with power functions	245
Exercise 5.4: Check integration by parts	248
Exercise 6.1: Compute the deflection of a cable with 2 P1 elements	306
Exercise 6.2: Compute the deflection of a cable with 1 P2 element	308
Exercise 6.3: Compute the deflection of a cable with a step load	310
Exercise 6.4: Compute with a non-uniform mesh	316
Problem 6.5: Solve a 1D finite element problem by hand	318
Exercise 6.6: Investigate exact finite element solutions	319
Exercise 6.7: Compare finite elements and differences for a radially symmetric Poisson equation	326
Exercise 6.8: Compute with variable coefficients and P1 elements by hand	327
Exercise 6.9: Solve a 2D Poisson equation using polynomials and sines	328
Exercise 6.10: Solve a 3D Laplace problem with FEniCS	328
Exercise 6.11: Solve a 1D Laplace problem with FEniCS	329
Exercise 7.1: Analyze a Crank-Nicolson scheme for the diffusion equation	359
Problem 8.1: Estimate order of convergence for the Cooling law	384
Problem 8.2: Estimate order of convergence for the Cooling law	384
Problem 10.1: Determine if equations are nonlinear or not	465
Exercise 10.2: Derive and investigate a generalized logistic model	466
Problem 10.3: Experience the behavior of Newton's method	473
Problem 10.4: Compute the Jacobian of a 2×2 system	474
Problem 10.5: Solve nonlinear equations arising from a vibration ODE	475
Exercise 10.6: Find the truncation error of arithmetic mean of products	475
Problem 10.7: Newton's method for linear problems	477
Exercise 10.8: Discretize a 1D problem with a nonlinear coefficient	477
Exercise 10.9: Linearize a 1D problem with a nonlinear coefficient	477
Problem 10.10: Finite differences for the 1D Bratu problem	478

Problem 10.11: Integrate functions of finite element expansions	478
Problem 10.12: Finite elements for the 1D Bratu problem.....	480
Exercise 10.13: Discretize a nonlinear 1D heat conduction PDE by finite differences.....	480
Exercise 10.14: Use different symbols for different approximations of the solution	481
Exercise 10.15: Derive Picard and Newton systems from a variational form	481
Exercise 10.16: Derive algebraic equations for nonlinear 1D heat conduction	482
Exercise 10.17: Differentiate a highly nonlinear term	482
Exercise 10.18: Crank-Nicolson for a nonlinear 3D diffusion equation	483
Exercise 10.19: Find the sparsity of the Jacobian	483
Problem 10.20: Investigate a 1D problem with a continuation method	483

Quick overview of the finite element method

2

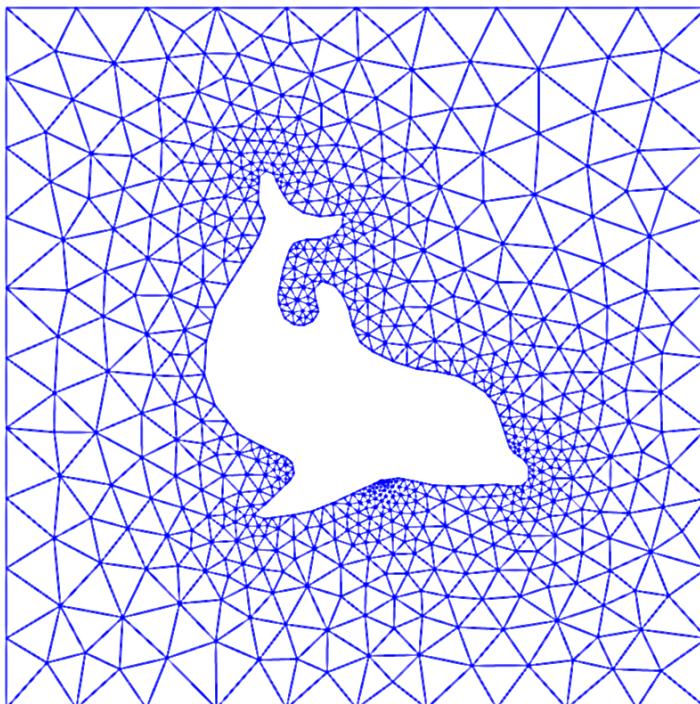


Fig. 2.1 Example on a complicated domain for solving PDEs.

The finite element method is a rich and versatile approach to construct computational schemes to solve any partial differential equation on any domain in any dimension. The method may at first glance appear cumbersome and even unnatural as it relies on variational formulations and polynomial spaces. However, the study of these somewhat abstract concepts pays off as the finite element method provides a general recipe for efficient and accurate simulations.

Let us start by outlining the concepts briefly. Consider the following PDE in 2D:

$$-\nabla^2 u = -u_{xx} - u_{yy} = f,$$

equipped with suitable boundary conditions. A finite difference scheme to solve the current PDE would in the simplest case be described by the stencil

$$-\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} - \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} = f_i \quad (2.1)$$

or reordered to the more recognizable

$$\frac{-u_{i-1,j} - u_{i,j-1} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1}}{h^2} = f_i. \quad (2.2)$$

On a structured mesh, the stencil appears natural and is convenient to implement. However, for a unstructured, “complicated” domain as shown in Figure 2.1, we would need to be careful when placing points and evaluating stencils and functions. In particular, it will be difficult to evaluate the stencil near the dolphin in 2.1 because some points will be on the inside and some outside on the outside of the dolphin. Both accuracy and efficiency may easily be sacrificed by a reckless implementation.

In general, a domain like the one represented in Figure 2.1 will be represented by a triangulation. The finite element method (and the finite volume method which often is a special case of the finite element method) is a methodology for creating stencils like (2.2) in a structured manner that adapt to the underlying triangulation.

The triangulation in Figure 2.1 is a mesh that consists of cells that are connected and defined in terms of vertices. The fundamental idea of the finite element method is to construct a procedure to compute a stencil on a general element and then apply this procedure to each element of the mesh. Let us therefore denote the mesh as Ω while Ω_e is the domain of a generic element such that $\Omega = \cup_e \Omega_e$.

This is exactly the point where the challenges of the finite element method start and where we need some new concepts. The basic question is: How should we create a stencil like (2.2) for a general element and a general PDE that has the maximal accuracy and minimal computational complexity at the current triangulation? The two basic building blocks of the finite element method are

1. the solution is represented in terms of a polynomial expression on the given general element, and
2. a variational formulation of the PDE where element-wise integration enables the PDE to be transformed to a stencil.

Step 1 is, as will be explained later, conveniently represented both implementation-wise and mathematically as a solution

$$u = \sum_{i=0}^N c_i \psi_i(x, y), \quad (2.3)$$

where $\{c_i\}$ are the coefficients to be determined (often called the degrees of freedom) and $\psi_i(x, y)$ are prescribed polynomials. The basis functions $\psi_i(x, y)$ used to express the solution is often called the trial functions. The next step is the variational formulation. This step may seem like a magic trick or a cumbersome mathematical exercise at first glance. We take the PDE and multiply by a function v (usually called the test function) and integrate over an element Ω_e and obtain the expression

$$\int_{\Omega_e} -\nabla^2 u v \, dx = \int_{\Omega_e} f v \, dx \quad (2.4)$$

A perfectly natural question at this point is: Why multiply with a test function v ? The simple answer is that there are $N + 1$ unknowns that need to be determined in u in (2.3) and for this we need $N + 1$ equations. The equations are obtained by using $N + 1$ different test functions which when used in (2.5) give rise to $N + 1$ linearly independent equations.

While (2.4) is a variational formulation of our PDE problem, it is not the most common form. It is common to re-write

$$\int_{\Omega_e} -\nabla^2 u v \, dx \quad (2.5)$$

to weaken the requirement of the polynomial space used for the trial functions (that here needs to be twice differentiable) and write this term in its corresponding weak form. That is, the term is rewritten in terms

of first-derivatives only (of both the trial and the test function) with the aid of Gauss-Green's lemma:

$$\int_{\Omega_e} -\nabla^2 u v \, dx = \int_{\Omega_e} \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega_e} \frac{\partial u}{\partial n} v \, dS \quad (2.6)$$

The reasons behind this alternative formulation are rather mathematical and will not be a major subject of this book as they are well described elsewhere. In fact, a precise explanation would need tools from functional analysis.

With the above rewrite and assuming now that the boundary term vanishes due to boundary conditions (why this is possible will be dealt with in detail later in this book) the stencil, corresponding to (2.2), is represented by

$$\int_{\Omega_e} \nabla u \cdot \nabla v \, dx$$

where u is called the *trial function*, v is called a *test function*, and Ω is an element of a triangulated mesh. The idea of software like FEniCS is that this piece of mathematics can be directly expressed in terms of Python code as

```
mesh = Mesh("some_file")
V = FunctionSpace(mesh, "some polynomial")
u = TrialFunction(V)
v = TestFunction(V)
a = dot(grad(u), grad(v))*dx
```

The methodology and code in this example is not tied to a particular equation, except the formula for a , holding the derivatives of our sample PDE, but any other PDE terms could be expressed via u , v , grad , and other symbolic operators in this line of code. In fact, finite element packages like FEniCS are typically structured as general toolboxes that can be adapted to any PDE as soon as the derivation of variational formulations is mastered. The main obstacle here for a novice FEM user is then to understand the concept of trial functions and test functions realized in terms of polynomial spaces.

Hence, a finite element formulation (or a weak formulation) of the Poisson problem that works on any mesh Ω can be written in terms of solving the problem:

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} f v \, dx .$$

By varying the trial and test spaces we obtain different stencils, some of which will be identical to finite difference schemes on particular meshes. We will now show a complete FEniCS program to illustrate how a typical finite element code may be structured

```
mesh = Mesh("some_file")
V = FunctionSpace(mesh, "some polynomial")
u = TrialFunction(V)
v = TestFunction(V)
a = dot(grad(u), grad(v))*dx
L = f*v*dx

bc = DirichletBC(V, "some_function", "some_domain")
solution = Function(V) # unknown FEM function
solve(a == L, solution, bc)
plot(solution)
```

While the finite element method is versatile and may be adapted to any PDE on any domain in any dimension, the different methods that are derived by using different trial and test functions may vary significantly in terms of accuracy and efficiency. In fact, a bad choice of polynomial space may in some cases lead to a completely wrong result. This is particularly the case for complicated PDEs. For this reason, it is dangerous to regard the method as a black box and not do proper verification of the method for a particular application.

In this book we will put focus on verification in the sense that we provide the reader with explicit calculations as well as demonstrations of how such computations can be performed by using symbolic or numerical calculations to control the various parts of the computational framework. In our view, there are three important tests that should be frequently employed during verification:

1. reducing the model problem to 1D and carefully check the calculations involved in the variational formulation on a small 1D mesh
2. perform the calculation involved on one general or random element
3. test whether convergences is obtained and to what order the method converge by refining the mesh

The two first tasks here should ideally be performed by independent calculations outside the framework used for the simulations. In our view `sympy` is a convenient tool that can be used to assist hand calculations.

So far, we have outlined how the finite element method handles derivatives in a PDE, but we also had a right-hand side function f . This term

is multiplied by the test function v as well, such that the entire Poisson equation is transformed to

$$\int_{\Omega} \nabla u \cdot \nabla v d\Omega = \int_{\Omega} f v d\Omega.$$

This statement is assumed valid for all test functions v in some function space V of polynomials. The right-hand side expression is coded in FEniCS as

```
L = f*v*dx
```

and the problem is then solved by the statements

```
u = Function(V) # unknown FEM function
solve(a == L, u, bc)
```

where `bc` holds information about boundary conditions. This information is connected to information about the triangulation, the *mesh*. Assuming $u = 0$ on the boundary, we can in FEniCS generate a triangular mesh over a rectangular domain $[-1, -1] \times [-1, 1]$ as follows:

```
mesh = RectangleMesh(Point(-1, -1), Point(1, 1), 10, 10)
bc = DirichletBC(V, 0, 'on_boundary')
```

Mathematically, the finite element method transforms our PDE to a sparse linear system. The `solve` step performs two tasks: construction of the linear system based on the given information about the domain and its elements, and then solution of the linear system by either an iterative or direct method.

We are now in a position to summarize all the parts of a FEniCS program that solves the Poisson equation by the finite element method:

```
from fenics import *
mesh = RectangleMesh(Point(-1, -1), Point(1, 1), 10, 10)
V = FunctionSpace(mesh, 'P', 2) # quadratic polynomials
bc = DirichletBC(V, 0, 'on_boundary')
u = TrialFunction(V)
v = TestFunction(V)
a = dot(grad(u), grad(v))*dx
L = f*v*dx
u = Function(V) # unknown FEM function to be computed
solve(a == L, u, bc)
vtkfile = File('poisson.pvd'); vtkfile << u # store solution
```

Solving a different PDE is a matter of changing `a` and `L`. We refer to the FEniCS tutorial [20, 19] for lots of examples.

Although we assert here that the finite element method is a tool that can solve any PDE problem on any domain of any complexity, the

fundamental ideas of the method are in fact even more general. We will therefore start the book by variational methods for approximation in general, then consider the finite element in a wide range of applications, and finally we end up with a short description of how the solution of linear systems also fit into this framework.

Function approximation by global functions

3

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

$$u(x) \approx \sum_{i=0}^N c_i \psi_i(x), \quad (3.1)$$

where $\psi_i(x)$ are prescribed functions and c_0, \dots, c_N are unknown coefficients to be determined. Solution methods for differential equations utilizing (3.1) must have a *principle* for constructing $N + 1$ equations to determine c_0, \dots, c_N . Then there is a *machinery* regarding the actual construction of the equations for c_0, \dots, c_N , in a particular problem. Finally, there is a *solve* phase for computing the solution c_0, \dots, 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 follow an idea of Larson and Bengzon [22] and introduce the computational machinery for a trivial equation: $u = f$. Solving this equation with f given and u of the form (3.1), means that we seek an approximation u to f . This approximation problem has the advantage of introducing most of the finite element toolbox, but without involving 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. That is, the functions are in general nonzero on the entire domain. Third, we introduce the finite element type of basis functions globally. These basis functions will later, in 4, be used with local support (meaning that each function is nonzero except in a small part of the domain) to enhance stability and efficiency. We explain all the details of the computational algorithms involving such functions. Four types of approximation principles are covered: 1) the least squares method, 2) the L_2 projection or Galerkin method, 3) interpolation or collocation, and 4) the regression method.

3.1 Approximation of vectors

We shall start by introducing two fundamental methods for determining the coefficients c_i in (3.1). These methods will be introduced for approximation of vectors. Using vectors in vector spaces to bring across the ideas is believed to be more intuitive to the reader than starting directly 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 $f - u$, measured in a certain norm, is minimized. That is, we aim at finding the best approximation u to f , with the given norm as measure of “distance”. The second method is not as intuitive: we find u such that the error $f - u$ is orthogonal to the space where u lies. This is known as *projection*, or in the context of differential equations, the idea is also well known as *Galerkin’s method*. When approximating vectors and functions, the two methods are equivalent, but this is no longer the case when applying the principles to differential equations.

3.1.1 Approximation of planar vectors

Let $\mathbf{f} = (3, 5)$ be a vector in the xy plane and suppose we want to approximate this vector by a vector aligned in the direction of another vector that is restricted to be aligned with some vector (a, b) . Figure 3.1 depicts the situation. This is the simplest approximation problem for vectors. Nevertheless, for many readers it will be wise to refresh some basic linear algebra by consulting a textbook. Exercise 3.1 suggests specific tasks to regain familiarity with fundamental operations on inner product vector spaces. Familiarity with such operations are assumed in the forthcoming text.

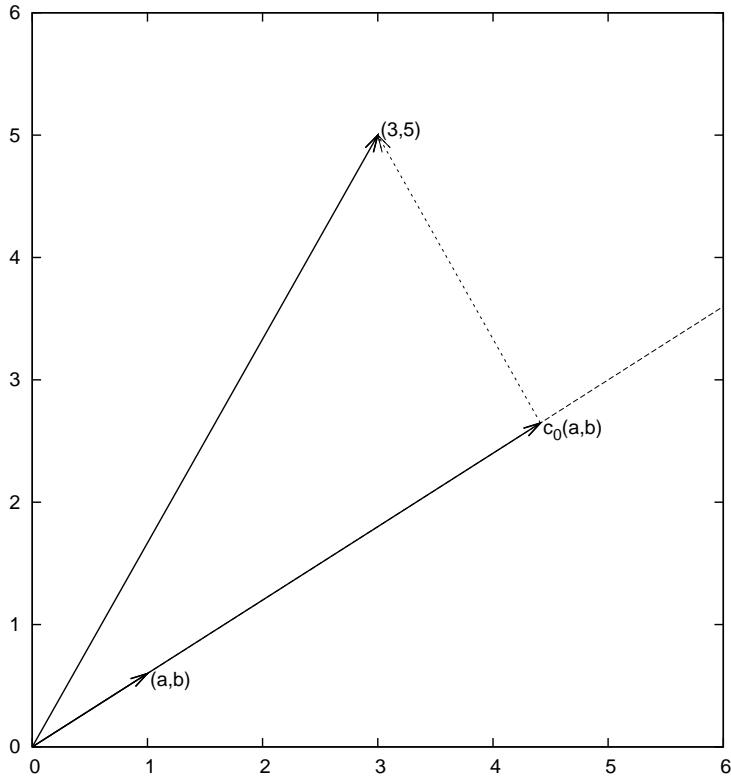


Fig. 3.1 Approximation of a two-dimensional vector in a one-dimensional vector space.

We introduce the vector space V spanned by the vector $\psi_0 = (a, b)$:

$$V = \text{span} \{ \psi_0 \}. \quad (3.2)$$

We say that ψ_0 is a *basis vector* in the space V . Our aim is to find the vector

$$\mathbf{u} = c_0 \psi_0 \in V \quad (3.3)$$

which best approximates the given vector $\mathbf{f} = (3, 5)$. A reasonable criterion for a best approximation could be to minimize the length of the difference between the approximate \mathbf{u} and the given \mathbf{f} . The difference, or error $\mathbf{e} = \mathbf{f} - \mathbf{u}$, has its length given by the *norm*

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

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

$$(\mathbf{u}, \mathbf{v}) = u_0 v_0 + u_1 v_1. \quad (3.4)$$

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 at the origin and ending in the point (a, b) , while (\mathbf{u}, \mathbf{v}) with vectors \mathbf{u} and \mathbf{v} means the inner product of these vectors. Since vectors are here written in boldface font there should be no confusion. We may add that the norm associated with this inner product is the usual Euclidean length of a vector, i.e.,

$$\|\mathbf{u}\| = \sqrt{(\mathbf{u}, \mathbf{u})} = \sqrt{u_0^2 + u_1^2}$$

The least squares method. We now want to determine the \mathbf{u} that minimizes $\|\mathbf{e}\|$, that is we want to compute the optimal c_0 in (3.3). The algebra is simplified if we minimize the square of the norm, $\|\mathbf{e}\|^2 = (\mathbf{e}, \mathbf{e})$, instead of the norm itself. Define the function

$$E(c_0) = (\mathbf{e}, \mathbf{e}) = (\mathbf{f} - c_0 \psi_0, \mathbf{f} - c_0 \psi_0). \quad (3.5)$$

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

$$E(c_0) = (\mathbf{f}, \mathbf{f}) - 2c_0(\mathbf{f}, \psi_0) + c_0^2(\psi_0, \psi_0). \quad (3.6)$$

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

$$(\alpha \mathbf{u}, \mathbf{v}) = \alpha (\mathbf{u}, \mathbf{v}), \quad \alpha \in \mathbb{R}, \quad (3.7)$$

$$(\mathbf{u} + \mathbf{v}, \mathbf{w}) = (\mathbf{u}, \mathbf{w}) + (\mathbf{v}, \mathbf{w}), \quad (3.8)$$

$$(\mathbf{u}, \mathbf{v}) = (\mathbf{v}, \mathbf{u}). \quad (3.9)$$

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

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

It turns out that E has one unique minimum and no maximum point. Now, when differentiating (3.6) with respect to c_0 , note that none of the inner product expressions depend on c_0 , so we simply get

$$\frac{\partial E}{\partial c_0} = -2(\mathbf{f}, \boldsymbol{\psi}_0) + 2c_0(\boldsymbol{\psi}_0, \boldsymbol{\psi}_0). \quad (3.10)$$

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

$$c_0 = \frac{(\mathbf{f}, \boldsymbol{\psi}_0)}{(\boldsymbol{\psi}_0, \boldsymbol{\psi}_0)}, \quad (3.11)$$

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

$$c_0 = \frac{3a + 5b}{a^2 + b^2}. \quad (3.12)$$

For later, it is worth mentioning that setting the key equation (3.10) to zero and ordering the terms lead to

$$(\mathbf{f} - c_0 \boldsymbol{\psi}_0, \boldsymbol{\psi}_0) = 0,$$

or

$$(\mathbf{e}, \boldsymbol{\psi}_0) = 0. \quad (3.13)$$

This implication of minimizing E is an important result that we shall make much use of.

The projection method. We shall now show that minimizing $\|\mathbf{e}\|^2$ implies that \mathbf{e} is orthogonal to *any* vector \mathbf{v} in the space V . This result is visually quite clear from Figure 3.1 (think of other vectors along the line (a, b) : all of them will lead to a larger distance between the approximation

and \mathbf{f}). Then we see mathematically that \mathbf{e} is orthogonal to any vector \mathbf{v} in the space V and we may express any $\mathbf{v} \in V$ as $\mathbf{v} = s\psi_0$ for any scalar parameter s (recall that two vectors are orthogonal when their inner product vanishes). Then we calculate the inner product

$$\begin{aligned} (\mathbf{e}, s\psi_0) &= (\mathbf{f} - c_0\psi_0, s\psi_0) \\ &= (\mathbf{f}, s\psi_0) - (c_0\psi_0, s\psi_0) \\ &= s(\mathbf{f}, \psi_0) - sc_0(\psi_0, \psi_0) \\ &= s(\mathbf{f}, \psi_0) - s \frac{(\mathbf{f}, \psi_0)}{(\psi_0, \psi_0)} (\psi_0, \psi_0) \\ &= s((\mathbf{f}, \psi_0) - (\mathbf{f}, \psi_0)) \\ &= 0. \end{aligned}$$

Therefore, instead of minimizing the square of the norm, we could demand that \mathbf{e} is orthogonal to any vector in V , which in our present simple case amounts to a single vector only. This method is known as *projection*. (The approach can also be referred to as a Galerkin method as explained at the end of Section 3.1.2.)

Mathematically, the projection method is stated by the equation

$$(\mathbf{e}, \mathbf{v}) = 0, \quad \forall \mathbf{v} \in V. \quad (3.14)$$

An arbitrary $\mathbf{v} \in V$ can be expressed as $s\psi_0$, $s \in \mathbb{R}$, and therefore (3.14) implies

$$(\mathbf{e}, s\psi_0) = s(\mathbf{e}, \psi_0) = 0,$$

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

$$(\mathbf{e}, \psi_0) = 0 \quad \text{or} \quad (\mathbf{f} - c_0\psi_0, \psi_0) = 0,$$

which is what we found in (3.13) from the least squares computations.

3.1.2 Approximation of general vectors

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

$$V = \text{span} \{ \psi_0, \dots, \psi_N \}.$$

We assume that the space has dimension $N + 1$ and that *basis vectors* ψ_0, \dots, ψ_N are linearly independent so that none of them are redundant. Any vector $\mathbf{u} \in V$ can then be written as a linear combination of the basis vectors, i.e.,

$$\mathbf{u} = \sum_{j=0}^N c_j \psi_j,$$

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

The least squares method. Now we want to find c_0, \dots, c_N , such that \mathbf{u} is the best approximation to \mathbf{f} in the sense that the distance (error) $\mathbf{e} = \mathbf{f} - \mathbf{u}$ is minimized. Again, we define the squared distance as a function of the free parameters c_0, \dots, c_N ,

$$\begin{aligned} E(c_0, \dots, c_N) &= (\mathbf{e}, \mathbf{e}) = (\mathbf{f} - \sum_j c_j \psi_j, \mathbf{f} - \sum_j c_j \psi_j) \\ &= (\mathbf{f}, \mathbf{f}) - 2 \sum_{j=0}^N c_j (\mathbf{f}, \psi_j) + \sum_{p=0}^N \sum_{q=0}^N c_p c_q (\psi_p, \psi_q). \end{aligned} \quad (3.15)$$

Minimizing this E with respect to the independent variables c_0, \dots, c_N is obtained by requiring

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

The first term in (3.15) is independent of c_i , so its derivative vanishes. The second term in (3.15) is differentiated as follows:

$$\frac{\partial}{\partial c_i} 2 \sum_{j=0}^N c_j (\mathbf{f}, \psi_j) = 2(\mathbf{f}, \psi_i), \quad (3.16)$$

since the expression to be differentiated is a sum and only one term, $c_i(\mathbf{f}, \psi_i)$, contains c_i (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 (3.15) is more tedious to differentiate. It can be wise to write out the double sum for $N = 1$ and perform differentiation with respect to c_0 and c_1 to see the structure of the expression. Thereafter, one can generalize to an arbitrary N and observe that

$$\frac{\partial}{\partial c_i} c_p c_q = \begin{cases} 0, & \text{if } p \neq i \text{ and } q \neq i, \\ c_q, & \text{if } p = i \text{ and } q \neq i, \\ c_p, & \text{if } p \neq i \text{ and } q = i, \\ 2c_i, & \text{if } p = q = i. \end{cases} \quad (3.17)$$

Then

$$\frac{\partial}{\partial c_i} \sum_{p=0}^N \sum_{q=0}^N c_p c_q (\psi_p, \psi_q) = \sum_{p=0, p \neq i}^N c_p (\psi_p, \psi_i) + \sum_{q=0, q \neq i}^N c_q (\psi_i, \psi_q) + 2c_i (\psi_i, \psi_i).$$

Since each of the two sums is missing the term $c_i(\psi_i, \psi_i)$, we may split the very last term in two, to get exactly that “missing” term for each sum. This idea allows us to write

$$\frac{\partial}{\partial c_i} \sum_{p=0}^N \sum_{q=0}^N c_p c_q (\psi_p, \psi_q) = 2 \sum_{j=0}^N c_i (\psi_j, \psi_i). \quad (3.18)$$

It then follows that setting

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

implies

$$-2(\mathbf{f}, \psi_i) + 2 \sum_{j=0}^N c_i (\psi_j, \psi_i) = 0, \quad i = 0, \dots, N.$$

Moving the first term to the right-hand side shows that the equation is actually a *linear system* for the unknown parameters c_0, \dots, c_N :

$$\sum_{j=0}^N A_{i,j} c_j = b_i, \quad i = 0, \dots, N, \quad (3.19)$$

where

$$A_{i,j} = (\psi_i, \psi_j), \quad (3.20)$$

$$b_i = (\psi_i, \mathbf{f}). \quad (3.21)$$

We have changed the order of the two vectors in the inner product according to (3.9):

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

simply because the sequence i - j looks more aesthetic.

The Galerkin or projection method. In analogy with the “one-dimensional” example in Section 3.1.1, it holds also here in the general case that minimizing the distance (error) \mathbf{e} is equivalent to demanding that \mathbf{e} is orthogonal to all $\mathbf{v} \in V$:

$$(\mathbf{e}, \mathbf{v}) = 0, \quad \forall \mathbf{v} \in V. \quad (3.22)$$

Since any $\mathbf{v} \in V$ can be written as $\mathbf{v} = \sum_{i=0}^N c_i \psi_i$, the statement (3.22) is equivalent to saying that

$$(\mathbf{e}, \sum_{i=0}^N c_i \psi_i) = 0,$$

for any choice of coefficients c_0, \dots, c_N . The latter equation can be rewritten as

$$\sum_{i=0}^N c_i (\mathbf{e}, \psi_i) = 0.$$

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

$$(\mathbf{e}, \psi_i) = 0, \quad i = 0, \dots, N. \quad (3.23)$$

These $N + 1$ equations result in the same linear system as (3.19):

$$(\mathbf{f} - \sum_{j=0}^N c_j \psi_j, \psi_i) = (\mathbf{f}, \psi_i) - \sum_{j=0}^N (\psi_i, \psi_j) c_j = 0,$$

and hence

$$\sum_{j=0}^N (\psi_i, \psi_j) c_j = (\mathbf{f}, \psi_i), \quad i = 0, \dots, N.$$

So, instead of differentiating the $E(c_0, \dots, c_N)$ function, we could simply use (3.22) as the principle for determining c_0, \dots, c_N , resulting in the $N + 1$ equations (3.23).

The names *least squares method* or *least squares approximation* are natural since the calculations consists of minimizing $\|\mathbf{e}\|^2$, and $\|\mathbf{e}\|^2$ is a

sum of squares of differences between the components in \mathbf{f} and \mathbf{u} . We find \mathbf{u} such that this sum of squares is minimized.

The principle (3.22), or the equivalent form (3.23), is known as *projection*. Almost the same mathematical idea was used by the Russian mathematician Boris Galerkin to solve differential equations, resulting in what is widely known as *Galerkin's method*.

3.2 Approximation principles

Let V be a function space spanned by a set of *basis functions* ψ_0, \dots, ψ_N ,

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

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

$$u = \sum_{j \in \mathcal{I}_s} c_j \psi_j. \quad (3.24)$$

That is, we consider functions as vectors in a vector space – a so-called function space – and we have a finite set of basis functions that span the space just as basis vectors or unit vectors span a vector space.

The index set \mathcal{I}_s is defined as $\mathcal{I}_s = \{0, \dots, N\}$ and is from now on used both for compact notation and for flexibility in the numbering of elements in sequences.

For now, in this introduction, we shall look at functions of a single variable x : $u = u(x)$, $\psi_j = \psi_j(x)$, $j \in \mathcal{I}_s$. Later, we will almost trivially extend the mathematical details to functions of two- or three-dimensional physical spaces. The approximation (3.24) is typically used to discretize a problem in space. Other methods, most notably finite differences, are common for time discretization, although the form (3.24) can be used in time as well.

3.2.1 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 3.1.2. That is, we minimize the distance between u and f . However, this requires a norm for measuring distances, and a

norm is most conveniently defined through an inner product. Viewing a function as a vector of infinitely many point values, one for each value of x , the inner product of two arbitrary functions $f(x)$ and $g(x)$ could intuitively be defined as the usual summation of pairwise “components” (values), with summation replaced by integration:

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

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

$$(f, g) = \int_{\Omega} f(x)g(x) \, dx. \quad (3.25)$$

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

$$E = (f(x) - \sum_{j \in \mathcal{I}_s} c_j \psi_j(x), f(x) - \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)). \quad (3.26)$$

Note the analogy with (3.15): the given function f plays the role of the given vector \mathbf{f} , and the basis function ψ_i plays the role of the basis vector ψ_i . We can rewrite (3.26), through similar steps as used for the result (3.15), leading to

$$E(c_1, \dots, c_N) = (f, f) - 2 \sum_{j \in \mathcal{I}_s} c_j (f, \psi_j) + \sum_{p \in \mathcal{I}_s} \sum_{q \in \mathcal{I}_s} c_p c_q (\psi_p, \psi_q). \quad (3.27)$$

Minimizing this function of $N + 1$ scalar variables $\{c_i\}_{i \in \mathcal{I}_s}$, requires differentiation with respect to c_i , for all $i \in \mathcal{I}_s$. 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 (3.19), with basically the same expressions:

$$A_{i,j} = (\psi_i, \psi_j), \quad (3.28)$$

$$b_i = (f, \psi_i). \quad (3.29)$$

The only difference from (3.19) is that the inner product is defined in terms of integration rather than summation.

3.2.2 The projection (or Galerkin) method

As in Section 3.1.2, the minimization of (e, e) is equivalent to

$$(e, v) = 0, \quad \forall v \in V. \quad (3.30)$$

This is known as a projection of a function f onto the subspace V . We may also call it a Galerkin method for approximating functions. Using the same reasoning as in (3.22)-(3.23), it follows that (3.30) is equivalent to

$$(e, \psi_i) = 0, \quad i \in \mathcal{I}_s. \quad (3.31)$$

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 (3.28) and right-hand side vector (3.29).

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

3.2.3 Example of linear approximation

Let us apply the theory in the previous section to a simple problem: given a parabola $f(x) = 10(x - 1)^2 - 1$ for $x \in \Omega = [1, 2]$, find the best approximation $u(x)$ in the space of all linear functions:

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

With our notation, $\psi_0(x) = 1$, $\psi_1(x) = x$, and $N = 1$. We seek

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

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

$$A_{0,0} = (\psi_0, \psi_0) = \int_1^2 1 \cdot 1 \, dx = 1, \quad (3.32)$$

$$A_{0,1} = (\psi_0, \psi_1) = \int_1^2 1 \cdot x \, dx = 3/2, \quad (3.33)$$

$$A_{1,0} = A_{0,1} = 3/2, \quad (3.34)$$

$$A_{1,1} = (\psi_1, \psi_1) = \int_1^2 x \cdot x \, dx = 7/3. \quad (3.35)$$

The corresponding right-hand side is

$$b_1 = (f, \psi_0) = \int_1^2 (10(x-1)^2 - 1) \cdot 1 \, dx = 7/3, \quad (3.36)$$

$$b_2 = (f, \psi_1) = \int_1^2 (10(x-1)^2 - 1) \cdot x \, dx = 13/3. \quad (3.37)$$

Solving the linear system results in

$$c_0 = -38/3, \quad c_1 = 10, \quad (3.38)$$

and consequently

$$u(x) = 10x - \frac{38}{3}. \quad (3.39)$$

Figure 3.2 displays the parabola and its best approximation in the space of all linear functions.

3.2.4 Implementation of the least squares method

Symbolic integration. The linear system can be computed either symbolically or numerically (a numerical integration rule is needed in the latter case). Let us first compute the system and its solution symbolically, i.e., using classical “pen and paper” mathematics with symbols. The Python package `sympy` can greatly help with this type of mathematics, and will therefore be frequently used in this text. Some basic familiarity with `sympy` is assumed, typically `symbols`, `integrate`, `diff`, `expand`, and `simplify`. Much can be learned by studying the many applications of `sympy` that will be presented.

Below is a function for symbolic computation of the linear system, where $f(x)$ is given as a `sympy` expression `f` involving the symbol `x`, `psi`

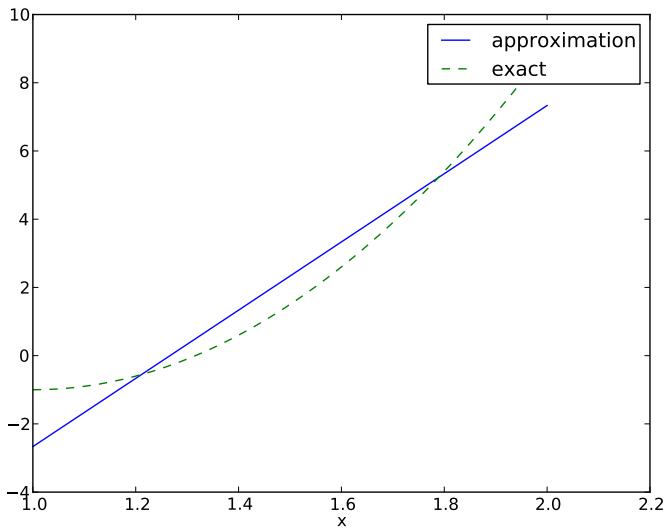


Fig. 3.2 Best approximation of a parabola by a straight line.

is a list of expressions for $\{\psi_i\}_{i \in \mathcal{I}_s}$, and Ω_{mega} is a 2-tuple/list holding the limits of the domain Ω :

```
import sympy as sym

def least_squares(f, psi, Omega):
    N = len(psi) - 1
    A = sym.zeros(N+1, N+1)
    b = sym.zeros(N+1, 1)
    x = sym.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            A[i,j] = sym.integrate(psi[i]*psi[j],
                                   (x, Omega[0], Omega[1]))
            A[j,i] = A[i,j]
        b[i,0] = sym.integrate(psi[i]*f, (x, Omega[0], Omega[1]))
    c = A.LUsolve(b)
    # Note: c is a sympy Matrix object, solution is in c[:,0]
    u = 0
    for i in range(len(psi)):
        u += c[i,0]*psi[i]
    return u, c
```

Observe that we exploit the symmetry of the coefficient matrix: only the upper triangular part is computed. Symbolic integration, also in

`sympy`, is often time consuming, and (roughly) halving the work has noticeable effect on the waiting time for the computations to finish.

Notice

We remark that the symbols in `sympy` are created and stored in a symbol factory that is indexed by the expression used in the construction and that repeated constructions from the same expression will not create new objects. The following code illustrates the behavior of the symbol factory:

```
>>> from sympy import *
>>> x0 = Symbol("x")
>>> x1 = Symbol("x")
>>> id(x0) == id(x1)
True
>>> a0 = 3.0
>>> a1 = 3.0
>>> id(a0) == id(a1)
False
```

Fall back on numerical integration. Obviously, `sympy` may fail to successfully integrate $\int_{\Omega} \psi_i \psi_j dx$, and especially $\int_{\Omega} f \psi_i dx$, symbolically. Therefore, we should extend the `least_squares` function such that it falls back on numerical integration if the symbolic integration is unsuccessful. In the latter case, the returned value from `sympy`'s `integrate` function is an object of type `Integral`. We can test on this type and utilize the `mpmath` module to perform numerical integration of high precision. Even when `sympy` manages to integrate symbolically, it can take an undesirably long time. We therefore include an argument `symbolic` that governs whether or not to try symbolic integration. Here is a complete and improved version of the previous function `least_squares`:

```
def least_squares(f, psi, Omega, symbolic=True):
    N = len(psi) - 1
    A = sym.zeros(N+1, N+1)
    b = sym.zeros(N+1, 1)
    x = sym.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            integrand = psi[i]*psi[j]
            if symbolic:
                I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
            if not symbolic or isinstance(I, sym.Integral):
                # Could not integrate symbolically, use numerical int.
```

```

integrand = sym.lambdify([x], integrand, 'mpmath')
I = mpmath.quad(integrand, [Omega[0], Omega[1]])
A[i,j] = A[j,i] = I

integrand = psi[i]*f
if symbolic:
    I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
if not symbolic or isinstance(I, sym.Integral):
    # Could not integrate symbolically, use numerical int.
    integrand = sym.lambdify([x], integrand, 'mpmath')
    I = mpmath.quad(integrand, [Omega[0], Omega[1]])
b[i,0] = I
if symbolic:
    c = A.LUsolve(b) # symbolic solve
    # c is a sympy Matrix object, numbers are in c[i,0]
    c = [sym.simplify(c[i,0]) for i in range(c.shape[0])]
else:
    c = mpmath.lu_solve(A, b) # numerical solve
    c = [c[i,0] for i in range(c.rows)]
u = sum(c[i]*psi[i] for i in range(len(psi)))
return u, c

```

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

Plotting the approximation. Comparing the given $f(x)$ and the approximate $u(x)$ visually is done with the following function, which utilizes `sympy`'s `lambdify` tool to convert a `sympy` expression to a Python function for numerical computations:

```

def comparison_plot(f, u, Omega, filename='tmp.pdf'):
    x = sym.Symbol('x')
    f = sym.lambdify([x], f, modules="numpy")
    u = sym.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` functions are found in the file `approx1D.py`. The `comparison_plot` function in this file is more advanced and flexible than the simplistic version shown above. The

file `ex_approx1D.py` applies the `approx1D` module to accomplish the forthcoming examples.

Notice

We remind the reader that the code examples can be found in a tarball at <http://hplgit.github.io/fem-book/doc/web/>. The following command shows a useful way to search for code

```
Terminal> find . -name '*.py' -exec grep least_squares {} \; -print
```

Here `'.'` specifies the directory for the search, `-name '*.py'` that files with suffix `*.py` should be searched through while

```
-exec grep least_squares {} \; -print
```

means that all lines containing the text `least_squares` should be printed to the screen.

3.2.5 Perfect approximation

Let us use the code above to recompute the problem from Section 3.2.3 where we want to approximate a parabola. What happens if we add an element x^2 to the set of basis functions 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 = sym.Symbol('x')
>>> f = 10*(x-1)**2-1
>>> u, c = least_squares(f=f, psi=[1, x, x**2], Omega=[1, 2])
>>> print(u)
10*x**2 - 20*x + 9
>>> print(sym.expand(f))
10*x**2 - 20*x + 9
```

Now, what if we use $\psi_i(x) = x^i$ for $i = 0, 1, \dots, N = 40$? The output from `least_squares` gives $c_i = 0$ for $i > 2$, which means that the method finds the perfect approximation.

In fact, we have a general result that if $f \in V$, the least squares and projection/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, $f = \sum_{j \in \mathcal{I}_s} d_j \psi_j$, for some coefficients $\{d_j\}_{j \in \mathcal{I}_s}$, and the right-hand side then has entries

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

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

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

which implies that $c_i = d_i$ for $i \in \mathcal{I}_s$.

3.2.6 The regression method

So far, the function to be approximated has been known in terms of a formula $f(x)$. Very often in applications, no formula is known, but the function value is known at a set of points. If we use $N + 1$ basis functions and know exactly $N + 1$ function values, we can determine the coefficients c_i by *interpolation* as explained in Section 3.4.1. The approximating function will then equal the f values at the points where the f values are sampled.

However, one normally has f sampled at a lot of points, here denoted by x_0, x_1, \dots, x_m , and we assume $m \gg N$. What can we do then to determine the coefficients? The answer is to find a least squares approximation. The resulting method is called *regression* and is well known from statistics when fitting a simple (usually polynomial) function to a set of data points.

Overdetermined equation system. Intuitively, we would demand u to equal f at all the data points x_i , $i = 0, 1, \dots, m$,

$$u(x_i) = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x_i) = f(x_i), \quad i = 0, 1, \dots, m. \quad (3.40)$$

The fundamental problem here is that we have more equations than unknowns since there are $N + 1$ unknowns and $m + 1 > N + 1$ equations. Such a system of equations is called an *overdetermined system*. We can write it in matrix form as

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i = 0, 1, \dots, m, \quad (3.41)$$

with the coefficient matrix and right-hand side vector given by

$$A_{i,j} = \psi_j(x_i), \quad (3.42)$$

$$b_i = f(x_i). \quad (3.43)$$

Note that the matrix is a *rectangular* $(m+1) \times (N+1)$ matrix since $i = 0, \dots, m$ and $j = 0, \dots, N$.

The normal equations derived from a least squares principle. The least squares method is a common technique for solving overdetermined equations systems. Let us write the overdetermined system $\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i$ more compactly in matrix form as $Ac = b$. Since we have more equations than unknowns, it is (in general) impossible to find a vector c that fulfills $Ac = b$. The best we can do is to make the residual $r = b - Ac$ as small as possible. That is, we can find c such that it minimizes the Euclidean norm of r : $\|r\|$. The algebra simplifies significantly by minimizing $\|r\|^2$ instead. This principle corresponds to a least squares method.

The i -th component of r reads $r_i = b_i - \sum_j A_{i,j} c_j$, so $\|r\|^2 = \sum_i r_i^2$. Minimizing $\|r\|^2$ with respect to the unknowns c_0, \dots, c_N implies that

$$\frac{\partial}{\partial c_k} \|r\|^2 = 0, \quad k = 0, \dots, N, \quad (3.44)$$

which leads to

$$\frac{\partial}{\partial c_k} \sum_i r_i^2 = \sum_i 2r_i \frac{\partial r_i}{\partial c_k} = \sum_i 2r_i \frac{\partial}{\partial c_k} (b_i - \sum_j A_{i,j} c_j) = 2 \sum_i r_i (-A_{i,k}) = 0.$$

By inserting $r_i = b_i - \sum_j A_{i,j} c_j$ in the last expression we get

$$\sum_i \left(b_i - \sum_j A_{i,j} c_j \right) (-A_{i,k}) = - \sum_i b_i A_{i,k} + \sum_j (\sum_i A_{i,j} A_{i,k}) c_j = 0.$$

Introducing the transpose of A , A^T , we know that $A_{i,j}^T = A_{j,i}$. Therefore, the expression $\sum_i A_{i,j} A_{i,k}$ can be written as $\sum_i A_{k,i}^T A_{i,j}$ and be recognized as the formula for the matrix-matrix product $A^T A$. Also, $\sum_i b_i A_{i,k}$ can be written $\sum_i A_{k,i}^T b_i$ and recognized as the matrix-vector product $A^T b$. These observations imply that (3.44) is equivalent to the linear system

$$\sum_j (\sum_i A_{k,i}^T A_{i,j}) c_j = \sum_j (A^T A)_{k,j} c_j = \sum_i A_{k,i}^T b_i = (A^T b)_k, \quad k = 0, \dots, N, \quad (3.45)$$

or in matrix form,

$$A^T A c = A^T b. \quad (3.46)$$

The equation system (3.45) or (3.46) are known as the *normal equations*. With A as an $(m+1) \times (N+1)$ matrix, $A^T A$ becomes an $(N+1) \times (N+1)$ matrix, and $A^T b$ becomes a vector of length $N+1$. Often, $m \gg N$, so $A^T A$ is much smaller than A .

Many prefer to write the linear system (3.45) in the standard form $\sum_j B_{i,j} c_j = d_i$, $i = 0, \dots, N$. We can easily do so by exchanging the i and k indices ($i \leftrightarrow k$), $\sum_i A_{k,i}^T A_{i,j} = \sum_k A_{i,k}^T A_{k,j}$, and setting $B_{i,j} = \sum_k A_{i,k}^T A_{k,j}$. Similarly, we exchange i and k in the right-hand side expression and get $\sum_k A_{i,k}^T b_k = d_i$. Expressing $B_{i,j}$ and d_i in terms of the ψ_i and x_i , using (3.42) and (3.43), we end up with the formulas

$$B_{i,j} = \sum_k A_{i,k}^T A_{k,j} = \sum_k A_{k,i} A_{k,j} = \sum_{k=0}^m \psi_i(x_k) \psi_j(x_k), \quad (3.47)$$

$$d_i = \sum_k A_{i,k}^T b_k = \sum_k A_{k,i} b_k = \sum_{k=0}^m \psi_i(x_k) f(x_k) \quad (3.48)$$

Implementation. The following function defines the matrix entries $B_{i,j}$ according to (3.47) and the right-hand side entries d_i according to (3.48). Thereafter, it solves the linear system $\sum_j B_{i,j} c_j = d_i$. The input data f and ψ_i hold $f(x)$ and ψ_i , $i = 0, \dots, N$, as a symbolic expression, but since m is thought to be much larger than N , and there are loops from 0 to m , we use numerical computing to speed up the computations.

```
def regression(f, psi, points):
    N = len(psi) - 1
    m = len(points)
    # Use numpy arrays and numerical computing
    B = np.zeros((N+1, N+1))
    d = np.zeros(N+1)
    # Wrap psi and f in Python functions rather than expressions
    # so that we can evaluate psi at points[i]
    x = sym.Symbol('x')
    psi_sym = psi # save symbolic expression
    psi = [sym.lambdify([x], psi[i]) for i in range(N+1)]
    f = sym.lambdify([x], f)
    for i in range(N+1):
        for j in range(N+1):
            B[i,j] = 0
            for k in range(m+1):
                B[i,j] += psi[i](points[k])*psi[j](points[k])
    d = np.array([f(points[i]) for i in range(m+1)])
    c = np.linalg.solve(B, d)
    return c
```

```

d[i] = 0
for k in range(m+1):
    d[i] += psi[i](points[k])*f(points[k])
c = np.linalg.solve(B, d)
u = sum(c[i]*psi_sym[i] for i in range(N+1))
return u, c

```

Example. We repeat the computational example from Section 3.4.1, but this time with many more points. The parabola $f(x) = 10(x - 1)^2 - 1$ is to be approximated by a linear function on $\Omega = [1, 2]$. We divide Ω into $m + 2$ intervals and use the inner $m + 1$ points:

```

import sympy as sym
x = sym.Symbol('x')
f = 10*(x-1)**2 - 1
psi = [1, x]
Omega = [1, 2]
m_values = [2-1, 8-1, 64-1]
# Create m+3 points and use the inner m+1 points
for m in m_values:
    points = np.linspace(Omega[0], Omega[1], m+3)[1:-1]
    u, c = regression(f, psi, points)
    comparison_plot(
        f, u, Omega,
        filename='parabola_by_regression_%d' % (m+1),
        points=points,
        points_legend='%d interpolation points' % (m+1),
        legend_loc='upper left')

```

Figure 3.3 shows results for $m + 1 = 2$ (left), $m + 1 = 8$ (middle), and $m + 1 = 64$ (right) data points. The approximating function is not so sensitive to the number of points as long as they cover a significant part of the domain (the first 2 point approximation puts too much weight on the center, while the 8 point approximation cover almost the entire domain and produces a good approximation which is barely improved with 64 points):

$$u(x) = 10x - 13.2, \quad 2 \text{ points}$$

$$u(x) = 10x - 12.7, \quad 8 \text{ points}$$

$$u(x) = 10x - 12.7, \quad 64 \text{ points}$$

To explicitly make the link to classical regression in statistics, we consider $f = 10(x - 1)^2 - 1 + \epsilon$, where ϵ is a random, normally distributed variable. The goal in classical regression is to find the straight line that best fits the data points (in a least squares sense). The only difference from the previous setup, is that the $f(x_i)$ values are based on a function

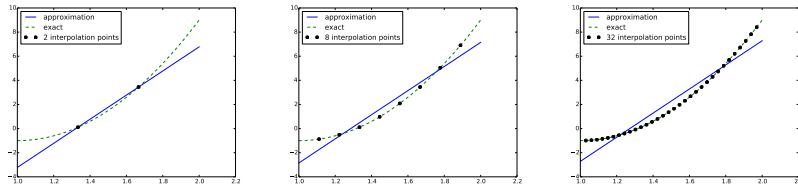


Fig. 3.3 Approximation of a parabola by a regression method with varying number of points.

formula, here $10(x - 1)^2 - 1$, *plus* normally distributed noise. Figure 3.4 shows three sets of data points, along with the original $f(x)$ function without noise, and the straight line that is a least squares approximation to the data points.

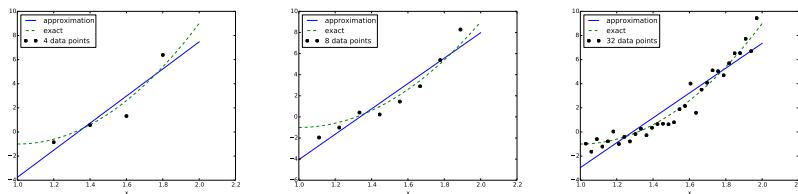


Fig. 3.4 Approximation of a parabola with noise by a straight line.

We can fit a parabola instead of a straight line, as done in Figure 3.5. When m becomes large, the fitted parabola and the original parabola without noise become very close.

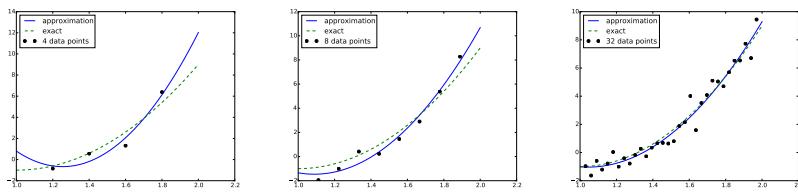


Fig. 3.5 Approximation of a parabola with noise by a parabola.

The regression method is not much used for approximating differential equations or a given function, but is central in uncertainty quantification methods such as polynomial chaos expansions.

The residual: an indirect but computationally cheap measure of the error

When attempting to solve a system $Ac = b$, we may question how far off a start vector or a current approximation c_0 is. The error is clearly the difference between c and c_0 , $e = c - c_0$, but since we do not know the true solution c we are unable to assess the error. However, the vector c_0 is the solution of the an alternative problem $Ac_0 = b_0$. If the input, i.e., the right-hand sides b_0 and b are close to each other then we expect the output of a solution process c and c_0 to be close to each other. Furthermore, while b_0 in principle is unknown, it is easily computable as $b_0 = Ac_0$ and does not require inversion of A . The vector $b - b_0$ is the so-called residual r defined by

$$r = b - b_0 = b - Ac_0 = Ac - Ac_0.$$

Clearly, the error and the residual are related by

$$Ae = r.$$

While the computation of the error requires inversion of A , which may be computationally expensive, the residual is easily computable and do only require a matrix-vector product and vector additions.

3.3 Orthogonal basis functions

Approximation of a function via orthogonal functions, especially sinusoidal functions or orthogonal polynomials, is a very popular and successful approach. The finite element method does not make use of orthogonal functions, but functions that are “almost orthogonal”.

3.3.1 Ill-conditioning

For basis functions that are not orthogonal the condition number of the matrix may create problems during the solution process due to, for example, round-off errors as will be illustrated in the following. The computational example in Section 3.2.5 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 of the form $u(x) = c_0 + c_1x + c_2x^2 + \dots + c_{10}x^{10}$. Analytically, we know that $c_j = 0$ for $j > 2$, but numerically we may get $c_j \neq 0$ 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 , $j = 0, 1, \dots, 10$, 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 `mpmath.fp.matrix` and the `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 very wrong values of the coefficients.

Increasing N to 12 makes the numerical solver in `numpy` 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 $\psi_i(x) = x^i$ 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 3.6. 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.

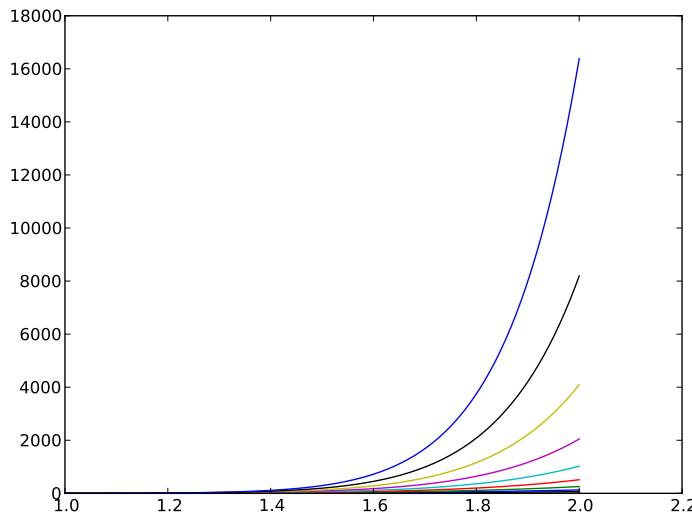


Fig. 3.6 The 15 first basis functions x^i , $i = 0, \dots, 14$.

On the other hand, the double precision `numpy` solver does 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 $c_j < 10^{-5}$ for $j > 20$). Even for $N = 100$ the approximation still 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 $\{1, x, x^2, x^3, \dots\}$ is a bad basis. The best basis from a matrix conditioning point of view is to have orthogonal functions such that $(\psi_i, \psi_j) = 0$ for $i \neq j$. There are many known sets of orthogonal polynomials and other functions. The functions used in the finite element method are almost orthogonal, and this property helps to avoid problems when 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 matrix systems arising from finite element discretizations.

3.3.2 Fourier series

A set of sine functions is widely used for approximating functions (note that the sines are orthogonal with respect to the L_2 inner product as can be easily verified using `sympy`). Let us take

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

That is,

$$\psi_i(x) = \sin((i+1)\pi x), \quad i \in \mathcal{I}_s.$$

An approximation to the parabola $f(x) = 10(x-1)^2 - 1$ for $x \in \Omega = [1, 2]$ from Section 3.2.3 can then be computed by the `least_squares` function from Section 3.2.4:

```
N = 3
import sympy as sym
x = sym.Symbol('x')
psi = [sym.sin(sym.pi*(i+1)*x) for i in range(N+1)]
f = 10*(x-1)**2 - 1
Omega = [0, 1]
u, c = least_squares(f, psi, Omega)
comparison_plot(f, u, Omega)
```

Figure 3.7 (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 3.7 (right).

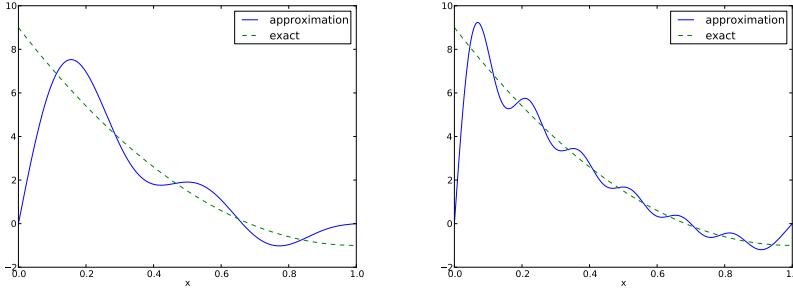


Fig. 3.7 Best approximation of a parabola by a sum of 3 (left) and 11 (right) sine functions.

There is an error $f(0) - u(0) = 9$ at $x = 0$ in Figure 3.7 regardless of how large N is, because all $\psi_i(0) = 0$ and hence $u(0) = 0$. We may help the approximation to be correct at $x = 0$ by seeking

$$u(x) = f(0) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x). \quad (3.49)$$

However, this adjustment 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:

$$u(x) = f(0)(1-x) + xf(1) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x). \quad (3.50)$$

This adjustment of u alters the linear system slightly. In the general case, we set

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x),$$

and the linear system becomes

$$\sum_{j \in \mathcal{I}_s} (\psi_i, \psi_j) c_j = (f - B, \psi_i), \quad i \in \mathcal{I}_s.$$

The calculations can still utilize the `least_squares` or `least_squares_orth` functions, but solve for $u - b$:

```
f0 = 0; f1 = -1
B = f0*(1-x) + x*f1
u_sum, c = least_squares_orth(f-b, psi, Omega)
u = B + u_sum
```

Figure 3.8 shows the result of the technique for ensuring the 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.

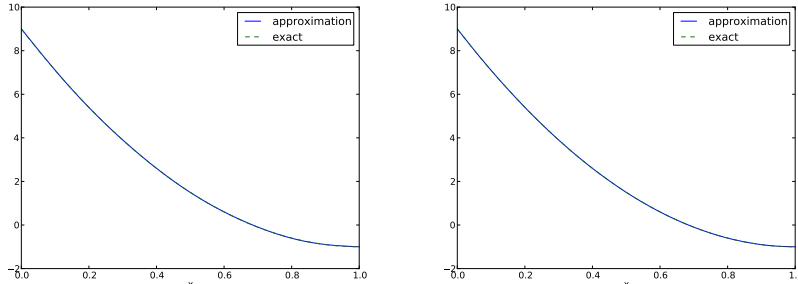


Fig. 3.8 Best approximation of a parabola by a sum of 3 (left) and 11 (right) sine functions with a boundary term.

3.3.3 Orthogonal basis functions

The choice of sine functions $\psi_i(x) = \sin((i + 1)\pi x)$ 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 (avoid i in the integrand as this symbol means the imaginary unit $\sqrt{-1}$). Asking WolframAlpha also about $\int_0^1 \sin^2(j\pi x) dx$, we find that it equals $1/2$. With a diagonal matrix we can easily solve for the coefficients by hand:

$$c_i = 2 \int_0^1 f(x) \sin((i + 1)\pi x) dx, \quad i \in \mathcal{I}_s, \quad (3.51)$$

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 3.2 results in the classical Fourier series for $f(x)$ (see Exercise 3.8 for details).

With 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, psi, Omega):
    N = len(psi) - 1
    A = [0]*(N+1)
    b = [0]*(N+1)
    x = sym.Symbol('x')
    for i in range(N+1):
        A[i] = sym.integrate(psi[i]**2, (x, Omega[0], Omega[1]))
        b[i] = sym.integrate(psi[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(psi)):
        u += c[i]*psi[i]
    return u, c
```

As mentioned in Section 3.2.4, symbolic integration may fail or take a very long time. It is therefore natural to extend the implementation above with a version where we can choose between symbolic and numerical integration and fall back on the latter if the former fails:

```
def least_squares_orth(f, psi, Omega, symbolic=True):
    N = len(psi) - 1
    A = [0]*(N+1)      # plain list to hold symbolic expressions
    b = [0]*(N+1)
    x = sym.Symbol('x')
    for i in range(N+1):
        # Diagonal matrix term
        A[i] = sym.integrate(psi[i]**2, (x, Omega[0], Omega[1]))

        # Right-hand side term
        integrand = psi[i]*f
        if symbolic:
            I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
        if not symbolic or isinstance(I, sym.Integral):
            print('numerical integration of', integrand)
            integrand = sym.lambdify([x], integrand, 'mpmath')
            I = mpmath.quad(integrand, [Omega[0], Omega[1]])
        b[i] = I
    c = [b[i]/A[i] for i in range(len(b))]
    u = 0
    u = sum(c[i]*psi[i] for i in range(len(psi)))
    return u, c
```

This function is found in the file `approx1D.py`. Observe that we here assume that $\int_{\Omega} \varphi_i^2 dx$ can always be symbolically computed, which is not an unreasonable assumption when the basis functions are orthogonal,

but there is no guarantee, so an improved version of the function above would implement numerical integration also for the $A[i,i]$ term.

3.3.4 Numerical computations

Sometimes the basis functions ψ_i and/or the function f have a nature that makes symbolic integration CPU-time consuming or impossible. Even though we implemented a fallback on numerical integration of $\int f \varphi_i dx$, considerable time might still be required by `sympy` just by attempting to integrate symbolically. Therefore, it will be handy to have a function for fast *numerical integration and numerical solution of the linear system*. Below is such a method. It requires Python functions `f(x)` and `psi(x, i)` for $f(x)$ and $\psi_i(x)$ as input. The output is a mesh function with values u on the mesh with points in the array `x`. Three numerical integration methods are offered: `scipy.integrate.quad` (precision set to 10^{-8}), `mpmath.quad` (about machine precision), and a Trapezoidal rule based on the points in `x` (unknown accuracy, but increasing with the number of mesh points in `x`).

```
def least_squares_numerical(f, psi, N, x,
                            integration_method='scipy',
                            orthogonal_basis=False):
    import scipy.integrate
    A = np.zeros((N+1, N+1))
    b = np.zeros(N+1)
    Omega = [x[0], x[-1]]
    dx = x[1] - x[0]      # assume uniform partition

    for i in range(N+1):
        j_limit = i+1 if orthogonal_basis else N+1
        for j in range(i, j_limit):
            print('(%d,%d)' % (i, j))
            if integration_method == 'scipy':
                A_ij = scipy.integrate.quad(
                    lambda x: psi(x,i)*psi(x,j),
                    Omega[0], Omega[1], epsabs=1E-9, epsrel=1E-9)[0]
            elif integration_method == 'sympy':
                A_ij = mpmath.quad(
                    lambda x: psi(x,i)*psi(x,j),
                    [Omega[0], Omega[1]])
            else:
                values = psi(x,i)*psi(x,j)
                A_ij = trapezoidal(values, dx)
            A[i,j] = A[j,i] = A_ij

    if integration_method == 'scipy':
        b_i = scipy.integrate.quad(
```

```

        lambda x: f(x)*psi(x,i), Omega[0], Omega[1],
        epsabs=1E-9, epsrel=1E-9)[0]
    elif integration_method == 'sympy':
        b_i = mpmath.quad(
            lambda x: f(x)*psi(x,i), [Omega[0], Omega[1]])
    else:
        values = f(x)*psi(x,i)
        b_i = trapezoidal(values, dx)
    b[i] = b_i

c = b/np.diag(A) if orthogonal_basis else np.linalg.solve(A, b)
u = sum(c[i]*psi(x, i) for i in range(N+1))
return u, c

def trapezoidal(values, dx):
    """Integrate values by the Trapezoidal rule (mesh size dx)."""
    return dx*(np.sum(values) - 0.5*values[0] - 0.5*values[-1])

```

Here is an example on calling the function:

```

from numpy import linspace, tanh, pi

def psi(x, i):
    return sin((i+1)*x)

x = linspace(0, 2*pi, 501)
N = 20
u, c = least_squares_numerical(lambda x: tanh(x-pi), psi, N, x,
                                 orthogonal_basis=True)

```

Remark. The `scipy.integrate.quad` integrator is usually much faster than `mpmath.quad`.

3.4 Interpolation

3.4.1 The interpolation (or collocation) principle

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 approaches as well. One is to demand that $u(x_i) = f(x_i)$ at some selected points x_i , $i \in \mathcal{I}_s$:

$$u(x_i) = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x_i) = f(x_i), \quad i \in \mathcal{I}_s. \quad (3.52)$$

We recognize that the equation $\sum_j c_j \psi_j(x_i) = f(x_i)$ is actually a linear system with $N + 1$ unknown coefficients $\{c_j\}_{j \in \mathcal{I}_s}$:

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s, \quad (3.53)$$

with coefficient matrix and right-hand side vector given by

$$A_{i,j} = \psi_j(x_i), \quad (3.54)$$

$$b_i = f(x_i). \quad (3.55)$$

This time the coefficient matrix is not symmetric because $\psi_j(x_i) \neq \psi_i(x_j)$ in general. The method is often referred to 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 this case the x_i points are called *interpolation points*. When the same approach is used to approximate differential equations, one usually applies the name *collocation method* and x_i are known as *collocation points*.

Given f as a `sympy` symbolic expression `f`, $\{\psi_i\}_{i \in \mathcal{I}_s}$ as a list `psi`, and a set of points $\{x_i\}_{i \in \mathcal{I}_s}$ as a list or array `points`, the following Python function sets up and solves the matrix system for the coefficients $\{c_i\}_{i \in \mathcal{I}_s}$:

```
def interpolation(f, psi, points):
    N = len(psi) - 1
    A = sym.zeros(N+1, N+1)
    b = sym.zeros(N+1, 1)
    psi_sym = psi # save symbolic expression
    x = sym.Symbol('x')
    psi = [sym.lambdify([x], psi[i], 'mpmath') for i in range(N+1)]
    f = sym.lambdify([x], f, 'mpmath')
    for i in range(N+1):
        for j in range(N+1):
            A[i,j] = psi[j](points[i])
        b[i,0] = f(points[i])
    c = A.LUsolve(b)
    # c is a sympy Matrix object, turn to list
    c = [sym.simplify(c[i,0]) for i in range(c.shape[0])]
    u = sym.simplify(sum(c[i]*psi_sym[i] for i in range(N+1)))
    return u, c
```

The `interpolation` function is a part of the `approx1D` module.

We found it convenient in the above function to turn the expressions `f` and `psi` into ordinary Python functions of `x`, which can be called with `float` values in the list `points` when building the matrix and the right-hand side. The alternative is to use the `subs` method to substitute the `x` variable in an expression by an element from the `points` list. The following session illustrates both approaches in a simple setting:

```
>>> import sympy as sym
>>> x = sym.Symbol('x')
>>> e = x**2                      # symbolic expression involving x
>>> p = 0.5                        # a value of x
>>> v = e.subs(x, p)               # evaluate e for x=p
>>> v
0.250000000000000
>>> type(v)
sympy.core.numbers.Float
>>> e = lambdify([x], e)    # make Python function of e
>>> type(e)
>>> function
>>> v = e(p)                   # evaluate e(x) for x=p
>>> v
0.25
>>> type(v)
float
```

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 the unit interval.

Example. Let us illustrate the interpolation 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 $x_1 = 1 + 2/3$:

```
import sympy as sym
x = sym.Symbol('x')
f = 10*(x-1)**2 - 1
psi = [1, x]
Omega = [1, 2]
points = [1 + sym.Rational(1,3), 1 + sym.Rational(2,3)]
u, c = interpolation(f, psi, 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 3.9 (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 3.9 (right).

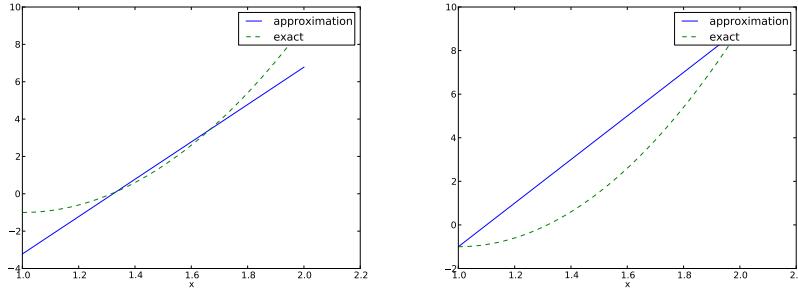


Fig. 3.9 Approximation of a parabola by linear functions computed by two interpolation points: $4/3$ and $5/3$ (left) versus 1 and 2 (right).

3.4.2 Lagrange polynomials

In Section 3.3.2 we explained the advantage of having a diagonal matrix: formulas for the coefficients $\{c_i\}_{i \in \mathcal{I}_s}$ can then be derived by hand. For an interpolation (or collocation) method a diagonal matrix implies that $\psi_j(x_i) = 0$ if $i \neq j$. One set of basis functions $\psi_i(x)$ 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.) Lagrange polynomials are key building blocks in the finite element method, so familiarity with these polynomials will be required anyway.

A Lagrange polynomial can be written as

$$\psi_i(x) = \prod_{j=0, j \neq i}^N \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_i - x_0} \dots \frac{x - x_{i-1}}{x_i - x_{i-1}} \frac{x - x_{i+1}}{x_i - x_{i+1}} \dots \frac{x - x_N}{x_i - x_N}, \quad (3.56)$$

for $i \in \mathcal{I}_s$. We see from (3.56) that all the ψ_i functions are polynomials of degree N which have the property

$$\psi_i(x_s) = \delta_{is}, \quad \delta_{is} = \begin{cases} 1, & i = s, \\ 0, & i \neq s, \end{cases} \quad (3.57)$$

when x_s is an interpolation (collocation) point. Here we have used the *Kronecker delta symbol* δ_{is} . This property implies that A is a diagonal matrix, i.e., $A_{i,j} = 0$ for $i \neq j$ and $A_{i,i} = 1$ when $i = j$. The solution of the linear system is then simply

$$c_i = f(x_i), \quad i \in \mathcal{I}_s, \quad (3.58)$$

and

$$u(x) = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x) = \sum_{j \in \mathcal{I}_s} f(x_j) \psi_j(x). \quad (3.59)$$

We remark however that (3.57) does not necessarily imply that the matrix obtained by the least squares or projection methods is diagonal.

The following function computes the Lagrange interpolating polynomial $\psi_i(x)$ on the unit interval $(0,1)$, given the interpolation points x_0, \dots, 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, ψ_0, \dots, ψ_N , using equidistant points throughout Ω :

```
def Lagrange_polyomials_01(x, N):
    if isinstance(x, sym.Symbol):
        h = sym.Rational(1, N-1)
    else:
        h = 1.0/(N-1)
    points = [i*h for i in range(N)]
    psi = [Lagrange_polynomial(x, i, points) for i in range(N)]
    return psi, points
```

When `x` is a `sym.Symbol` object, we let the spacing between the interpolation points, `h`, be a `sympy` rational number, so that we get nice end results in the formulas for ψ_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 - just think of `x` being a `sym.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 sym
>>> x = sym.Symbol('x')
>>> psi, points = Lagrange_polyomials_01(x, N=2)
>>> points
[0, 1/2, 1]
>>> psi
[(1 - x)*(1 - 2*x), 2*x*(2 - 2*x), -x*(1 - 2*x)]

>>> x = 0.5 # numerical computing
```

```
>>> psi, points = Lagrange_polynomials_01(x, N=2)
>>> points
[0.0, 0.5, 1.0]
>>> psi
[-0.0, 1.0, 0.0]
```

That is, when used symbolically, the `Lagrange_polynomials_01` function returns the symbolic expression for the Lagrange functions while when `x` is a numerical value the function returns the value of the basis function evaluate in `x`. In the present example only the second basis function should be 1 in the mid-point while the others are zero according to (3.57).

Approximation of a polynomial. The Galerkin or least squares methods lead to an exact approximation if f lies in the space spanned by the basis functions. It could be of interest to see how the interpolation method with Lagrange polynomials as the basis is able to approximate a polynomial, e.g., a parabola. Running

```
for N in 2, 4, 5, 6, 8, 10, 12:
    f = x**2
    psi, points = Lagrange_polynomials_01(x, N)
    u = interpolation(f, psi, points)
```

shows the result that up to $N=4$ we achieve an exact approximation, and then round-off errors start to grow, such that $N=15$ leads to a 15-degree polynomial for u where the coefficients in front of x^r for $r > 2$ are of size 10^{-5} and smaller. As the matrix is ill-conditioned and we use floating-point arithmetic, we do not obtain the exact solution. Still, we get a solution that is visually identical to the exact solution. The reason is that the ill-conditioning causes the system to have many solutions very close to the true solution. While we are lucky for $N=15$ and obtain a solution that is visually identical to the true solution, ill-conditioning may also result in completely wrong results. As we continue with higher values, $N=20$ reveals that the procedure is starting to fall apart as the approximate solution is around 0.9 at $x = 1.0$, where it should have been 1.0. At $N=30$ the approximate solution is around $5 \cdot 10^8$ at $x = 1$.

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 = sym.Symbol('x')
f = sym.sin(2*sym.pi*x)
psi, points = Lagrange_polynomials_01(x, N)
Omega=[0, 1]
```

```

u, c = least_squares(f, psi, Omega)
comparison_plot(f, u, Omega)
u, c = interpolation(f, psi, points)
comparison_plot(f, u, Omega)

```

Figure 3.10 shows the results. There is a difference between the least squares and the interpolation technique but the difference decreases rapidly with increasing N .

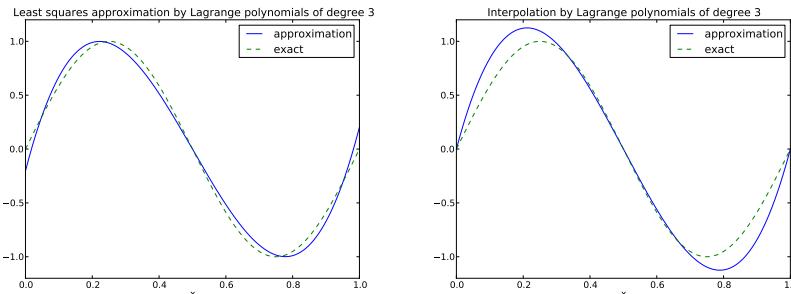


Fig. 3.10 Approximation via least squares (left) and interpolation (right) of a sine function by Lagrange interpolating polynomials of degree 3.

Less successful example. The next example concerns interpolating $f(x) = |1 - 2x|$ on $\Omega = [0, 1]$ using Lagrange polynomials. Figure 3.11 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 3.12 shows two such polynomials, $\psi_2(x)$ and $\psi_7(x)$, both of degree 11 and computed from uniformly spaced points $x_i = i/11$, $i = 0, \dots, 11$, marked with circles. We clearly see the property of Lagrange polynomials: $\psi_2(x_i) = 0$ and $\psi_7(x_i) = 0$ for all i , except $\psi_2(x_2) = 1$ and $\psi_7(x_7) = 1$. The most striking feature, however, is the dominating oscillation near the boundary where $\psi_2 > 5$ and $\psi_7 = -10$ in some points. The reason is easy to understand: since we force the functions to be zero at so many points, a polynomial of high degree is forced to oscillate between the points. This is called *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*:

$$x_i = \frac{1}{2}(a + b) + \frac{1}{2}(b - a) \cos\left(\frac{2i + 1}{2(N + 1)}\pi\right), \quad i = 0, \dots, N, \quad (3.60)$$

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, sym.Symbol):
            h = sym.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)
    psi = [Lagrange_polynomial(x, i, points) for i in range(N+1)]
    return psi, 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 3.13 shows the improvement of using Chebyshev nodes, compared with the equidistant points in Figure 3.11. The reason for this improvement is that the corresponding Lagrange polynomials have much smaller oscillations, which can be seen by comparing Figure 3.14 (Chebyshev points) with Figure 3.12 (equidistant points). Note the different scale on the vertical axes in the two figures and also that the Chebyshev points tend to cluster more around the element boundaries.

Another cure for undesired oscillations of higher-degree interpolating polynomials is to use lower-degree Lagrange polynomials on many small patches of the domain. This is actually 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.

How does the least squares or projection methods work with Lagrange polynomials? We can just call the `least_squares` function, but `sympy` has problems integrating the $f(x) = |1 - 2x|$ function times a polynomial, so we need to fall back on numerical integration.

```
def least_squares(f, psi, Omega):
    N = len(psi) - 1
    A = sym.zeros(N+1, N+1)
    b = sym.zeros(N+1, 1)
    x = sym.Symbol('x')
```

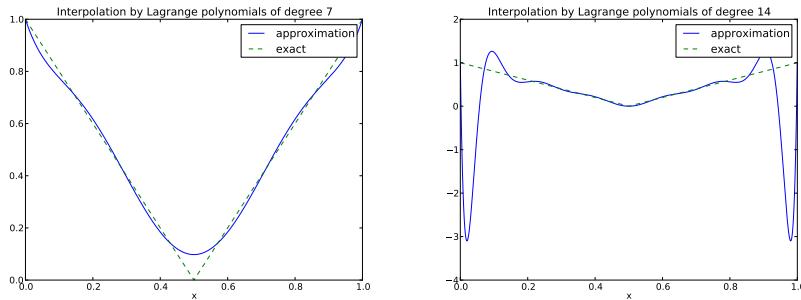


Fig. 3.11 Interpolation of an absolute value function by Lagrange polynomials and uniformly distributed interpolation points: degree 7 (left) and 14 (right).

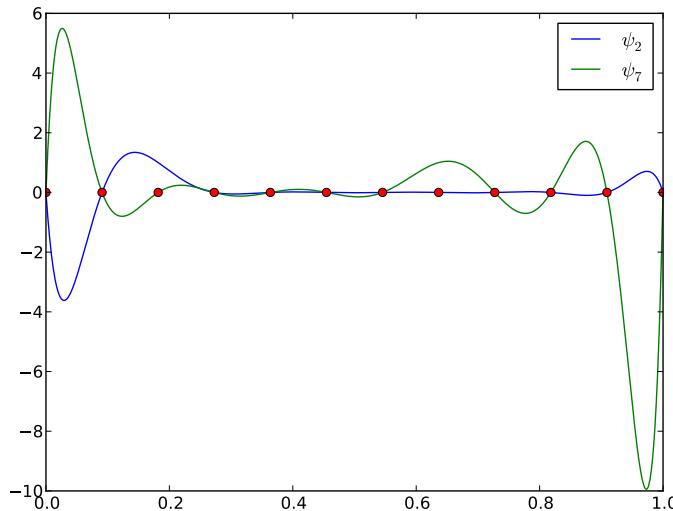


Fig. 3.12 Illustration of the oscillatory behavior of two Lagrange polynomials based on 12 uniformly spaced points (marked by circles).

```

for i in range(N+1):
    for j in range(i, N+1):
        integrand = psi[i]*psi[j]
        I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
        if isinstance(I, sym.Integral):
            # Could not integrate symbolically, fall back
            # on numerical integration with mpmath.quad
            integrand = sym.lambdify([x], integrand, 'mpmath')
            I = mpmath.quad(integrand, [Omega[0], Omega[1]])
        A[i,j] = A[j,i] = I
    integrand = psi[i]*f
    I = sym.integrate(integrand, (x, Omega[0], Omega[1]))

```

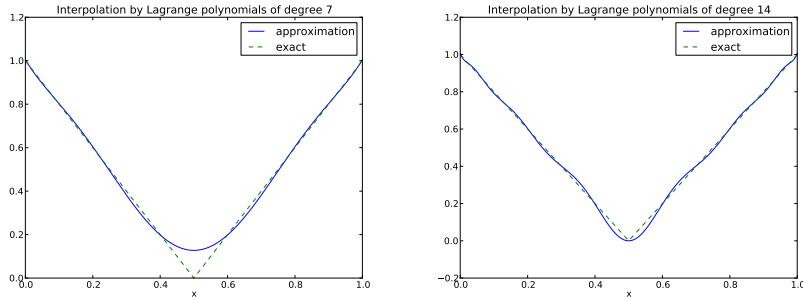


Fig. 3.13 Interpolation of an absolute value function by Lagrange polynomials and Chebyshev nodes as interpolation points: degree 7 (left) and 14 (right).

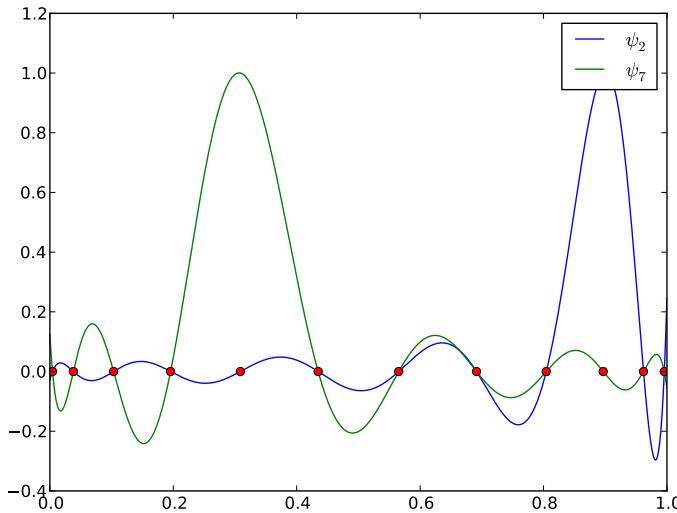


Fig. 3.14 Illustration of the less oscillatory behavior of two Lagrange polynomials based on 12 Chebyshev points (marked by circles). Note that the y-axis is different from Figure 3.12.

```

if isinstance(I, sym.Integral):
    integrand = sym.lambdify([x], integrand, 'mpmath')
    I = mpmath.quad(integrand, [Omega[0], Omega[1]])
    b[i,0] = I
c = A.LUsolve(b)
c = [sym.simplify(c[i,0]) for i in range(c.shape[0])]
u = sum(c[i]*psi[i] for i in range(len(psi)))
return u, c

```

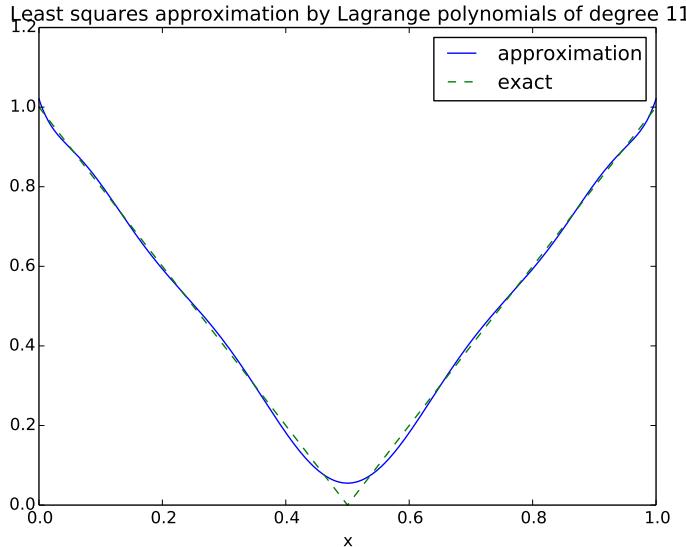


Fig. 3.15 Illustration of an approximation of the absolute value function using the least square method .

3.4.3 Bernstein polynomials

An alternative to the Taylor and Lagrange families of polynomials are the Bernstein polynomials. These polynomials are popular in visualization and we include a presentation of them for completeness. Furthermore, as we will demonstrate, the choice of basis functions may matter in terms of accuracy and efficiency. In fact, in finite element methods, a main challenge, from a numerical analysis point of view, is to determine appropriate basis functions for a particular purpose or equation.

On the unit interval, the Bernstein polynomials are defined in terms of powers of x and $1 - x$ (the barycentric coordinates of the unit interval) as

$$B_{i,n} = \binom{n}{i} x^i (1-x)^{n-i}, \quad i = 0, \dots, n. \quad (3.61)$$

The Figure 3.16 shows the basis functions of a Bernstein basis of order 8. This figure should be compared against Figure 3.17, which shows the corresponding Lagrange basis of order 8. The Lagrange basis is convenient because it is a nodal basis, that is; the basis functions are 1 in their nodal points and zero at all other nodal points as described by (3.57). However, looking at Figure 3.17 we also notice that the basis function

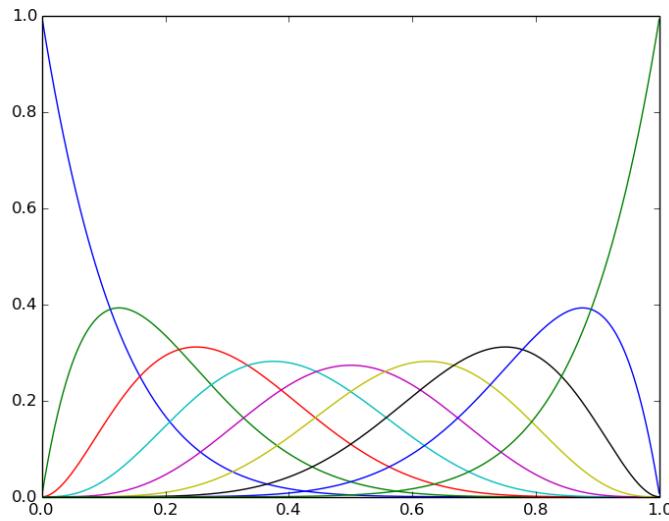


Fig. 3.16 The nine functions of a Bernstein basis of order 8.

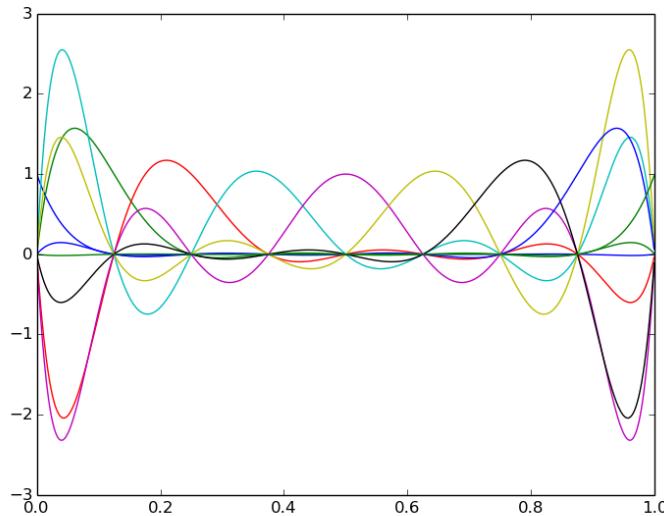


Fig. 3.17 The nine functions of a Lagrange basis of order 8.

are oscillatory and have absolute values that are significantly larger than 1 between the nodal points. Consider for instance the basis function

represented by the purple color. It is 1 at $x = 0.5$ and 0 at all other nodal points and hence this basis function represents the value at the mid-point. However, this function also has strong negative contributions close to the element boundaries where it takes negative values lower than -2 . For the Bernstein basis, all functions are positive and all functions output values in $[0, 1]$. Therefore there is no oscillatory behavior. The main disadvantage of the Bernstein basis is that the basis is not a nodal basis. In fact, all functions contribute everywhere except $x = 0$ and $x = 1$.

Both Lagrange and Bernstein polynomials take larger values towards the element boundaries than in the middle of the element, but the Bernstein polynomials always remain less than or equal to 1.

We remark that the Bernstein basis is easily extended to polygons in 2D and 3D in terms of the barycentric coordinates. For example, consider the reference triangle in 2D consisting of the faces $x = 0$, $y = 0$, and $x + y = 1$. The barycentric coordinates are $b_1(x, y) = x$, $b_2(x, y) = y$, and $b_3(x, y) = 1 - x - y$ and the Bernstein basis functions of order n is of the form

$$B_{i,j,k} = \frac{n!}{i!j!k!} x^i y^j (1 - x - y)^k, \quad \text{for } i + j + k = n.$$

Notice

We have considered approximation with a sinusoidal basis and with Lagrange or Bernstein polynomials, all of which are frequently used for scientific computing. The Lagrange polynomials (of various order) are extensively used in finite element methods, while the Bernstein polynomials are more used in computational geometry. However, we mention a few recent efforts in finite element computations that explore the combination of symbolic and numerical evaluation for finite element methods and have demonstrated that the Bernstein basis enables fast computations through their explicit representation (of both the basis functions and their derivatives) [1, 15]. The Lagrange and the Bernstein families are, however, but a few in the jungle of polynomial spaces used for finite element computations and their efficiency and accuracy can vary quite substantially. Furthermore, while a method may be efficient and accurate for one type of PDE it might not even converge for another type of PDE. The

development and analysis of finite element methods for different purposes is currently an intense research field and has been so for several decades. Some structure in this vast jungle of methods can be found in [2]. FEniCS has implemented a wide range of polynomial spaces [16] and has a general framework for implementing new elements [17]. While finite element methods explore different families of polynomials, the so-called spectral methods explore the use of sinusoidal functions or polynomials with high order. From an abstract point of view, the different methods can all be obtained simply by changing the basis for the trial and test functions. However, their efficiency and accuracy may vary substantially, as we will also see in the following.

3.5 Approximation properties and convergence rates

We will now compare the different approximation methods in terms of accuracy and efficiency. We consider four different series for generating approximations: Taylor, Lagrange, sinusoidal, and Bernstein. For all families we expect that the approximations improve as we increase the number of basis functions in our representations. We also expect that the computational complexity increases. Let us therefore try to quantify the accuracy and efficiency of the different methods in terms of the number of basis functions N . In the present example we consider the least squares method.

Let us consider the approximation of a Gaussian bell function, i.e., that the exact solution is

$$u_e = \exp(-(x - 0.5)^2) - \exp(-0.5^2)$$

We remark that u_e is zero at $x = 0$ and $x = 1$ and that we have chosen the bell function because it cannot be expressed as a finite sum of either polynomials or sines. We may therefore study the behavior as $N \rightarrow \infty$.

To quantify the behavior of the error as well as the complexity of the computations we compute the approximation with an increasing number of basis functions and time the computations by using `time.clock` (returning the CPU time so far in the program). A code example goes as follows:

```
def convergence_rate_analysis(series_type, func):
```

```

N_values = [2, 4, 8, 16]
norms = []
cpu_times = []
for N in N_values:

    psi = series(series_type, N)
    t0 = time.clock()
    u, c = least_squares_non_verbose(
        gauss_bell, psi, Omega, False)
    t1 = time.clock()

    error2 = sym.lambdify([x], (func - u)**2)
    L2_norm = scipy.integrate.quad(error2, Omega[0], Omega[1])
    L2_norm = scipy.sqrt(L2_norm)
    norms.append(L2_norm[0])
    cpu_times.append(t1-t0)

return N_values, norms, cpu_times

```

We run the analysis as follows

```

Omega = [0, 1]
x = sym.Symbol("x")
gaussian_bell = sym.exp(-(x-0.5)**2) - sym.exp(-0.5**2)
step = sym.Piecewise((1, 0.25 < x), (0, True)) - \
    sym.Piecewise((1, 0.75 < x), (0, True))
func = gaussian_bell

import pylab as plt
series_types = ["Taylor", "Sinusoidal", "Bernstein", "Lagrange"]
for series_type in series_types:
    N_values, norms, cpu_times = \
        convergence_rate_analysis(series_type, func)
    plt.loglog(N_values, norms)
plt.show()

```

and the different families of basis functions are:

```

def Lagrange_series(N):
    psi = []
    h = 1.0/N
    points = [i*h for i in range(N+1)]
    for i in range(len(points)):
        p = 1
        for k in range(len(points)):
            if k != i:
                p *= (x - points[k])/(points[i] - points[k])
        psi.append(p)
    return psi

def Bernstein_series(N):
    psi = []
    for k in range(0,N+1):
        psi_k = sym.binomial(N, k)*x**k*(1-x)**(N-k)

```

```

    psi.append(psi_k)
    return psi

def Sinusoidal_series(N):
    psi = []
    for k in range(1,N):
        psi_k = sym.sin(sym.pi*k*x)
        psi.append(psi_k)
    return psi

def Taylor_series(N):
    psi = []
    for k in range(1,N):
        psi_k = x**k
        psi.append(psi_k)
    return psi

def series(series_type, N):
    if   series_type== "Taylor"      : return Taylor_series(N)
    elif series_type== "Sinusoidal"  : return Sinusoidal_series(N)
    elif series_type== "Bernstein"   : return Bernstein_series(N)
    elif series_type== "Lagrange"    : return Lagrange_series(N)
    else: print("series type unknown ")

```

Below we list the numerical error for different N when approximating the Gaussian bell function.

N	2	4	8	16
Taylor	9.83e-02	2.63e-03	7.83e-07	3.57e-10
sine	2.70e-03	6.10e-04	1.20e-04	2.17e-05
Bernstein	2.10e-03	4.45e-05	8.73e-09	4.49e-15
Lagrange	2.10e-03	4.45e-05	8.73e-09	2.45e-12

It is quite clear that the different methods have different properties. For example, the Lagrange basis for $N = 16$ is 145 times more accurate than the Taylor basis. However, Bernstein is actually more than 500 times more accurate than the Lagrange basis! The approximations obtained by sines are far behind the polynomial approximations for $N > 4$.

The corresponding CPU times for the required computations also vary quite a bit:

N	2	4	8	16
Taylor	0.0123	0.0325	0.108	0.441
sine	0.0113	0.0383	0.229	1.107
Bernstein	0.0384	0.1100	0.3368	1.187
Lagrange	0.0807	0.3820	2.5233	26.52

Here, the timings are in seconds. The Taylor basis is the most efficient and is in fact more than 60 times faster than the Lagrange basis for $N = 16$ (with our naive implementation of basic formulas).

In order to get a more precise idea of how the error of our different approximation methods behave as N increases, we investigate two simple data models which may be used in a regression analysis. The error is modeled in terms of either a polynomial or an exponential function defined as follows

$$E_1(N) = \alpha_1 N^{\beta_1}, \quad (3.62)$$

$$E_2(N) = \alpha_2 \exp(\beta_2 N). \quad (3.63)$$

Taking the logarithm of (3.62) we obtain

$$\log(E_1(N)) = \beta_1 \log(N) + \log(\alpha_1).$$

Hence, letting $x = \log(N)$ be the independent variable and $y = \log(E_1(N))$ the dependent one, we simply have the straight line $y = ax + b$ with $a = \beta_1$ and $b = \log(\alpha_1)$. Then, we may perform a regression analysis as earlier with respect to the basis functions $(1, x)$ and obtain an estimate of the order of convergence in terms of β_1 . For the second model (3.63), we take the natural logarithm and obtain

$$\ln(E_2(N)) = \beta_2 N + \ln(\alpha_2).$$

Again, regression analysis provides the means to estimate the convergence, but here we let $x = N$ be the independent variable, $y = \ln(E_2(N))$, $a = \beta_2$ and $b = \ln(\alpha_2)$. To summarize, the polynomial model should have the data around a straight line in a log-log plot, while the exponential model has its data around a straight line in a log plot with the logarithmic scale on the y axis.

Before we perform the regression analysis, a good rule is to inspect the behavior visually in log and log-log plots. Figure 3.18 shows a log-log plot of the error with respect to N for the various methods. Clearly, the sinusoidal basis seems to have a polynomial convergence rate as the log-log plot is linear. The Bernstein, Lagrange, and Taylor methods appear to have convergence that is faster than polynomial. It is then interesting to consider a log plot and see if the behavior is exponential. Figure 3.19 is a log plot. Here, the Bernstein approximation appears to be linear which suggests that the convergence is exponential.

The following program computes the order of convergence for the sines using the polynomial model (3.62) while the Bernstein approximation is estimated in terms of model (3.63). We avoid computing estimates for

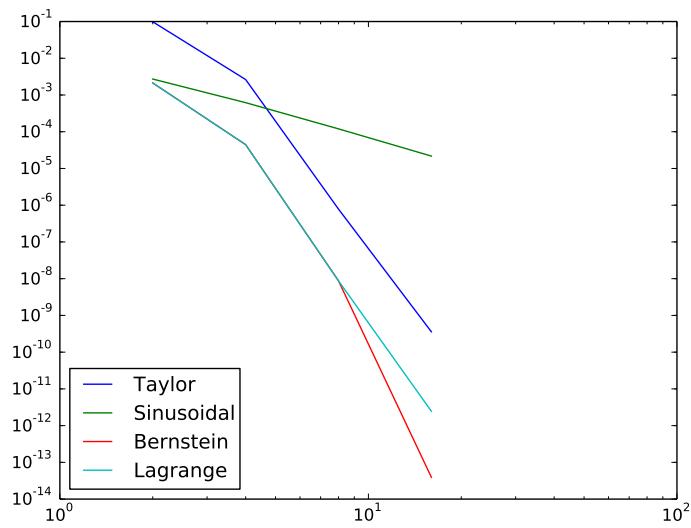


Fig. 3.18 Convergence of least square approximation using basis function in terms of the Taylor, sinusoidal, Bernstein and Lagrange basis in a log-log plot.

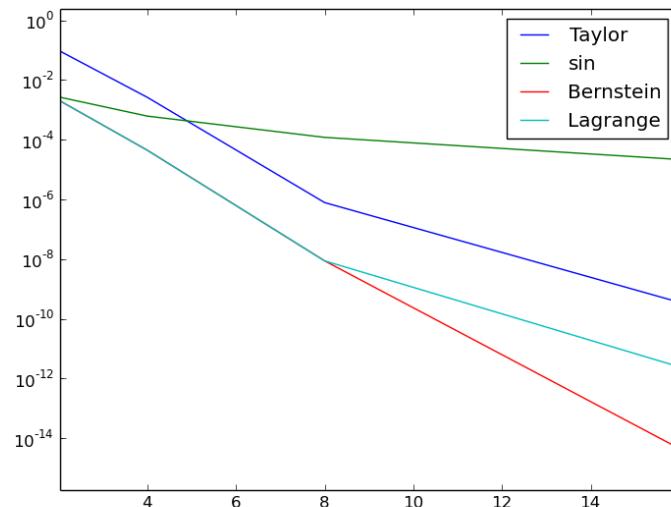


Fig. 3.19 Convergence of least square approximation using basis function in terms of the Taylor, sinusoidal, Bernstein and Lagrange basis in a log plot.

the Taylor and Lagrange approximations as neither the log-log plot nor the log plot demonstrated linear behavior.

```

N_values = [2, 4, 8, 16, 32]
Taylor    = [0.0983, 0.00263, 7.83e-07, 3.57e-10]
Sinusoidal = [0.0027, 0.00061, 0.00012, 2.17e-05]
Bernstein  = [0.0021, 4.45e-05, 8.73e-09, 4.49e-15]
Lagrange   = [0.0021, 4.45e-05, 8.73e-09, 2.45e-12]

x = sym.Symbol('x')
psi = [1, x]

u, c = regression_with_noise(log2(Sinusoidal), psi, log2(N_values))
print("estimated model for sine: %3.2e*N**(%3.2e)" % \
      (2**c[0]), c[1]))

# Check the numbers estimated by the model by manual inspection
for N in N_values:
    print(2**c[0] * N**c[1])

u, c = regression_with_noise(log(Bernstein), psi, N_values)
print("estimated model for Bernstein: %3.2e*exp(%3.2e*N)" % \
      (exp(c[0]), c[1]))

# Check the numbers estimated by the model by manual inspection
for N in N_values:
    print(exp(c[0]) * exp(N * c[1]))

```

The program estimates the sinusoidal approximation convergences as $1.410^{-2}N^{-2.3}$, which means that the convergence is slightly above second order. The Bernstein approximation on the other hand is $8.0110^{-2}\exp(-1.9N)$. Considering now that we have $N = 100$ then we can estimate that the sinusoidal approximation would give us an error of $\approx 3.610^{-7}$ while the estimate for the Bernstein polynomials amounts to $\approx 3.310^{-85}$ and is hence vastly superior. We remark here that floating point errors likely will be an issue, but libraries with arbitrary precision are available in Python.

The CPU time in the example here would be significantly faster if the algorithms were implemented in a compiled language like C/C++ or Fortran and we should be careful in drawing conclusions about the efficiency of the different methods based on this example alone. However, for completeness we include a log-log plot in Figure 3.20 to illustrate the polynomial increase in CPU time with respect to N . It seems that the efficiency of both the Taylor and Bernstein approximations can be estimated to be of the order of N^2 , but the sinusoidal and Lagrange approximations seem to grow faster.

The complete code can be found in [convergence_rate_local.py](#).

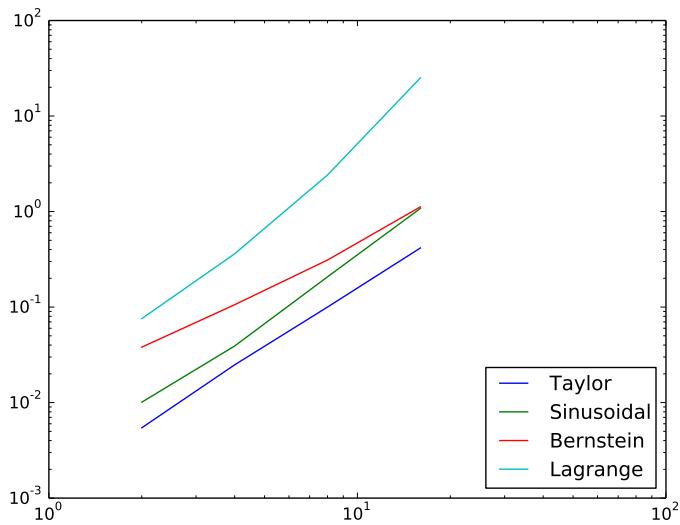


Fig. 3.20 CPU timings of the approximation with the difference basis in a log-log plot.

The code for the regression algorithm is as follows:

```
def regression_with_noise(f, psi, points):
    """
    Given a data points in the array f, return the approximation
    to the data in the space V, spanned by psi, using a regression
    method based on f and the corresponding coordinates in points.
    Must have len(points) = len(f) > len(psi).
    """
    N = len(psi) - 1
    m = len(points) - 1
    # Use numpy arrays and numerical computing
    B = np.zeros((N+1, N+1))
    d = np.zeros(N+1)
    # Wrap psi and f in Python functions rather than expressions
    # so that we can evaluate psi at points[i]
    x = sym.Symbol('x')
    psi_sym = psi # save symbolic expression for u
    psi = [sym.lambdify([x], psi[i]) for i in range(N+1)]
    if not isinstance(f, np.ndarray):
        raise TypeError('f is %s, must be ndarray' % type(f))
    print('...evaluating matrix...')
    for i in range(N+1):
        for j in range(N+1):
            B[i,j] = 0
            for k in range(m+1):
                B[i,j] += psi[i](points[k])*psi[j](points[k])
    # Solve the system of linear equations
    d = np.linalg.solve(B, f)
    # Create the approximation function
    def approx(x):
        return sum(d[i]*psi[i](x) for i in range(N+1))
    return approx
```

```

d[i] = 0
for k in range(m+1):
    d[i] += psi[i](points[k])*f[k]
print('B:\n', B, '\nd:\n', d)
c = np.linalg.solve(B, d)
print('coeff:', c)
u = sum(c[i]*psi_sym[i] for i in range(N+1))
print('approximation:', sym.simplify(u))

```

3.6 Approximation of functions in higher dimensions

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 consist of defining basis functions $\psi_i(x, y)$ or $\psi_i(x, y, z)$ over some domain Ω , and for the least squares and Galerkin methods, the integration is done over Ω .

As in 1D, the least squares and projection/Galerkin methods lead to linear systems

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s,$$

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

$$b_i = (f, \psi_i),$$

where the inner product of two functions $f(x, y)$ and $g(x, y)$ is defined completely analogously to the 1D case (3.25):

$$(f, g) = \int_{\Omega} f(x, y)g(x, y)dxdy. \quad (3.64)$$

3.6.1 2D basis functions as tensor products of 1D functions

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

$$V_x = \text{span}\{\hat{\psi}_0(x), \dots, \hat{\psi}_{N_x}(x)\}. \quad (3.65)$$

A similar space for a function's variation in y can be defined,

$$V_y = \text{span}\{\hat{\psi}_0(y), \dots, \hat{\psi}_{N_y}(y)\}. \quad (3.66)$$

We can then form 2D basis functions as *tensor products* of 1D basis functions.

Tensor products

Given two vectors $a = (a_0, \dots, a_M)$ and $b = (b_0, \dots, b_N)$, their *outer tensor product*, also called the *dyadic product*, is $p = a \otimes b$, defined through

$$p_{i,j} = a_i b_j, \quad i = 0, \dots, M, \quad j = 0, \dots, N.$$

In the tensor terminology, a and b are first-order tensors (vectors with one index, also termed rank-1 tensors), and then their outer tensor product is a second-order tensor (matrix with two indices, also termed rank-2 tensor). The corresponding *inner tensor product* is the well-known scalar or dot product of two vectors: $p = a \cdot b = \sum_{j=0}^N a_j b_j$. Now, p is a rank-0 tensor.

Tensors are typically represented by arrays in computer code. In the above example, a and b are represented by one-dimensional arrays of length M and N , respectively, while $p = a \otimes b$ must be represented by a two-dimensional array of size $M \times N$.

Tensor products can be used in a variety of contexts.

Given the vector spaces V_x and V_y as defined in (3.65) and (3.66), the tensor product space $V = V_x \otimes V_y$ has a basis formed as the tensor product of the basis for V_x and V_y . That is, if $\{\psi_i(x)\}_{i \in \mathcal{I}_x}$ and $\{\psi_i(y)\}_{i \in \mathcal{I}_y}$ are basis for V_x and V_y , respectively, the elements in the basis for V arise from the tensor product: $\{\psi_i(x)\psi_j(y)\}_{i \in \mathcal{I}_x, j \in \mathcal{I}_y}$. The index sets are $I_x = \{0, \dots, N_x\}$ and $I_y = \{0, \dots, N_y\}$.

The notation for a basis function in 2D can employ a double index as in

$$\psi_{p,q}(x, y) = \hat{\psi}_p(x)\hat{\psi}_q(y), \quad p \in \mathcal{I}_x, q \in \mathcal{I}_y.$$

The expansion for u is then written as a double sum

$$u = \sum_{p \in \mathcal{I}_x} \sum_{q \in \mathcal{I}_y} c_{p,q} \psi_{p,q}(x, y).$$

Alternatively, we may employ a single index,

$$\psi_i(x, y) = \hat{\psi}_p(x)\hat{\psi}_q(y),$$

and use the standard form for u ,

$$u = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x, y).$$

The single index can be expressed in terms of the double index through $i = p(N_y + 1) + q$ or $i = q(N_x + 1) + p$.

3.6.2 Example on polynomial basis in 2D

Let us again consider an approximation with the least squares method, but now with basis functions in 2D. Suppose we choose $\hat{\psi}_p(x) = x^p$, and try an approximation with $N_x = N_y = 1$:

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

Using a mapping to one index like $i = q(N_x + 1) + p$, we get

$$\psi_0 = 1, \quad \psi_1 = x, \quad \psi_2 = y, \quad \psi_3 = xy.$$

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

$$\begin{aligned}
A_{0,0} &= (\psi_0, \psi_0) = \int_0^{L_y} \int_0^{L_x} 1 dx dy = L_x L_y, \\
A_{0,1} &= (\psi_0, \psi_1) = \int_0^{L_y} \int_0^{L_x} x dx dy = \frac{L_x^2 L_y}{2}, \\
A_{0,2} &= (\psi_0, \psi_2) = \int_0^{L_y} \int_0^{L_x} y dx dy = \frac{L_x L_y^2}{2}, \\
A_{0,3} &= (\psi_0, \psi_3) = \int_0^{L_y} \int_0^{L_x} xy dx dy = \frac{L_x^2 L_y^2}{4}, \\
A_{1,0} &= (\psi_1, \psi_0) = \int_0^{L_y} \int_0^{L_x} x^2 dx dy = \frac{L_x^3 L_y}{2}, \\
A_{1,1} &= (\psi_1, \psi_1) = \int_0^{L_y} \int_0^{L_x} x^2 dx dy = \frac{L_x^3 L_y}{3}, \\
A_{1,2} &= (\psi_1, \psi_2) = \int_0^{L_y} \int_0^{L_x} xy dx dy = \frac{L_x^2 L_y^2}{4}, \\
A_{1,3} &= (\psi_1, \psi_3) = \int_0^{L_y} \int_0^{L_x} x^2 y dx dy = \frac{L_x^3 L_y^2}{6}, \\
A_{2,0} &= (\psi_2, \psi_0) = \int_0^{L_y} \int_0^{L_x} y^2 dx dy = \frac{L_x L_y^3}{2}, \\
A_{2,1} &= (\psi_2, \psi_1) = \int_0^{L_y} \int_0^{L_x} xy dx dy = \frac{L_x^2 L_y^2}{4}, \\
A_{2,2} &= (\psi_2, \psi_2) = \int_0^{L_y} \int_0^{L_x} y^2 dx dy = \frac{L_x L_y^3}{3}, \\
A_{2,3} &= (\psi_2, \psi_3) = \int_0^{L_y} \int_0^{L_x} x y^2 dx dy = \frac{L_x^2 L_y^3}{6}, \\
A_{3,0} &= (\psi_3, \psi_0) = \int_0^{L_y} \int_0^{L_x} x y dx dy = \frac{L_x^2 L_y^2}{4}, \\
A_{3,1} &= (\psi_3, \psi_1) = \int_0^{L_y} \int_0^{L_x} x^2 y dx dy = \frac{L_x^3 L_y^2}{6}, \\
A_{3,2} &= (\psi_3, \psi_2) = \int_0^{L_y} \int_0^{L_x} x y^2 dx dy = \frac{L_x^2 L_y^3}{6}, \\
A_{3,3} &= (\psi_3, \psi_3) = \int_0^{L_y} \int_0^{L_x} x^2 y^2 dx dy = \frac{L_x^3 L_y^3}{9}.
\end{aligned}$$

The right-hand side vector has the entries

$$\begin{aligned}
b_0 &= (\psi_0, f) = \int_0^{L_y} \int_0^{L_x} 1 \cdot (1 + x^2)(1 + 2y^2) dx dy \\
&= \int_0^{L_y} (1 + 2y^2) dy \int_0^{L_x} (1 + x^2) dx = (L_y + \frac{2}{3}L_y^3)(L_x + \frac{1}{3}L_x^3) \\
b_1 &= (\psi_1, f) = \int_0^{L_y} \int_0^{L_x} x(1 + x^2)(1 + 2y^2) dx dy \\
&= \int_0^{L_y} (1 + 2y^2) dy \int_0^{L_x} x(1 + x^2) dx = (L_y + \frac{2}{3}L_y^3)(\frac{1}{2}L_x^2 + \frac{1}{4}L_x^4) \\
b_2 &= (\psi_2, f) = \int_0^{L_y} \int_0^{L_x} y(1 + x^2)(1 + 2y^2) dx dy \\
&= \int_0^{L_y} y(1 + 2y^2) dy \int_0^{L_x} (1 + x^2) dx = (\frac{1}{2}L_y^2 + \frac{1}{2}L_y^4)(L_x + \frac{1}{3}L_x^3) \\
b_3 &= (\psi_3, f) = \int_0^{L_y} \int_0^{L_x} xy(1 + x^2)(1 + 2y^2) dx dy \\
&= \int_0^{L_y} y(1 + 2y^2) dy \int_0^{L_x} x(1 + x^2) dx = (\frac{1}{2}L_y^2 + \frac{1}{2}L_y^4)(\frac{1}{2}L_x^2 + \frac{1}{4}L_x^4).
\end{aligned}$$

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

$$\begin{aligned}
A_{i,j} &= (\psi_i, \psi_j) = \int_0^{L_y} \int_0^{L_x} \psi_i \psi_j dx dy \\
&= \int_0^{L_y} \int_0^{L_x} \psi_{p,q} \psi_{r,s} dx dy = \int_0^{L_y} \int_0^{L_x} \hat{\psi}_p(x) \hat{\psi}_q(y) \hat{\psi}_r(x) \hat{\psi}_s(y) dx dy \\
&= \int_0^{L_y} \hat{\psi}_q(y) \hat{\psi}_s(y) dy \int_0^{L_x} \hat{\psi}_p(x) \hat{\psi}_r(x) dx \\
&= \hat{A}_{p,r}^{(x)} \hat{A}_{q,s}^{(y)},
\end{aligned}$$

where

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

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

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

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

and

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

for $p, r \in \mathcal{I}_x$ and $q, s \in \mathcal{I}_y$.

Corresponding reasoning for the right-hand side leads to

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

Choosing $L_x = L_y = 2$, we have

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

Figure 3.21 illustrates the result.

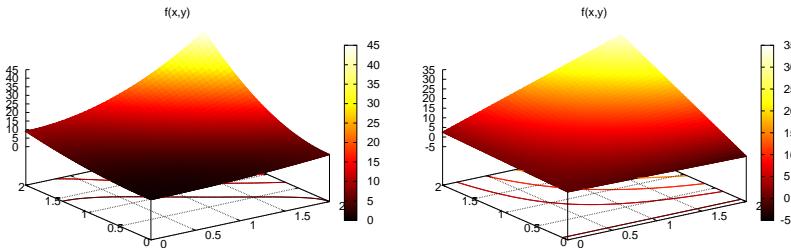


Fig. 3.21 Approximation of a 2D quadratic function (left) by a 2D bilinear function (right) using the Galerkin or least squares method.

The calculations can also be done using `sympy`. The following code computes the matrix of our example

```
import sympy as sym
```

```

x, y, Lx, Ly = sym.symbols("x y L_x L_y")

def integral(integrand):
    Ix = sym.integrate(integrand, (x, 0, Lx))
    I = sym.integrate(Ix, (y, 0, Ly))
    return I

basis = [1, x, y, x*y]
A = sym.Matrix(sym.zeros(4,4))

for i in range(len(basis)):
    psi_i = basis[i]
    for j in range(len(basis)):
        psi_j = basis[j]
        A[i,j] = integral(psi_i*psi_j)

```

We remark that `sympy` may even write the output in L^AT_EX or C++ format using the functions `latex` and `ccode`.

3.6.3 Implementation

The `least_squares` function from Section 3.3.3 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:

```

import sympy as sym

integrand = psi[i]*psi[j]
I = sym.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, sym.Integral):
    integrand = sym.lambdify([x,y], integrand, 'mpmath')
    I = 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. (We should add that `mpmath.quad` is sufficiently fast even in 2D, but `scipy.integrate.nquad` is much faster.)

Third, we must construct a list of 2D basis functions. Here are two examples based on tensor products of 1D "Taylor-style" polynomials x^i and 1D sine functions $\sin((i+1)\pi x)$:

```
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 [sym.sin(sym.pi*(i+1)*x)*sym.sin(sym.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

```
>>> from approx2D import *
>>> f = (1+x**2)*(1+2*y**2)
>>> psi = taylor(x, y, 1, 1)
>>> Omega = [[0, 2], [0, 2]]
>>> u, c = least_squares(f, psi, Omega)
>>> print(u)
8*x*y - 2*x/3 + 4*y/3 - 1/9
>>> print(sym.expand(f))
2*x**2*y**2 + x**2 + 2*y**2 + 1
```

We may continue with adding higher powers to the basis:

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

For $N_x \geq 2$ and $N_y \geq 2$ we recover the exact function f , as expected, since in that case $f \in V$, see Section 3.2.5.

3.6.4 Extension to 3D

Extension to 3D is in principle straightforward once the 2D extension is understood. The only major difference is that we need the repeated outer tensor product,

$$V = V_x \otimes V_y \otimes V_z.$$

In general, given vectors (first-order tensors) $a^{(q)} = (a_0^{(q)}, \dots, a_{N_q}^{(q)})$, $q = 0, \dots, m$, the tensor product $p = a^{(0)} \otimes \dots \otimes a^m$ has elements

$$p_{i_0, i_1, \dots, i_m} = a_{i_1}^{(0)} a_{i_1}^{(1)} \cdots a_{i_m}^{(m)}.$$

The basis functions in 3D are then

$$\psi_{p,q,r}(x, y, z) = \hat{\psi}_p(x)\hat{\psi}_q(y)\hat{\psi}_r(z),$$

with $p \in \mathcal{I}_x$, $q \in \mathcal{I}_y$, $r \in \mathcal{I}_z$. The expansion of u becomes

$$u(x, y, z) = \sum_{p \in \mathcal{I}_x} \sum_{q \in \mathcal{I}_y} \sum_{r \in \mathcal{I}_z} c_{p,q,r} \psi_{p,q,r}(x, y, z).$$

A single index can be introduced also here, e.g., $i = rN_x N_y + qN_x + p$, $u = \sum_i c_i \psi_i(x, y, z)$.

Use of tensor product spaces

Constructing a multi-dimensional space and basis from tensor products of 1D spaces is a standard technique when working with global basis functions. In the world of finite elements, constructing basis functions by tensor products is much used on quadrilaterals and hexahedra cell shapes, but not on triangles and tetrahedra. Also, the global finite element basis functions are almost exclusively denoted by a single index and not by the natural tuple of indices that arises from tensor products.

3.7 Exercises

Problem 3.1: Linear algebra refresher

Look up the topic of *vector space* in your favorite linear algebra book or search for the term at Wikipedia.

a) Prove that vectors in the plane spanned by the vector (a, b) form a vector space by showing that all the axioms of a vector space are satisfied.

Solution. The axioms of a vector space go as follows (see [Wikipedia](#), but we use slightly different notation below):

1. The sum of u and v , denoted by $u + v$, is in V .
2. $u + v = v + u$.
3. $(u + v) + w = u + (v + w)$.

4. There is a *zero* vector 0 in V such that $u + 0 = u$.
5. For each u in V , there is a vector $-u$ in V such that $u + (-u) = 0$.
6. The scalar multiple of u by γ , denoted by γu , is in V .
7. $\gamma(u + v) = \gamma u + \gamma v$.
8. $(\gamma + \delta)u = \gamma u + \delta u$ for scalar γ and δ .
9. $\gamma(\delta u) = \gamma\delta u$.
10. $1u = u$.

We must show that each axiom is fulfilled by planar vectors and their mathematical rules. Let $u = (a, b)$ and $v = (c, d)$.

Axiom 1:

$$u + v = (a, b) + (c, d) = (a + c, b + d),$$

is a planar vector and therefore in V .

Axiom 2:

$$u + v = (a, b) + (c, d) = (a + c, b + d) = (c + a, d + b) = v + u.$$

Axiom 3:

$$(u + v) + w = ((a, b) + (c, d)) + (e, f) = (a, b) + ((c, d) + (e, f)) = u + (v + w).$$

Axiom 4: The $(0, 0)$ vector is the 0 element,

$$u + 0 = (a, b) + (0, 0) = (a + 0, b + 0) = (a, b) = u.$$

Axiom 5: Let $-u$ element is $(-a, -b)$, so

$$u + (-u) = (a, b) + (-a, -b) = (0, 0) = 0.$$

Axiom 6:

$$\gamma u = \gamma \cdot (a, b) = (\gamma a, \gamma b),$$

is also a planar vector and therefore in V .

Axiom 7:

$$\gamma(u + v) = \gamma \cdot ((a, b) + (c, d)) = \gamma(a, b) + \gamma(c, d).$$

Axiom 8:

$$(\gamma + \delta)u = (\gamma + \delta)(a, b) = \gamma(a, b) + \delta(a, b) = \gamma u + \delta u .$$

Axiom 9:

$$\gamma(\delta u) = \gamma(\delta \cdot (a, b)) = \gamma(\delta a, \delta b) = (\gamma\delta a, \gamma\delta b) = \gamma\delta u .$$

Axiom 10:

$$1u = 1(a, b) = (1 \cdot a, 1 \cdot b) = (a, b) = u .$$

- b)** Prove that all linear functions of the form $ax + b$ constitute a vector space, $a, b \in \mathbb{R}$.

Solution. Let $u = ax + b$ and $v = cx + d$. We verify each axiom.

Axiom 1:

$$u + v = ax + b + cx + d = (a + c)x + (b + d),$$

is also a linear function and therefore in V .

Axiom 2:

$$u + v = ax + b + cx + d = cx + d + ax + b = v + u .$$

Axiom 3:

$$(u+v)+w = (ax+b+cx+d)+ex+f = ax+b+cx+d+ex+f+ax+b+(cx+d+ex+f) = u+(v+w) .$$

Axiom 4: The $0x + 0 = 0$ function is the 0 element,

$$u + 0 = ax + b + 0 = ax + b = u .$$

Axiom 5: Let $-u$ element is $-ax - b$, so

$$u + (-u) = ax + b + (-ax - b) = 0 .$$

Axiom 6:

$$\gamma u = \gamma(ax + b) = \gamma ax + \gamma b,$$

is also a linear function and therefore in V .

Axiom 7:

$$\gamma(u+v) = \gamma(ax+b+cx+d) = \gamma ax + \gamma b + \gamma cx + \gamma d = \gamma(ax+b) + \gamma(cx+d) = \gamma u + \gamma v .$$

Axiom 8:

$$(\gamma + \delta)u = (\gamma + \delta)(ax + b) = \gamma(ax + b) + \delta(ax + b) = \gamma u + \delta u.$$

Axiom 9:

$$\gamma(\delta u) = \gamma(\delta(ax + b)) = \gamma\delta u.$$

Axiom 10:

$$1u = 1(ax + b) = ax + b = u.$$

- c)** Show that all quadratic functions of the form $1 + ax^2 + bx$ do not constitute a vector space.

Solution. Let $u = ax^2 + bx + 1$ and $v = cx^2 + dx + 1$. We try to verify each axiom.

Axiom 1:

$$u + v = ax^2 + bx + 1 + cx^2 + dx + 1 = (a + c)x^2 + (b + d)x + 2,$$

but this quadratic function is not in V because the constant term is 2 and not 1. Consequently, quadratic functions of the particular form $1 + ax^2 + bx$ do not constitute a vector space. The more general form $ax^2 + bx + c$ for arbitrary constants a , b , and c makes functions that span a function space.

- d)** Check out the topic of *inner product spaces*. Suggest a possible inner product for the space of all linear functions of the form $ax + b$, $a, b \in \mathbb{R}$, defined on some interval $\Omega = [A, B]$. Show that this particular inner product satisfies the general requirements of an inner product in a vector space.

Solution. According to Wikipedia, an inner product space, with an inner product (u, v) , has three axioms (we drop the possibility of complex numbers and assume everything is real):

1. Symmetry: $(u, v) = (v, u)$
2. Linearity in the first argument: $\gamma u, v) = \gamma(u, v)$
3. Positive-definiteness: $(u, u) \geq 0$ and $(u, u) = 0$ implies $u = 0$

A possible inner product for linear functions on a domain Ω is

$$(u, v) = \int_A^B uv \, dx = \int_A^B (ax + b)(cx + d) \, dx .$$

Symmetry is obvious since $uv = vu$:

$$(u, v) = \int_A^B uv \, dx = \int_A^B vud = (v, u) .$$

Linearity in the first argument:

$$(\gamma u, v) = \gamma \int_A^B (\gamma u)v \, dx = \gamma \int_A^B uv \, dx = \gamma(u, v) .$$

Positive-definiteness:

$$(u, u) = \int_A^B (ax + b)^2 \, dx \geq 0,$$

since the integral of a function $f(x) \geq$ must be greater than or equal to zero, and in particular,

$$(u, u) = 0 \quad \Rightarrow \quad \int_A^B (ax + b)^2 \, dx = 0,$$

implies $ax + b = 0$.

Filename: linalg1.

Problem 3.2: Approximate a three-dimensional vector in a plane

Given $\mathbf{f} = (1, 1, 1)$ in \mathbb{R}^3 , find the best approximation vector \mathbf{u} in the plane spanned by the unit vectors $(1, 0)$ and $(0, 1)$. Repeat the calculations using the vectors $(2, 1)$ and $(1, 2)$.

Solution. We have the vector $\mathbf{f} = (1, 1, 1)$. Our aim is to approximate this with a vector in the vector-space spanned by the unit vectors ϕ_0 and ϕ_1 . We seek a solution $u = c_0\phi_0 + c_1\phi_1$. To do this we use the least-square-method and solve the equation $\mathbf{Ac} = \mathbf{b}$. Or written out:

$$\begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$

where $A_{i,j} = (\phi_i, \phi_j)$ and $b_i = (\phi_i, \mathbf{f})$.

We start with $\phi_0 = (0, 1)$, $\phi_1 = (1, 0)$. Calculations give

$$A_{0,0} = ([1, 0], [1, 0]) = 1,$$

$$A_{0,1} = ([1, 0], [0, 1]) = 0, A_{1,0} = ([0, 1], [1, 0]) = 0, A_{1,1} = ([0, 1], [0, 1]) = 1, b_0 = ([1, 0],$$

The result becomes $c_0 = c_1 = 1$, and hence

$$u = 1 \cdot \phi_0 + 1 \cdot \phi_1.$$

Then we proceed with $\phi_0 = (2, 1)$, $\phi_1 = (1, 2)$. Calculations give

$$A_{0,0} = ([2, 1], [2, 1]) = 2 \cdot 2 + 1 \cdot 1 = 5, A_{0,1} = ([2, 1], [1, 2]) = 2 \cdot 1 + 1 \cdot 2 = 4, A_{1,0} = ([1, 2], [2, 1]) = 1 \cdot 2 + 2 \cdot 1 = 4,$$

Solving for the c_i values results in $c_0 = c_1 = \frac{1}{3}$ and hence

$$u = \frac{1}{3} \cdot \phi_0 + \frac{1}{3} \cdot \phi_1.$$

Filename: `vec111_approx.`

Problem 3.3: Approximate a parabola by a sine

Given the function $f(x) = 1 + 2x(1 - x)$ on $\Omega = [0, 1]$, we want to find an approximation in the function space

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

a) Sketch or plot $f(x)$. Think intuitively how an expansion in terms of the basis functions of V , $\psi_0(x) = 1$, $\psi_1(x) = \sin(\pi x)$, can be constructed to yield a best approximation to f . Or phrased differently, see if you can guess the coefficients c_0 and c_1 in the expansion

$$u(x) = c_0\psi_0 + c_1\psi_1 = c_0 + c_1 \sin(\pi x).$$

Compute the L^2 error $\|f - u\|_{L^2} = (\int_0^1 (f - u)^2 dx)^{1/2}$.

Hint. If you make a mesh function `e` of the error on some mesh with uniformly spaced coordinates in the array `xc`, the integral can be approximated as `np.sqrt(dx*np.sum(e**2))`, where `dx=xc[0]-xc[1]` is the mesh spacing and `np` is an alias for the `numpy` module in Python.

Solution. The function $\psi_0 = 1$ can be used to “take care of” of the constant 1 in f , while $\psi_1 = \sin(\pi x)$ can approximate the parabola. The maximum of f is $3/2$, so we should use $u(x) = 1 \cdot \psi_0 + \frac{1}{2}\psi_1$.

Plotting and the computation of the L^2 error can be done by

```

import numpy as np
import matplotlib.pyplot as plt

xc = np.linspace(0, 1, 101) # x coordinates for plotting

def f(x):
    return 1 + 2*x*(1-x)

import sympy as sym
x = sym.symbols('x')
psi_0 = 1
psi_1 = sym.sin(sym.pi*x)

half = sym.Rational(1,2)
u = 1*psi_0 + half*psi_1

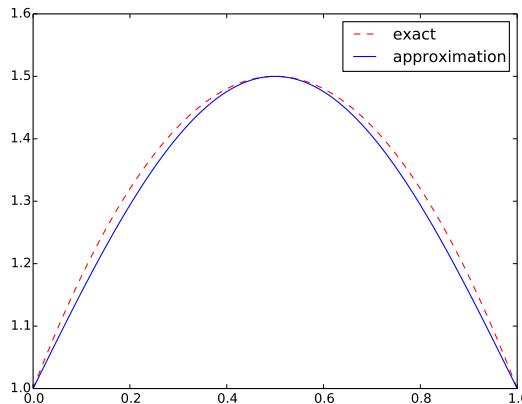
# How to combine c_0*psi_0 + c_1*psi_1 to match f?
# Intuitively, c_0=c_1=1...
# Turn u to function so we can plot and compute with it
u = sym.lambdify([x], u, modules='numpy')

print('L2 error of intuitive approximation:', end=' ')
e = f(xc) - u(xc)
dx = xc[1] - xc[0]
print(np.sqrt(dx*np.sum(e**2)))

plt.plot(xc, f(xc), 'r--')
plt.plot(xc, u(xc), 'b-')
plt.legend(['exact', 'intuitive approximation'])

```

The L^2 error becomes 0.0179.



- b)** Perform the hand calculations for a least squares approximation.

Solution. The least squares method ends up with a linear system where the coefficient matrix has entries $A_{i,j} = \int_0^1 \psi_i \psi_j dx$ and the right-hand side has entries $b_i = \int_0^1 \psi_i dx$. We can use `sympy` do carry out the integrals:

```
# Do the calculations in the least squares or project method
A = sym.zeros(2, 2)
b = sym.zeros(2, 1)
A[0,0] = sym.integrate(psi_0*psi_0, (x, 0, 1))
A[0,1] = sym.integrate(psi_0*psi_1, (x, 0, 1))
A[1,0] = A[0,1]
A[1,1] = sym.integrate(psi_1*psi_1, (x, 0, 1))
b[0] = sym.integrate(f(x)*psi_0, (x, 0, 1))
b[1] = sym.integrate(f(x)*psi_1, (x, 0, 1))
print('A:', A)
print('b:', b)
c = A.LUsolve(b)
c = [sym.simplify(c[i,0]) for i in range(c.shape[0])]
print('c:', c, [c_.evalf() for c_ in c])
u = c[0]*psi_0 + c[1]*psi_1
print('u:', u)
```

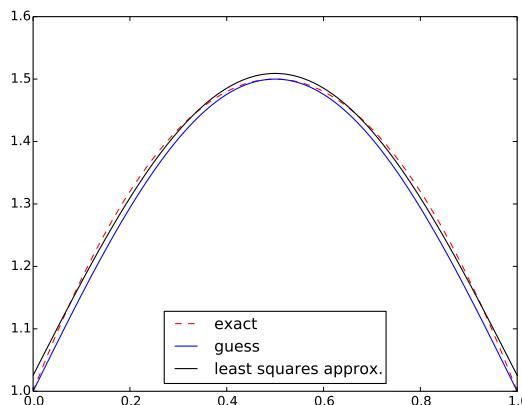
Looking at the results, we get the linear system

$$\begin{pmatrix} 1 & \frac{2}{\pi} \\ \frac{2}{\pi} & \frac{1}{2} \end{pmatrix} \begin{pmatrix} \frac{-24\pi^2 - 96 + 4\pi^4}{3\pi^2(-8 + \pi^2)} \\ \frac{-4\pi^2 + 48}{3\pi(-8 + \pi^2)} \end{pmatrix} = \begin{pmatrix} \frac{4}{3} \\ \frac{8}{\pi^3} + \frac{2}{\pi} \end{pmatrix}$$

The resulting best approximation reads

$$u(x) = \frac{4(-\pi^2 + 12)\sin(\pi x)}{3\pi(-8 + \pi^2)} + \frac{-24\pi^2 - 96 + 4\pi^4}{3\pi^2(-8 + \pi^2)} \approx 1.025 + 0.484\sin(\pi x)$$

The L^2 error turns out to be 0.00876. To conclude, the least squares method is slightly better than the intuition in this case.



Filename: `parabola_sin.`

Problem 3.4: Approximate the exponential function by power functions

Let V be a function space with basis functions x^i , $i = 0, 1, \dots, N$. Find the best approximation to $f(x) = \exp(-x)$ on $\Omega = [0, 8]$ among all functions in V for $N = 2, 4, 6$. Illustrate the three approximations in three separate plots.

Hint. Apply the `lest_squares` and `comparison_plot` functions in the `approx1D.py` module as these make the exercise easier to solve.

Solution. A suitable code is

```
from approx1D import least_squares, comparison_plot
import matplotlib.pyplot as plt
import sympy as sym
from math import factorial
import numpy as np

x = sym.Symbol('x')
f = sym.exp(-x)

Omega = [0, 8]

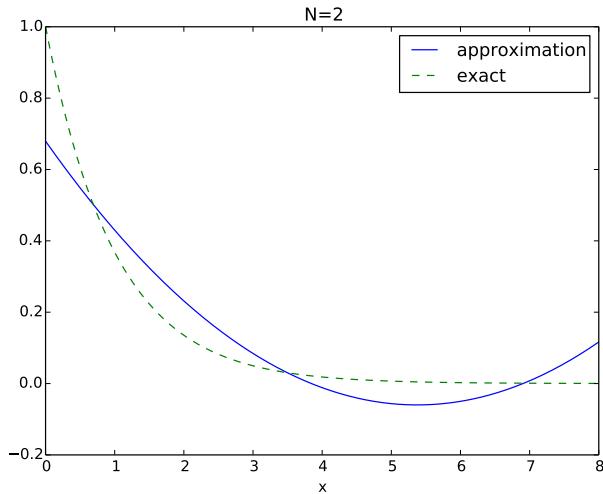
for N in 2,4,6:
    psi = [x**i for i in range(N+1)]
    u, c = least_squares(f,psi,Omega)
    print(N, u)
```

```
comparison_plot(f, u, Omega, filename='tmp_exp_%d' % N,
                 plot_title='N=%d' % N)
```

For the case $N = 2$ the program prints the following, here edited for clearer reading:

```
A:
Matrix([[8, 32, 512/3], [32, 512/3, 1024], [512/3, 1024, 32768/5]])
b:
Matrix([[[-exp(-8) + 1], [-9*exp(-8) + 1], [-82*exp(-8) + 2]]])

f: exp(-x)
u: x**2*(-465*exp(-8)/4096 + 105/4096) + x*(-141/512 +
405*exp(-8)/512) - 111*exp(-8)/128 + 87/128
```



Filename: `exp_powers.`

Problem 3.5: Approximate the sine function by power functions

In this exercise we want to approximate the sine function by polynomials of order $N + 1$. Consider two bases:

$$V_1 = \{x, x^3, x^5, \dots, x^{N-2}, x^N\},$$

$$V_2 = \{1, x, x^2, x^3, \dots, x^N\}.$$

The basis V_1 is motivated by the fact that the Taylor polynomial approximation to the sine function has only odd powers, while V_2 is motivated by the assumption that including the even powers could improve the approximation in a least-squares setting.

Compute the best approximation to $f(x) = \sin(x)$ among all functions in V_1 and V_2 on two domains of increasing sizes: $\Omega_{1,k} = [0, k\pi]$, $k = 2, 3, \dots, 6$ and $\Omega_{2,k} = [-k\pi/2, k\pi/2]$, $k = 2, 3, 4, 5$. Make plots for all combinations of V_1 , V_2 , Ω_1 , Ω_2 , $k = 2, 3, \dots, 6$.

Add a plot of the N -th degree Taylor polynomial approximation of $\sin(x)$ around $x = 0$.

Hint. You can make a loop over V_1 and V_2 , a loop over Ω_1 and Ω_2 , and a loop over k . Inside the loops, call the functions `least_squares` and `comparison_plot` from the `approx1D` module. $N = 7$ is a suggested value.

Solution. Suitable code is

```
import sympy as sym
from approx1D import least_squares, comparison_plot
from math import pi
import matplotlib.pyplot as plt

x = sym.Symbol('x')
f = sym.sin(x)
N = 7
psi_bases = [[x**i for i in range(1, N+1, 2)], # V_1
              [x**i for i in range(0, N+1)]]      # V_2
symbolic = False

for V, psi in enumerate(psi_bases):
    for domain_no in range(1, 3):
        for k in range(2, 6):
            if symbolic:
                Omega = [0, k*sym.pi] if domain_no == 1 else \
                         [-k*sym.pi/2, k*sym.pi/2]
            else:
                # cannot use sym.pi with numerical sympy computing
                Omega = [0, k*pi] if domain_no == 1 else \
                         [-k*pi/2, k*pi/2]

            u, c = least_squares(f, psi, Omega, symbolic=symbolic)

            comparison_plot(
                f, u, Omega,
                ymin=-2, ymax=2,
```

```

filename='tmp_N%d_V%dOmega%d' %
(N, V, k, domain_no),
plot_title='sin(x) on [0,%d*pi/2] by %s' %
(k, ','.join([str(p) for p in psi]))
# Need to kill the plot to proceed!
for ext in 'png', 'pdf':
    cmd = 'doconce combine_images -2 ' + \
        ' '.join(['tmp_N%d_V%dOmega%d.' %
            (N, V, k, domain_no) + ext
            for k in range(2, 6)]) + \
        ' sin_powers_N%d_V%d_Omega%d.' % (N, V, domain_no) + ext
    print(cmd)
    os.system(cmd)

# Show the standard Taylor series approximation
from math import factorial, pi
import time
Omega = [0, 12*pi/2.]
u = 0
for k in range(0,N+1):
    u = u + ((-1)**k*x**((1+2*k))/float(factorial(1+2*k)))
# Shorter: u = sum((-1)**k*x**((1+2*k))/float(factorial(1+2*k)))
# for k in range(0,10))
comparison_plot(f, u, Omega, 'sin_taylor%d' % k,
    ymin=-1.5, ymax=1.5)

```

The odd powers (V_1 space) behave not so good on $\Omega_{1,k}$, but better on $\Omega_{2,k}$:

Including also even powers (V_2 space) is clearly much better:

Comparison with a standard Taylor series shows that it is very inferior as an approximation over the entire domain, but much more accurate close to the origin (as expected, since the Taylor series is constructed with this property, while the least squares method tries to find a good approximation over the entire domain).

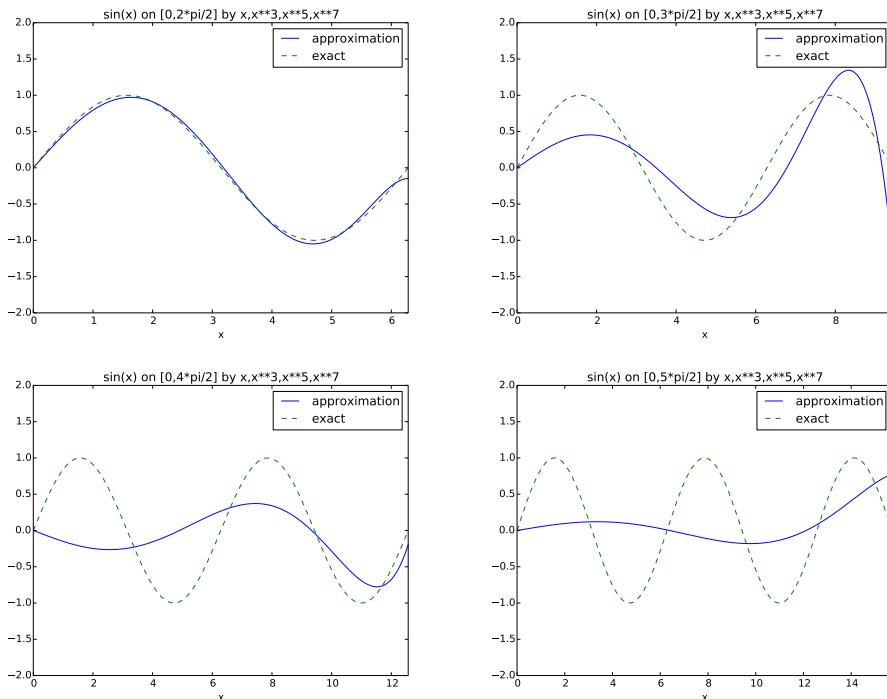
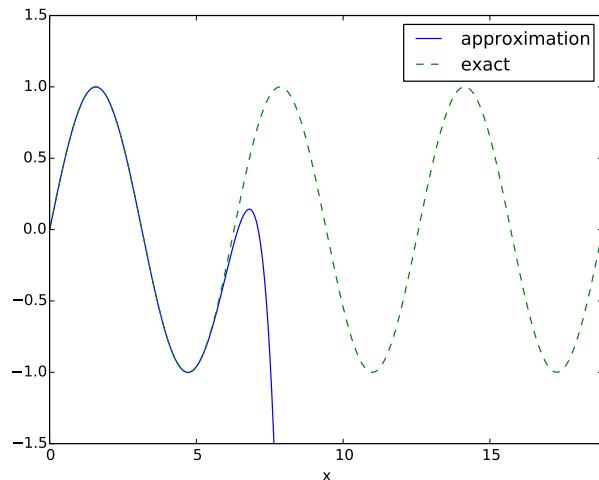


Fig. 3.22 V_1 space, $\Omega_{1,k}$ domain.



Filename: `sin_powers`.

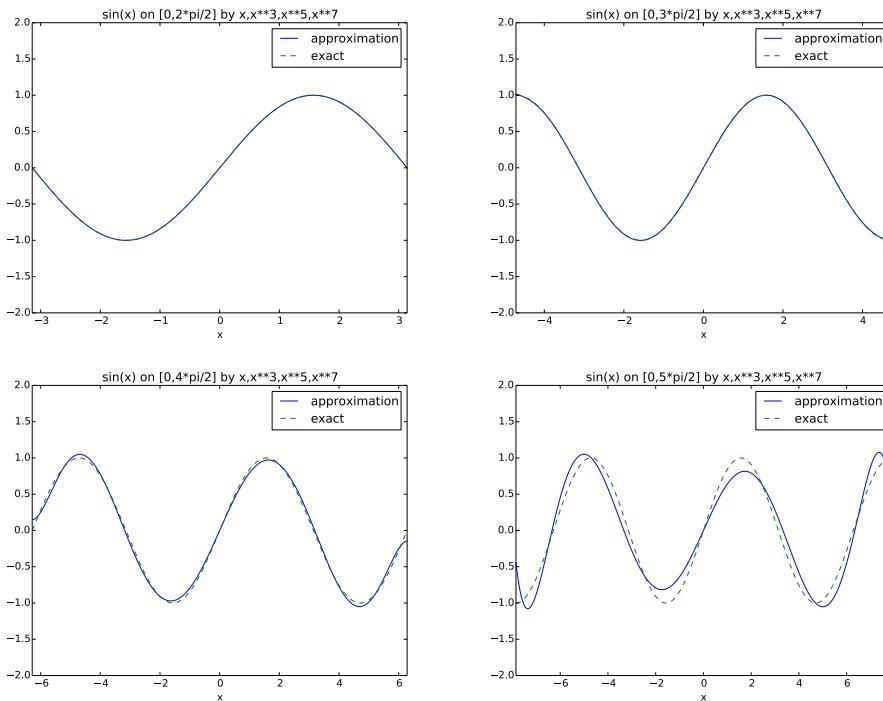


Fig. 3.23 V_1 space, $\Omega_{2,k}$ domain.

Problem 3.6: Approximate a steep function by sines

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

Hint 1. One may naively call the `least_squares_orth` and `comparison_plot` from the `approx1D` module in a loop and extend the basis with one new element in each pass. This approach implies a lot of recomputations. A more efficient strategy is to let `least_squares_orth` compute with only one basis function at a time and accumulate the corresponding u in the total solution.

Hint 2. `ffmpeg` or `avconv` may skip frames when plot files are combined to a movie. Since there are few files and we want to see each of them, use `convert` to make an animated GIF file (`-delay 200` is suitable).

Solution. The code may read

```
import sympy as sym
```

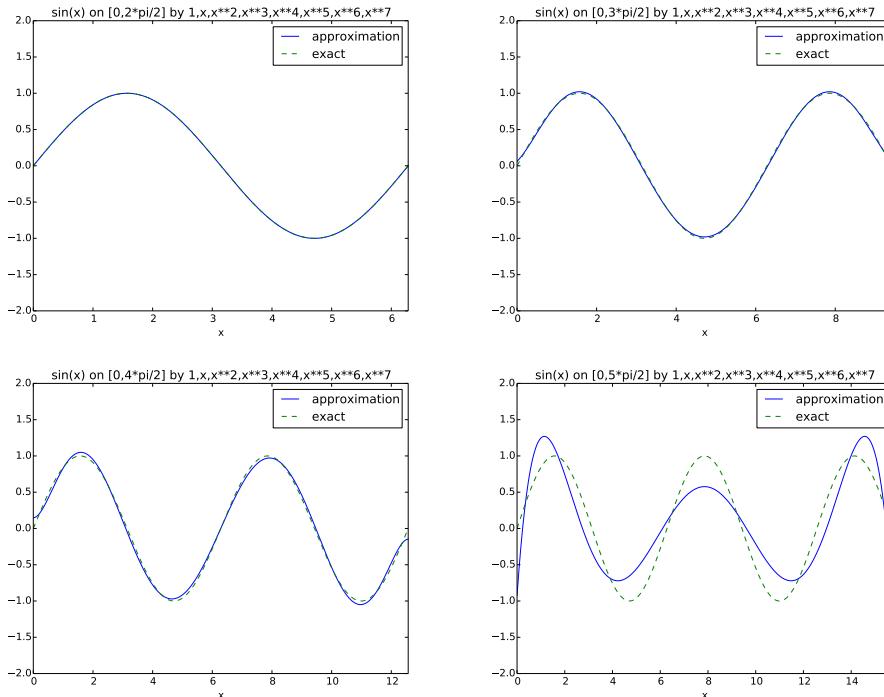


Fig. 3.24 V_2 space, $\Omega_{1,k}$ domain.

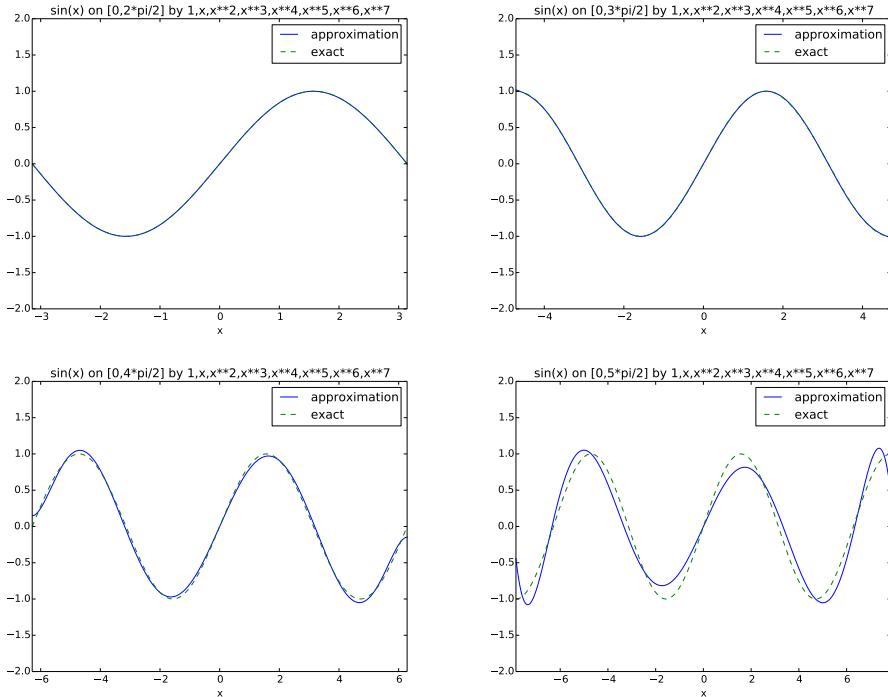


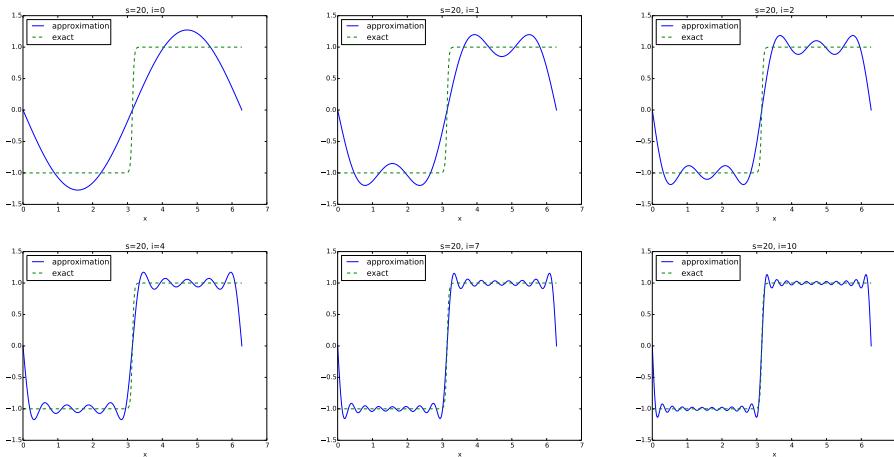
Fig. 3.25 V_2 space, $\Omega_{2,k}$ domain.

```

plot_title='s=%g, i=%d' % (s, i))

if __name__ == '__main__':
    s = 20 # steepness
    f = sym.tanh(s*(x-sym.pi))
    from math import pi
    Omega = [0, 2*pi] # sym.pi did not work here
    efficient(f, s, Omega, N=10)
    # Make movie
    # avconv/ffmpeg skips frames, use convert instead (few files)
    cmd = 'convert -delay 200 tmp_sin*.png tanh_sines_approx.gif'
    os.system(cmd)
    # Make static plots, 3 figures on 2 lines
    for ext in 'pdf', 'png':
        cmd = 'doconce combine_images %s -3' % ext
        cmd += 'tmp_sin00x tmp_sin01x tmp_sin02x tmp_sin04x '
        cmd += 'tmp_sin07x tmp_sin10x tanh_sines_approx'
        os.system(cmd)
    plt.show()

```



Filename: `tanh_sines`.

Remarks. Approximation of a discontinuous (or steep) $f(x)$ by sines, results in slow convergence and oscillatory behavior of the approximation close to the abrupt changes in f . This is known as the [Gibb's phenomenon](#).

Problem 3.7: Approximate a steep function by sines with boundary adjustment

We study the same approximation problem as in Problem 3.6. Since $\psi_i(0) = \psi_i(2\pi) = 0$ for all i , $u = 0$ at the boundary points $x = 0$ and $x = 2\pi$, while $f(0) = -1$ and $f(2\pi) = 1$. This discrepancy at the boundary can be removed by adding a boundary function $B(x)$:

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x),$$

where $B(x)$ has the right boundary values: $B(x_L) = f(x_L)$ and $B(x_R) = f(x_R)$, with $x_L = 0$ and $x_R = 2\pi$ as the boundary points. A linear choice of $B(x)$ is

$$B(x) = \frac{(x_R - x)f(x_L) + (x - x_L)f(x_R)}{x_R - x_L}.$$

- a) Use the basis $\psi_i(x) = \sin((i+1)x)$, $i \in \mathcal{I}_s = \{0, \dots, N\}$ and plot u and f for $N = 16$. (It suffices to make plots for even i .)

Solution. With a boundary term $B(x)$ we call `least_squares_orth` with $f-B$ as right-hand side function, and we must remember to add B to u .

We can extend the code from Exercise 3.6 and let the function `efficient` handle different choices of basis. Appropriate code for all three subexercises is

```

import sympy as sym
from approx1D import least_squares_orth, comparison_plot
import matplotlib.pyplot as plt
x = sym.Symbol('x')

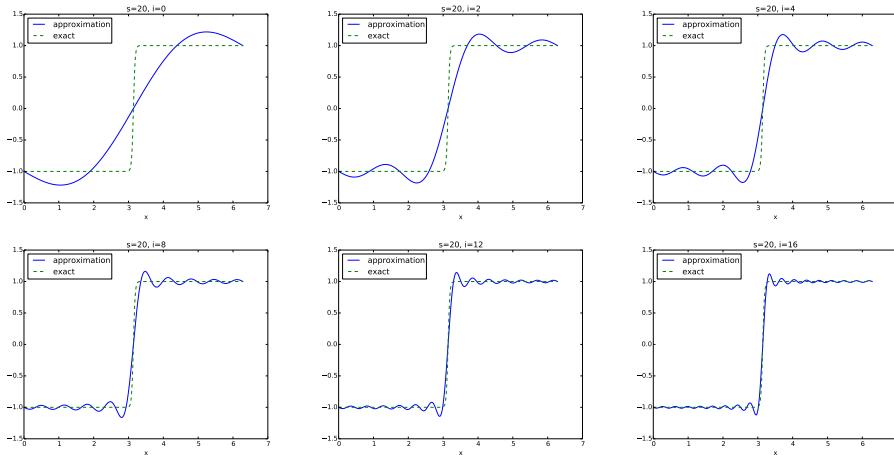
def efficient(f, B, s, Omega, N=10, basis='a'):
    u = B
    for i in range(N+1):
        if basis == 'a':
            psi = [sym.sin((i+1)*x)]
        elif basis == 'b':
            psi = [sym.sin((2*i+1)*x)]
        elif basis == 'c':
            psi = [sym.sin(2*(i+1)*x)]
    next_term, c = least_squares_orth(f-B, psi, Omega, False)
    u = u + next_term
    # Make only plot for i even
    if i % 2 == 0:
        comparison_plot(f, u, Omega, 'tmp_sin%02dx' % i,
                         legend_loc='upper left', show=False,
                         plot_title='s=%g, i=%d' % (s, i))

if __name__ == '__main__':
    s = 20 # steepness
    f = sym.tanh(s*(x-sym.pi))
    from math import pi
    Omega = [0, 2*pi] # sym.pi did not work here

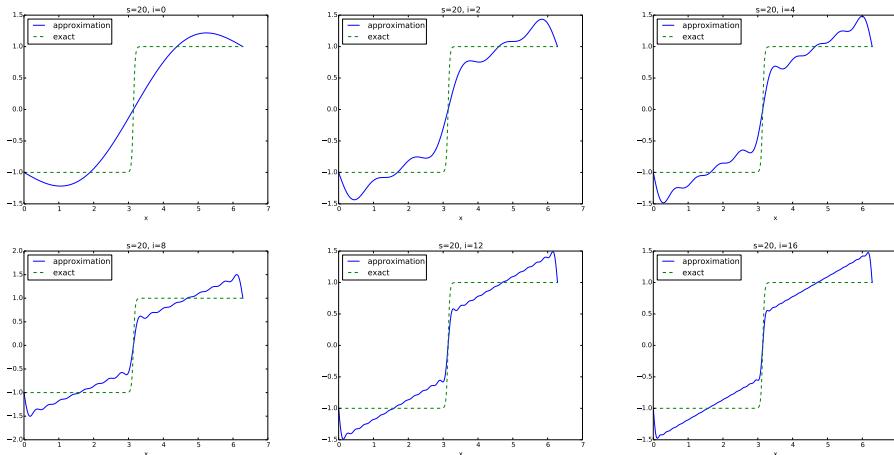
    # sin((i+1)*x) basis
    xL = Omega[0]
    xR = Omega[1]
    B = ((xR-x)*f.subs(x, xL) + (x-xL)*f.subs(x, xR))/(xR-xL)
    for exercise in 'a', 'b', 'c':
        efficient(f, B, s, Omega, N=16, basis=exercise)
        # Make movie
        cmd = 'convert -delay 200 tmp_sin*.png '
        cmd += 'tanh_sines_boundary_term_%s.gif' % exercise
        os.system(cmd)
    # Make static plots, 3 figures on 2 lines
    for ext in 'pdf', 'png':
        cmd = 'doconce combine_images %s -3' % ext
        cmd += 'tmp_sin0x tmp_sin02x tmp_sin04x tmp_sin08x '
        cmd += 'tmp_sin12x tmp_sin16x '
        cmd += 'tanh_sines_boundary_term_%s' % exercise

```

```
os.system(cmd)
```

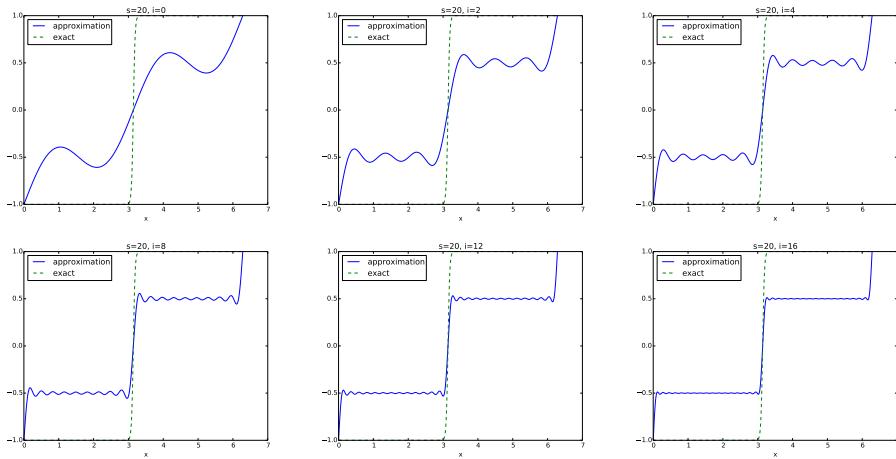


- b)** Use the basis from Exercise 3.6, $\psi_i(x) = \sin((2i+1)x)$, $i \in \mathcal{I}_s = \{0, \dots, N\}$. (It suffices to make plots for even i .) Observe that the approximation converges to a piecewise linear function!



Solution.

- c)** Use the basis $\psi_i(x) = \sin(2(i+1)x)$, $i \in \mathcal{I}_s = \{0, \dots, N\}$, and observe that the approximation converges to a piecewise constant function.



Solution. Filename: `tanh_sines_boundary_term`.

Remarks. The strange results in b) and c) are due to the choice of basis. In b), $\varphi_i(x)$ is an odd function around $x = \pi/2$ and $x = 3\pi/2$. No combination of basis functions is able to approximate the flat regions of f . All basis functions in c) are even around $x = \pi/2$ and $x = 3\pi/2$, but odd at $x = 0, \pi, 2\pi$. With all the sines represented, as in a), the approximation is not constrained with a particular symmetry behavior.

Exercise 3.8: Fourier series as a least squares approximation

a) Given a function $f(x)$ on an interval $[0, L]$, look up the formula for the coefficients a_j and b_j in the Fourier series of f :

$$f(x) = \frac{1}{2}a_0 + \sum_{j=1}^{\infty} a_j \cos\left(j\frac{2\pi x}{L}\right) + \sum_{j=1}^{\infty} b_j \sin\left(j\frac{2\pi x}{L}\right).$$

Solution. From Wikipedia we have

$$\begin{aligned} a_j &= \frac{2}{L} \int_0^L f(x) \cos\left(j\frac{2\pi x}{L}\right) dx, \\ b_j &= \frac{2}{L} \int_0^L f(x) \sin\left(j\frac{2\pi x}{L}\right) dx. \end{aligned}$$

b) Let an infinite-dimensional vector space V have the basis functions $\cos j\frac{2\pi x}{L}$ for $j = 0, 1, \dots, \infty$ and $\sin j\frac{2\pi x}{L}$ for $j = 1, \dots, \infty$. Show that

the least squares approximation method from Section 3.2 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 3.3.2).

Hint. You may choose

$$\psi_{2i} = \cos\left(i\frac{2\pi}{L}x\right), \quad \psi_{2i+1} = \sin\left(i\frac{2\pi}{L}x\right), \quad (3.67)$$

for $i = 0, 1, \dots, N \rightarrow \infty$.

Solution. The entries in the linear system arising from the least squares method are $A_{i,j} = \int_0^L \psi_i \psi_j dx$ and $b_i = \int_0^L f(x) \psi_i dx$. To avoid name clash between the right-hand side components of the linear system and the b_i coefficients in the Fourier series, we use the symbol q_i for the former. With the basis functions in (3.67) we get four different types of integrals:

$$\begin{aligned} A_{2i,2j} &= \int_0^L \cos\left(i\frac{2\pi}{L}x\right) \cos\left(j\frac{2\pi}{L}x\right) dx = A_{2j,2i}, \\ A_{2i,2j+1} &= \int_0^L \cos\left(i\frac{2\pi}{L}x\right) \sin\left(j\frac{2\pi}{L}x\right) dx, \\ A_{2i+1,2j} &= \int_0^L \sin\left(i\frac{2\pi}{L}x\right) \cos\left(j\frac{2\pi}{L}x\right) dx, \\ A_{2i+1,2j+1} &= \int_0^L \sin\left(i\frac{2\pi}{L}x\right) \sin\left(j\frac{2\pi}{L}x\right) dx, \\ q_{2i} &= \int_0^L f(x) \cos\left(i\frac{2\pi}{L}x\right) dx, \\ q_{2i+1} &= \int_0^L f(x) \sin\left(i\frac{2\pi}{L}x\right) dx. \end{aligned}$$

Now, the sine and cosine basis functions are orthogonal on $[0, L]$. We have in general

$$\begin{aligned}
\int_0^L \cos\left(i \frac{2\pi}{L} x\right) \cos\left(j \frac{2\pi}{L} x\right) dx &= 0, \quad i \neq j, \\
\int_0^L \cos\left(i \frac{2\pi}{L} x\right) \cos\left(j \frac{2\pi}{L} x\right) dx &= \frac{L}{2}, \quad i = j \neq 0, \\
\int_0^L \cos\left(i \frac{2\pi}{L} x\right) \cos\left(j \frac{2\pi}{L} x\right) dx &= L, \quad i = j = 0, \\
\int_0^L \sin\left(i \frac{2\pi}{L} x\right) \sin\left(j \frac{2\pi}{L} x\right) dx &= 0, \quad i \neq j, \\
\int_0^L \sin\left(i \frac{2\pi}{L} x\right) \sin\left(j \frac{2\pi}{L} x\right) dx &= \frac{L}{2}, \quad i = j, \\
\int_0^L \cos\left(i \frac{2\pi}{L} x\right) \sin\left(j \frac{2\pi}{L} x\right) dx &= 0.
\end{aligned}$$

These results imply that only diagonal terms in the coefficient matrix are different from zero. We have

$$\begin{aligned}
A_{0,0} &= L, \\
A_{2i,2i} &= \frac{L}{2}, \quad i > 0, \\
A_{2i+1,2i+1} &= \frac{L}{2}.
\end{aligned}$$

The unknown vector with components c_i must be arranged as

$$(a_0, b_1, a_1, b_2, a_2, b_3, \dots).$$

We then get

$$A_{0,0}a_0 = q_0, \quad A_{1,1}b_1 = q_1, \quad A_{2,2}a_1 = q_2, \quad A_{3,3}b_2 = q_3, \dots$$

These equations lead to the formulas

$$\begin{aligned}a_0 &= \frac{1}{L} \int_0^P f(x) dx, \\b_1 &= \frac{2}{L} \int_0^P f(x) \sin\left(\frac{2\pi}{L}x\right) dx, \\a_1 &= \frac{2}{L} \int_0^P f(x) \cos\left(\frac{2\pi}{L}x\right) dx, \\b_2 &= \frac{2}{L} \int_0^P f(x) \sin\left(2\frac{2\pi}{L}x\right) dx, \\a_2 &= \frac{2}{L} \int_0^P f(x) \cos\left(2\frac{2\pi}{L}x\right) dx.\end{aligned}$$

which can be generalized to

$$\begin{aligned}a_0 &= \frac{1}{L} \int_0^P f(x) dx, \\a_j &= \frac{2}{L} \int_0^P f(x) \cos\left(j\frac{2\pi}{L}x\right) dx, \quad j > 0, \\b_j &= \frac{2}{L} \int_0^P f(x) \sin\left(j\frac{2\pi}{L}x\right) dx, \quad j > 0,\end{aligned}$$

and these are the standard formulas for the Fourier coefficients in a) if we recognize that the a_0 above is twice the a_0 in the expressions in a).

- c)** Choose $f(x) = H(x - \frac{1}{2})$ on $\Omega = [0, 1]$, where H is the Heaviside function: $H(x) = 0$ for $x < 0$, $H(x) = 1$ for $x > 0$ and $H(0) = \frac{1}{2}$. Find the coefficients a_j and b_j in the Fourier series for $f(x)$. Plot the sum for $j = 2N + 1$, where $N = 5$ and $N = 100$.

Solution. The formulas give

$$\begin{aligned}a_0 &= 2 \int_0^1 f(x) dx = 2 \int_{\frac{1}{2}}^1 dx, \\a_j &= 2 \int_0^1 f(x) \cos(2j\pi x) dx = 2 \int_{\frac{1}{2}}^1 \cos(2j\pi x) dx, \\b_j &= 2 \int_{\frac{1}{2}}^1 \sin(2j\pi x) dx.\end{aligned}$$

The integrals are readily computed by `sympy`:

```
>>> import sympy as sym
>>> j = sym.symbols('k', integer=True)
```

```
>>> x = sym.symbols('x', real=True)
>>> I = integrate(cos(2*j*pi*x), (x,Rational(1,2),1))
>>> I
0
>>> I = integrate(cos(2*0*pi*x), (x,Rational(1,2),1))
>>> I
1/2
>>> I = integrate(sin(2*j*pi*x), (x,Rational(1,2),1))
>>> I
(-1)**j/(2*pi*j) - 1/(2*pi*j)
```

This means that we have the series

$$u(x) = \frac{1}{2} + 2 \sum_{j=1}^{\infty} \frac{(-1)^j - 1}{2\pi j} \sin(2j\pi x).$$

We only get a nonzero coefficient for j odd:

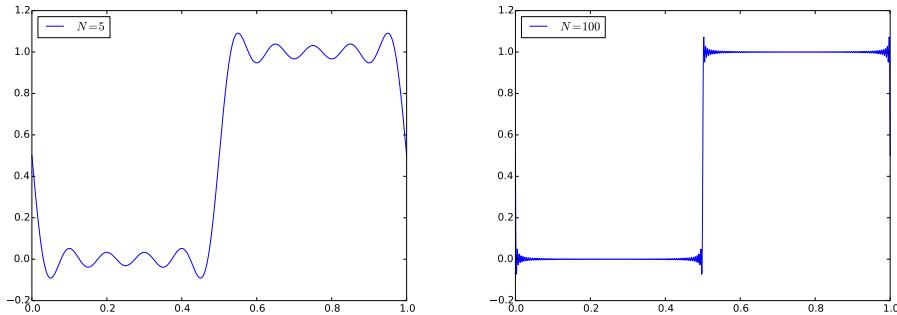
$$u(x) = \frac{1}{2} - 2 \sum_{k=1}^{\infty} \frac{1}{(2k+1)\pi} \sin(2(2k+1)\pi x).$$

Appropriate computer code for visualizing the series goes like

```
import numpy as np
import matplotlib.pyplot as plt
from math import pi
from numpy import sin

def Heaviside_series(x, N):
    s = 0.5
    for k in range(N):
        s += -2.0/((2*k+1)*pi)*sin(2*(2*k+1)*pi*x)
    return s

x = np.linspace(0, 1, 1001)
for N in 5, 100:
    H = Heaviside_series(x, N)
    plt.figure()
    plt.plot(x, H)
    plt.legend(['$N=%d$' % N], loc='upper left')
    plt.savefig('tmp_%d.png' % N)
    plt.savefig('tmp_%d.pdf' % N)
plt.show()
```



We clearly see the Gibbs' phenomenon: oscillations and overshoot around the point of discontinuity in the function we try to approximate. Filename: `Fourier_ls`.

Problem 3.9: Approximate a steep function by Lagrange polynomials

Use interpolation with uniformly distributed points and Chebychev nodes to approximate

$$f(x) = -\tanh\left(s\left(x - \frac{1}{2}\right)\right), \quad x \in [0, 1],$$

by Lagrange polynomials for $s = 5$ and $s = 20$, and $N = 3, 7, 11, 15$. Combine 2×2 plots of the approximation for the four N values, and create such figures for the four combinations of s values and point types.

Solution. The following code does the work (symbolically):

```
import sys, os
sys.path.insert(0, os.path.join(os.pardir, 'src'))
from approx1D import interpolation, comparison_plot
from Lagrange import Lagrange_polynomials
import sympy as sym

x = sym.Symbol('x')
Omega = [0, 1]
N_values = 3, 7, 11, 15

for s in 5, 20:
    f = -sym.tanh(s*(x-0.5)) # sympy expression
    for distribution in 'uniform', 'Chebyshev':
        for N in N_values:
            phi, points = Lagrange_polynomials(
                x, N, Omega,
```

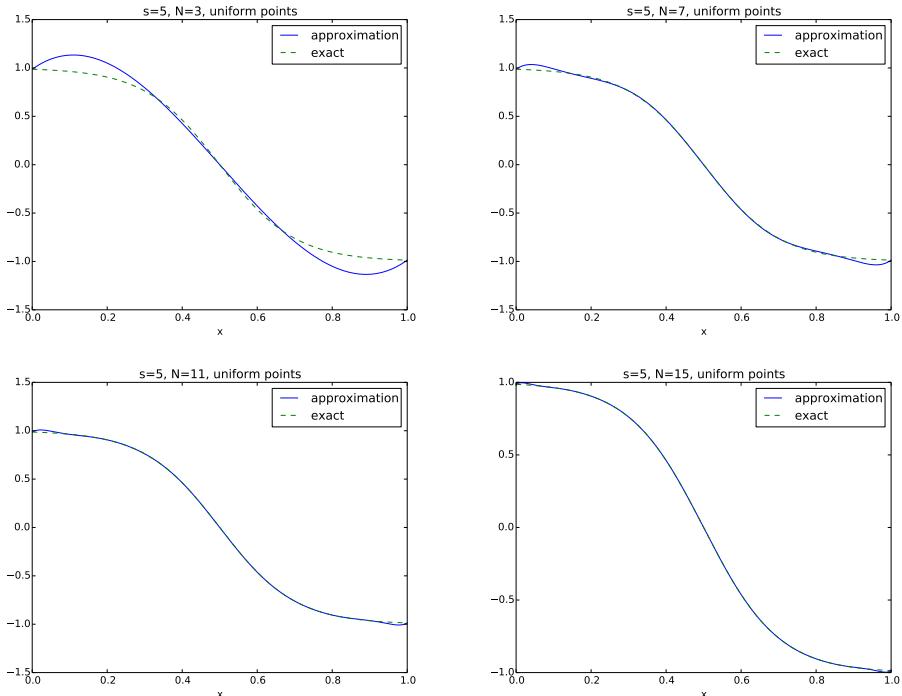
```

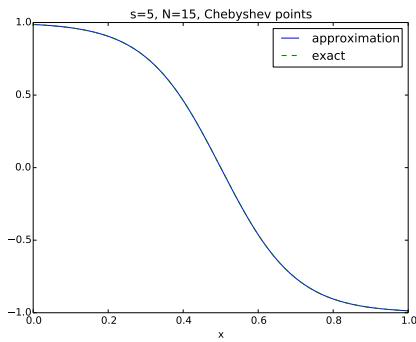
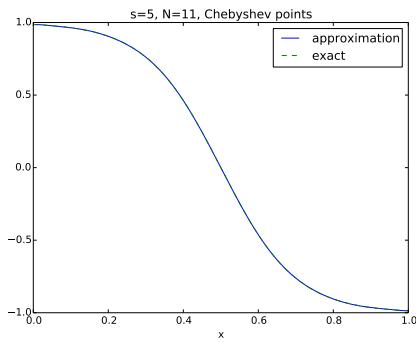
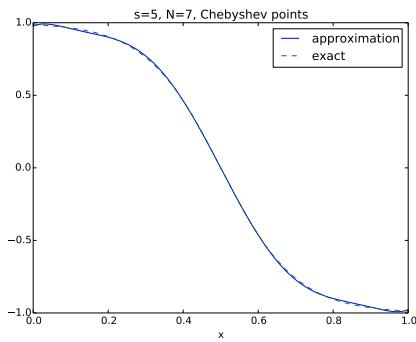
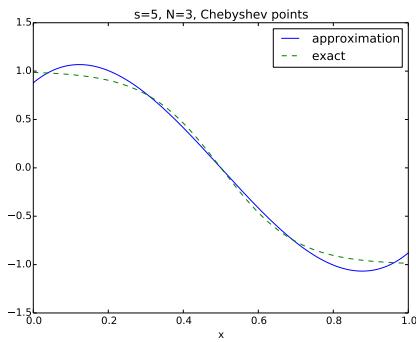
    point_distribution=distribution)

    u, c = interpolation(f, phi, points)
    filename = 'tmp_tanh_%d_%d_%s' % (N, s, distribution)
    comparison_plot(f, u, Omega, filename,
                     plot_title='s=%g, N=%d, %s points' %
                     (s, N, distribution))
# Combine plot files (2x2)
for ext in 'png', 'pdf':
    cmd = 'doconce combine_images ' + ext + ' '
    cmd += ' '.join([
        'tmp_tanh_%d_%d_%s' % (N, s, distribution)
        for N in N_values])
    cmd += ' tanh_Lagrange_%s_s%s' % (distribution, s)
    os.system(cmd)

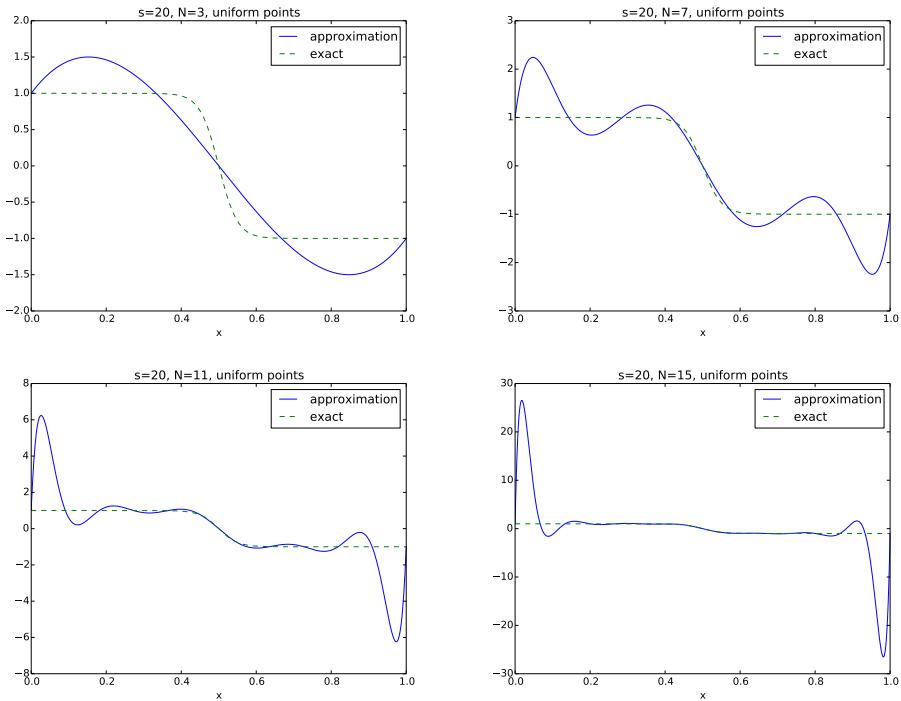
```

For a smooth function ($s = 5$), the difference between uniform points and Chebyshev nodes is not substantial:

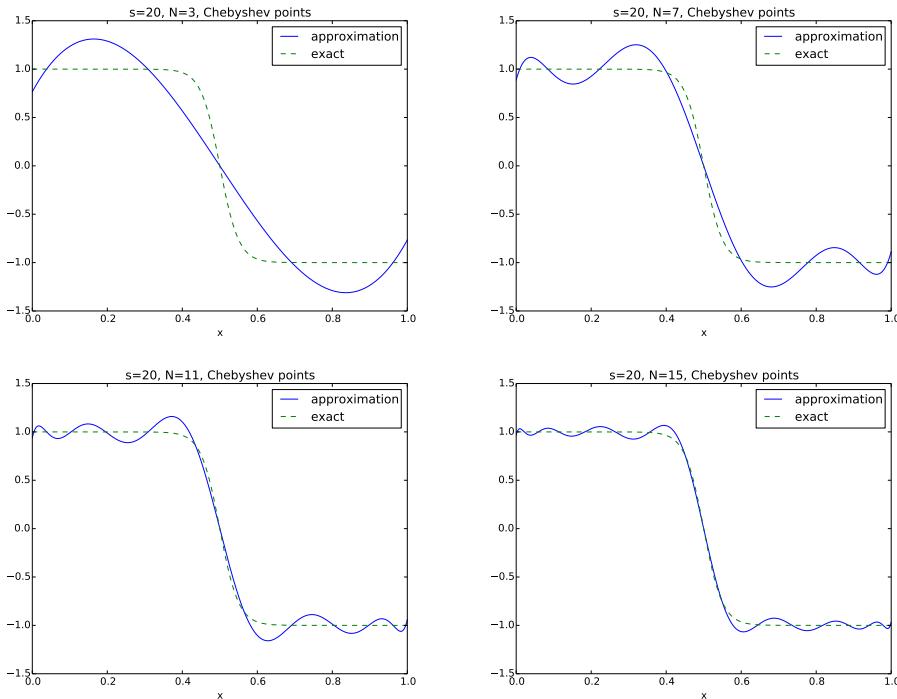




However, for a steep function ($s = 20$) the overshoot and oscillations associated with uniform points must be considered unacceptable for larger N values:



Switching to Chebyshev points does give a great improvement, but we still have oscillatory approximations:



Filename: `tanh_Lagrange`.

Problem 3.10: Approximate a steep function by Lagrange polynomials and regression

Redo Problem 3.9, but apply a regression method with N -degree Lagrange polynomials and $2N + 1$ data points. Recall that Problem 3.9 applies $N + 1$ points and the resulting approximation interpolates f at these points, while a regression method with more points does not interpolate f at the data points. Do more points and a regression method help reduce the oscillatory behavior of Lagrange polynomial approximations?

Solution. We start out with the program from Problem 3.9. This time we need to call `Lagrange_polyomials` twice: first to compute the $\psi(x)$ functions (of degree N) and then to compute the data points corresponding to a uniform or Chebyshev distribution of $2N + 1$ nodes.

```
import sys, os
sys.path.insert(0, os.path.join(os.pardir, 'src'))
from approx1D import regression, comparison_plot
```

```

from Lagrange import Lagrange_polynomials
import sympy as sym
import numpy as np

x = sym.Symbol('x')
Omega = [0,1]
N_values = 3, 7, 11, 15

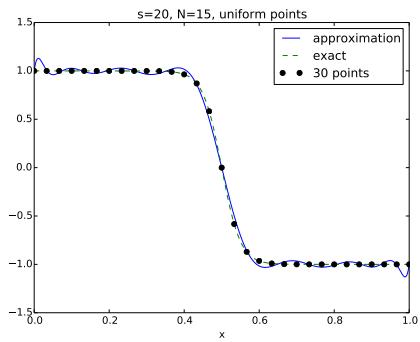
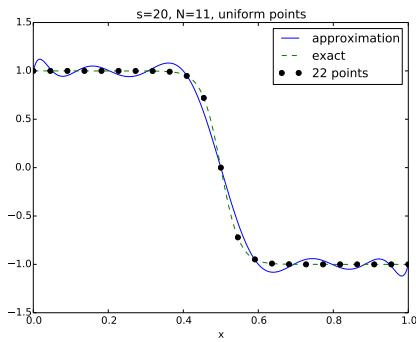
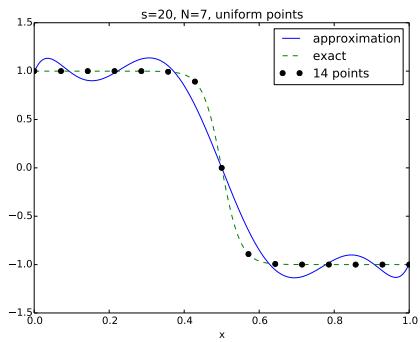
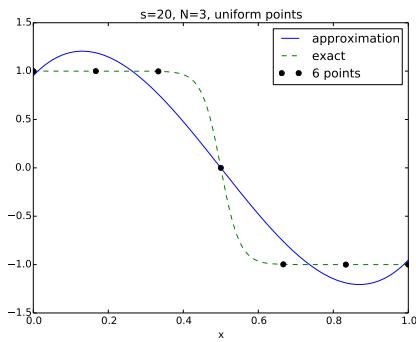
for s in 5, 20:
    f = -sym.tanh(s*(x-0.5)) # sympy expression
    for distribution in 'uniform', 'Chebyshev':
        for N in N_values:
            # Compute the points from a 2*N Lagrange polynomial
            dummy, points = Lagrange_polynomials(
                x, 2*N, Omega,
                point_distribution=distribution)
            # Compute phi from N points Lagrange polynomial
            phi, dummy = Lagrange_polynomials(
                x, N, Omega,
                point_distribution=distribution)
            points = np.array(points, dtype=float)
            point_values = -np.tanh(s*(points-0.5))

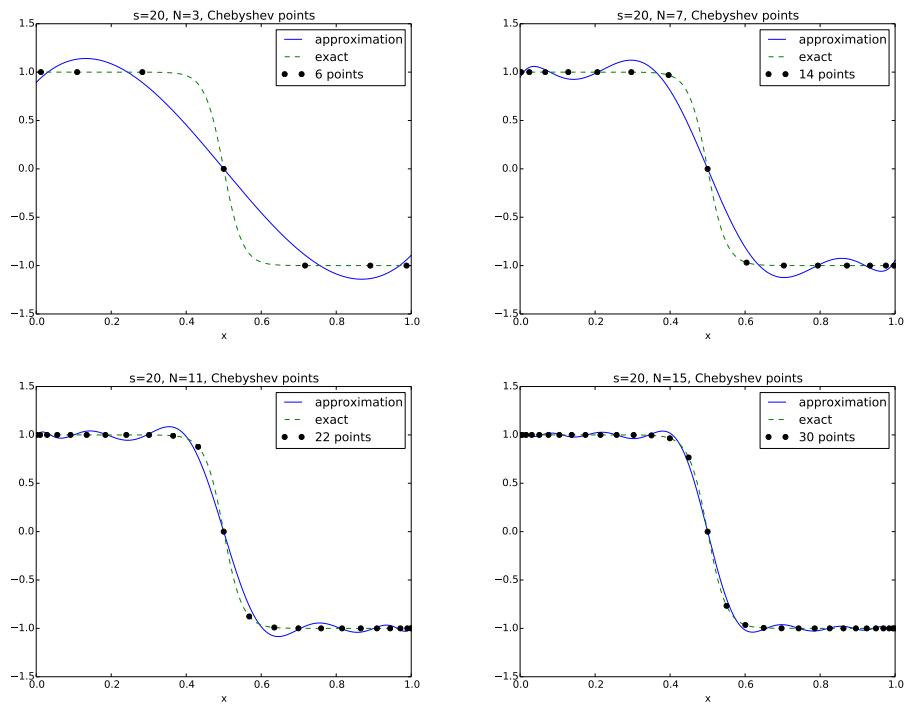
            u, c = regression(f, phi, points)
            filename = 'tmp_tanh_%d_%d_%s' % (N, s, distribution)
            comparison_plot(f, u, Omega, filename,
                            plot_title='s=%g, N=%d, %s points' %
                            (s, N, distribution),
                            points=points, point_values=point_values,
                            points_legend='%s points' % (2*N))
        # Combine plot files (2x2)
        for ext in 'png', 'pdf':
            cmd = 'doconce combine_images ' + ext + ' '
            cmd += ' '.join([
                'tmp_tanh_%d_%d_%s' % (N, s, distribution)
                for N in N_values])
            cmd += ' tanh_Lagrange_regr_%s_%s' % (distribution, s)
            os.system(cmd)

```

An important point is to convert `points` to a `numpy` array using `dtype=float`. Leaving out this second argument makes an array of objects of symbolic expressions, and we cannot apply `tanh` to it.

The oscillatory behavior is much reduced using more points and a regression method, and the difference between uniform and Chebyshev points is minor, even in the steep case $s = 20$:





Filename: `tanh_Lagrange_regression.`

The purpose of this chapter is to use the ideas from the previous chapter on how to approximate functions, but the basis functions are now of finite element type.

4.1 Finite element basis functions

The specific basis functions exemplified in Section 3.2 are in general nonzero on the entire domain Ω , as can be seen in Figure 4.1, where we plot two sinusoidal basis functions $\psi_0(x) = \sin \frac{1}{2}\pi x$ and $\psi_1(x) = \sin 2\pi x$ together with the sum $u(x) = 4\psi_0(x) - \frac{1}{2}\psi_1(x)$. We shall now turn our attention to basis functions that have *compact support*, meaning that they are nonzero on a small portion of Ω only. Moreover, we shall restrict the functions to be *piecewise polynomials*. This means that the domain is split into subdomains and each basis function is a polynomial on one or more of these subdomains, see Figure 4.2 for a sketch involving locally defined hat functions that make $u = \sum_j c_j \psi_j$ piecewise linear. At the boundaries between subdomains, one normally just forces continuity of u , so that when connecting two polynomials from two subdomains, the derivative becomes discontinuous. This type of basis functions is fundamental in the finite element method. (One may wonder why continuity of derivatives is not desired, and it is, but it turns out to be mathematically challenging in 2D and 3D, and it is not strictly needed.)

We first introduce the concepts of elements and nodes in a simplistic fashion. Later, we shall generalize the concept of an element, which is a

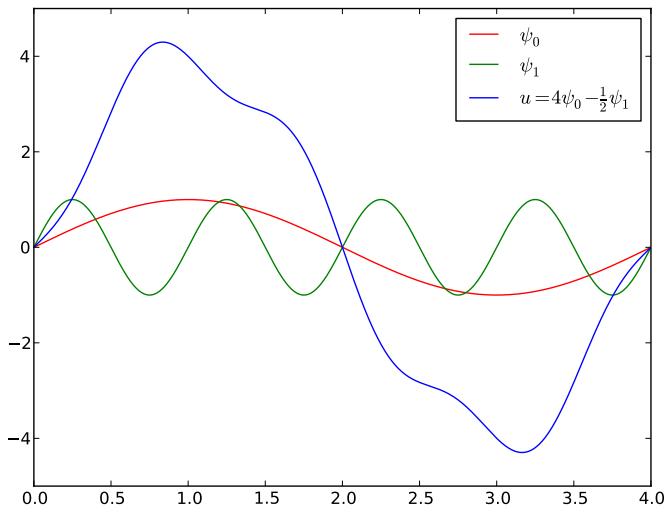


Fig. 4.1 A function resulting from a weighted sum of two sine basis functions.

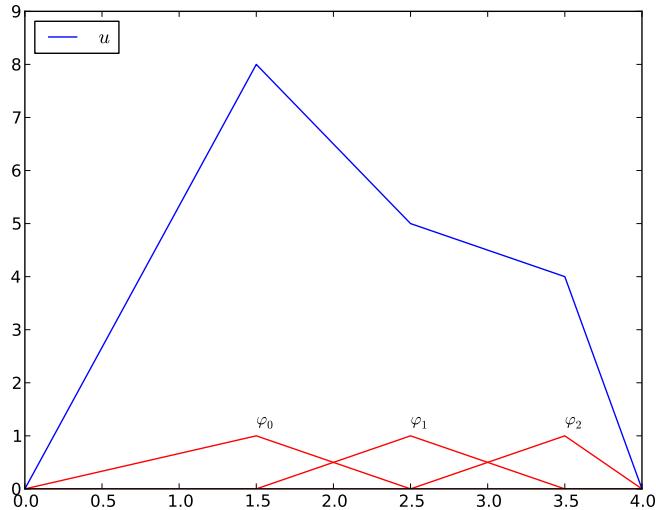


Fig. 4.2 A function resulting from a weighted sum of three local piecewise linear (hat) functions.

necessary step before treating a wider class of approximations within the

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

4.1.1 Elements and nodes

Let u and f be defined on an interval Ω . We divide Ω into N_e non-overlapping subintervals $\Omega^{(e)}$, $e = 0, \dots, N_e - 1$:

$$\Omega = \Omega^{(0)} \cup \dots \cup \Omega^{(N_e)}. \quad (4.1)$$

We shall for now refer to $\Omega^{(e)}$ as an *element*, identified by the unique 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. Local nodes are numbered with an index $r = 0, \dots, d$, while the N_n global nodes are numbered as $i = 0, \dots, N_n - 1$. Figure 4.3 shows nodes as small circular disks and element boundaries as small vertical lines. Global node numbers appear under the nodes, but local node numbers are not shown. Since there are two nodes in each element, the local nodes are numbered 0 (left) and 1 (right) in each element.

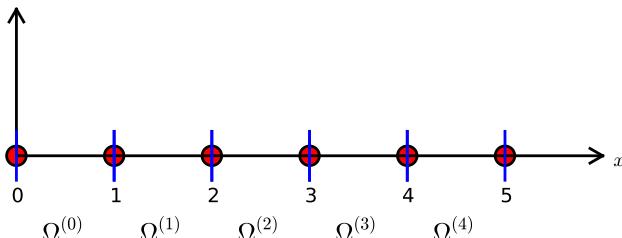


Fig. 4.3 Finite element mesh with 5 elements and 6 nodes.

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. Figure 4.3 shows an example on a uniformly partitioned mesh. The strength of the finite element method (in contrast to the finite difference method) is that it is just as easy to work with a non-uniformly partitioned mesh in 3D as a uniformly partitioned mesh in 1D.

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. Figure 4.4 shows the mesh with $N_e = 2$ elements and $N_n = 2N_e + 1 = 5$ nodes. A node's coordinate is denoted by x_i , where i is either a global node number or a local one. In the latter case we also need to know the element number to uniquely define the node.

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.

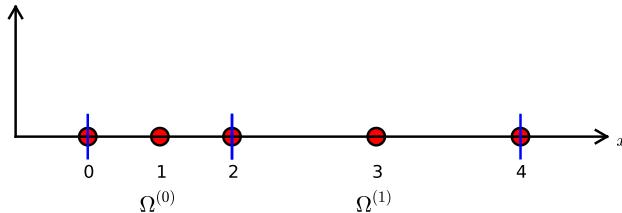


Fig. 4.4 Finite element mesh with 2 elements and 5 nodes.

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. By defining `elements` as a list of lists, where each sublist contains the global node numbers of one particular element, the indices of each sublist will correspond to local node numbers for that element. 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, e.g., local node number 2 in element 1, is done by `nodes[elements[1][2]]` (recall that nodes and elements start their numbering at 0). The corresponding global node number is 4, so we could alternatively look up the coordinate as `nodes[4]`.

The numbering of elements and nodes does not need to be regular. Figure 4.5 shows an example corresponding to

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

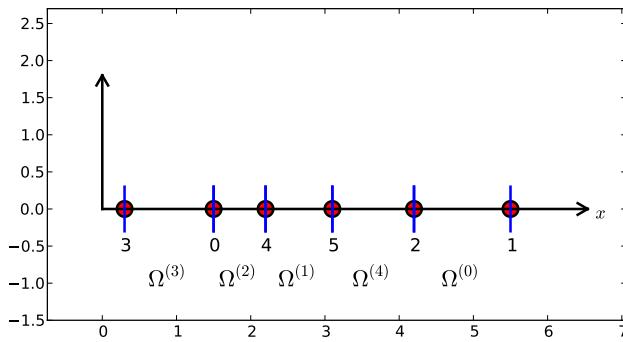


Fig. 4.5 Example on irregular numbering of elements and nodes.

4.1.2 The basis functions

Construction principles. Finite element basis functions are in this text recognized by the notation $\varphi_i(x)$, where the index (now in the beginning) corresponds to a global node number. Since ψ_i is the symbol for basis functions in general in this text, the particular choice of finite element basis functions means that we take $\psi_i = \varphi_i$.

Let i be the global node number corresponding to local node r in element number e with $d+1$ local nodes. We distinguish between *internal* nodes in an element and *shared* nodes. The latter are nodes that are shared with the neighboring elements. The finite element basis functions φ_i are now defined as follows.

- For an internal node, with global number i and local number r , take $\varphi_i(x)$ to be the Lagrange polynomial that is 1 at the local node r and zero at all other nodes in the element. The degree of the polynomial is d , according to (3.56). On all other elements, $\varphi_i = 0$.
- For a shared node, let φ_i be made up of the Lagrange polynomial on this element that is 1 at node i , combined with the Lagrange polynomial over the neighboring element that is also 1 at node i . On all other elements, $\varphi_i = 0$.

A visual impression of three such basis functions is given in Figure 4.6. When solving differential equations, we need the derivatives of these basis functions as well, and the corresponding derivatives are shown in Figure 4.7. Note that the derivatives are highly discontinuous! In these figures, the domain $\Omega = [0, 1]$ is divided into four equal-sized elements, each having three local nodes. The element boundaries are marked by vertical dashed lines and the nodes by small circles. The function $\varphi_2(x)$

is composed of a quadratic Lagrange polynomial over element 0 and 1, $\varphi_3(x)$ corresponds to an internal node in element 1 and is therefore nonzero on this element only, while $\varphi_4(x)$ is like $\varphi_2(x)$ composed to two Lagrange polynomials over two elements. Also observe that the basis function φ_i is zero at all nodes, except at global node number i . We also remark that the shape of a basis function over an element is completely determined by the coordinates of the local nodes in the element.

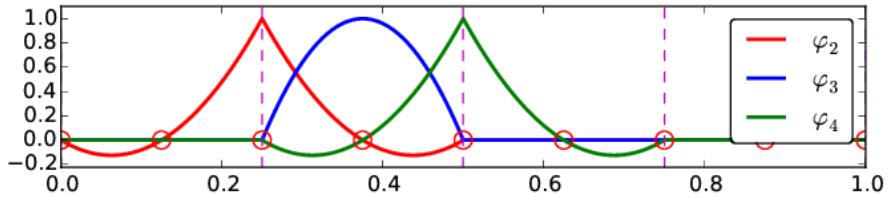


Fig. 4.6 Illustration of the piecewise quadratic basis functions associated with nodes in an element.

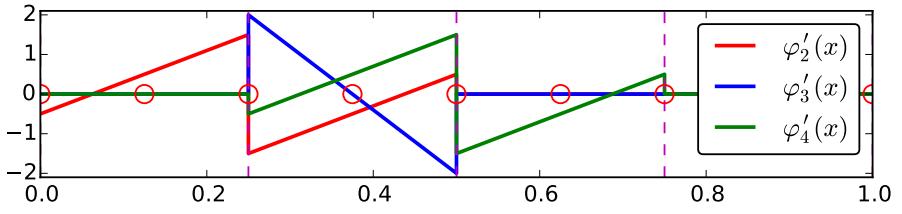


Fig. 4.7 Illustration of the derivatives of the piecewise quadratic basis functions associated with nodes in an element.

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

$$\varphi_i(x_j) = \delta_{ij}, \quad \delta_{ij} = \begin{cases} 1, & i = j, \\ 0, & i \neq j, \end{cases} \quad (4.2)$$

when x_j is a node in the mesh with global node number j . The result $\varphi_i(x_j) = \delta_{ij}$ is obtained as the Lagrange polynomials are constructed to have exactly this property. The property also implies a convenient interpretation of c_i as the value of u at node i . To show this, we expand u in the usual way as $\sum_j c_j \psi_j$ and choose $\psi_i = \varphi_i$:

$$u(x_i) = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x_i) = \sum_{j \in \mathcal{I}_s} c_j \varphi_j(x_i) = c_i \varphi_i(x_i) = c_i.$$

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.

In our example so far, each element has $d + 1$ nodes, resulting in local Lagrange polynomials of degree d (according to Section 3.4.2), but it is not a requirement to have the same d value in each element.

4.1.3 Example on quadratic finite element functions

Let us set up the `nodes` and `elements` lists corresponding to the mesh implied by Figure 4.6. Figure 4.8 sketches the mesh and the numbering. We have

```
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]]
```

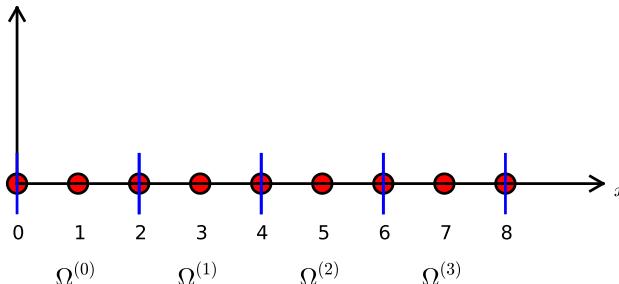


Fig. 4.8 Sketch of mesh with 4 elements and 3 nodes per element.

Let us explain mathematically how the basis functions are constructed according to the principles. Consider element number 1 in Figure 4.8, $\Omega^{(1)} = [0.25, 0.5]$, 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 (global node 3) makes up the basis function $\varphi_3(x)$ over this element, with $\varphi_3(x) = 0$ outside the element.
2. The polynomial that is 1 at local node 0 (global node 2) is the “right part” of the global basis function $\varphi_2(x)$. The “left part” of $\varphi_2(x)$ consists of a Lagrange polynomial associated with local node 2 in the neighboring element $\Omega^{(0)} = [0, 0.25]$.
3. Finally, the polynomial that is 1 at local node 2 (global node 4) is the “left part” of the global basis function $\varphi_4(x)$. The “right part” comes from the Lagrange polynomial that is 1 at local node 0 in the neighboring element $\Omega^{(2)} = [0.5, 0.75]$.

The specific mathematical form of the polynomials *over element 1* is given by the formula (3.56):

$$\begin{aligned}\varphi_3(x) &= \frac{(x - 0.25)(x - 0.5)}{(0.375 - 0.25)(0.375 - 0.5)}, \quad x \in \Omega^{(1)} \\ \varphi_2(x) &= \frac{(x - 0.375)(x - 0.5)}{(0.25 - 0.375)(0.25 - 0.5)}, \quad x \in \Omega^{(1)} \\ \varphi_4(x) &= \frac{(x - 0.25)(x - 0.375)}{(0.5 - 0.25)(0.5 - 0.25)}, \quad x \in \Omega^{(1)}.\end{aligned}$$

As mentioned earlier, any global basis function $\varphi_i(x)$ is zero on elements that do not contain the node with global node number i . Clearly, the property (4.2) is easily verified, see for instance that $\varphi_3(0.375) = 1$ while $\varphi_3(0.25) = 0$ and $\varphi_3(0.5) = 0$.

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 in Figure 4.6, while the global basis functions associated with shared nodes have the same shape as shown φ_2 and φ_4 . If the elements were of different length, the basis functions would be stretched according to the element size and hence be different.

4.1.4 Example on linear finite element functions

Figure 4.9 shows piecewise linear basis functions ($d = 1$) (with derivatives in Figure 4.10). These are mathematically simpler than the quadratic

functions in the previous section, and one would therefore think that it is easier to understand the linear functions first. However, linear basis functions do not involve internal nodes and are therefore a special case of the general situation. That is why we think it is better to understand the construction of quadratic functions first, which easily generalize to any $d > 2$, and then look at the special case $d = 1$.

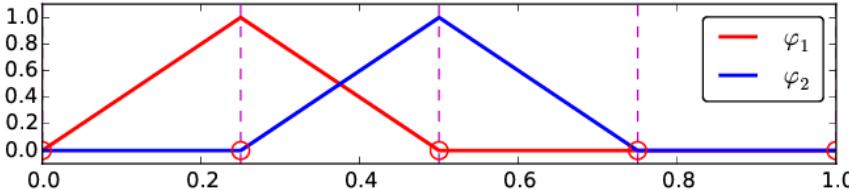


Fig. 4.9 Illustration of the piecewise linear basis functions associated with nodes in an element.

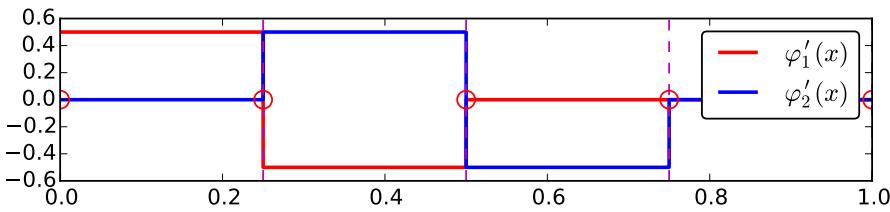


Fig. 4.10 Illustration of the derivatives of piecewise linear basis functions associated with nodes in an element.

We have the same four elements on $\Omega = [0, 1]$. Now there are no internal nodes in the elements so that all basis functions are associated with shared nodes and hence made up of two Lagrange polynomials, one from each of the two neighboring elements. For example, $\varphi_1(x)$ results from the Lagrange polynomial in element 0 that is 1 at local node 1 and 0 at local node 0, combined with the Lagrange polynomial in element 1 that is 1 at local node 0 and 0 at local node 1. The other basis functions are constructed similarly.

Explicit mathematical formulas are needed for $\varphi_i(x)$ in computations. In the piecewise linear case, the formula (3.56) leads to

$$\varphi_i(x) = \begin{cases} 0, & x < x_{i-1}, \\ (x - x_{i-1})/(x_i - x_{i-1}), & x_{i-1} \leq x < x_i, \\ 1 - (x - x_i)/(x_{i+1} - x_i), & x_i \leq x < x_{i+1}, \\ 0, & x \geq x_{i+1}. \end{cases} \quad (4.3)$$

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

$$\varphi_i(x) = \begin{cases} 0, & x < x_{i-1}, \\ (x - x_{i-1})/h, & x_{i-1} \leq x < x_i, \\ 1 - (x - x_i)/h, & x_i \leq x < x_{i+1}, \\ 0, & x \geq x_{i+1}. \end{cases} \quad (4.4)$$

4.1.5 Example on cubic finite element functions

Piecewise cubic basis functions can be defined by introducing four nodes per element. Figure 4.11 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 only on element number 1, while φ_3 and φ_6 are made up of Lagrange polynomials on two neighboring elements.

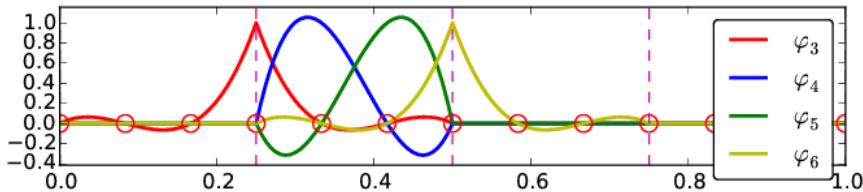


Fig. 4.11 Illustration of the piecewise cubic basis functions associated with nodes in an element.

We see that all the piecewise linear basis functions have the same “hat” shape. They are naturally referred to as *hat functions*, also called *chapeau functions*. The piecewise quadratic functions in Figure 4.6 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 4.11.

A common terminology is to speak about *linear elements* as elements with two local nodes associated with piecewise linear basis functions. Sim-

ilarly, *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 in the following, are P1 for linear elements, P2 for quadratic elements, and so forth: P d signifies degree d of the polynomial basis functions.

4.1.6 Calculating the linear system

The elements in the coefficient matrix and right-hand side are given by the formulas (3.28) and (3.29), but now the choice of ψ_i is φ_i . Consider P1 elements where $\varphi_i(x)$ is piecewise linear. Nodes and elements numbered consecutively from left to right in a uniformly partitioned mesh imply the nodes

$$x_i = ih, \quad i = 0, \dots, N_n - 1,$$

and the elements

$$\Omega^{(i)} = [x_i, x_{i+1}] = [ih, (i+1)h], \quad i = 0, \dots, N_e - 1. \quad (4.5)$$

We have in this case N_e elements and $N_n = N_e + 1$ nodes. The parameter N denotes the number of unknowns in the expansion for u , and with the P1 elements, $N = N_n$. The domain is $\Omega = [x_0, x_N]$. The formula for $\varphi_i(x)$ is given by (4.4) and a graphical illustration is provided in Figures 4.9 and 4.13.

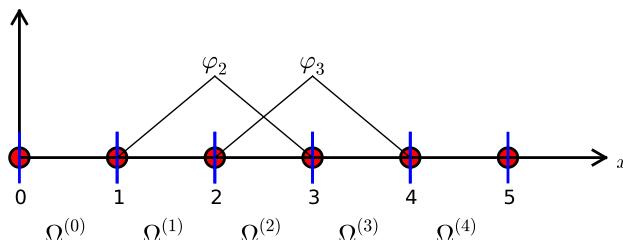


Fig. 4.12 Illustration of the piecewise linear basis functions corresponding to global node 2 and 3.

Calculating specific matrix entries. Let us calculate the specific matrix entry $A_{2,3} = \int_{\Omega} \varphi_2 \varphi_3 dx$. Figure 4.12 shows what φ_2 and φ_3 look like. We realize from this figure that the product $\varphi_2 \varphi_3 \neq 0$ only over element 2, which contains node 2 and 3. The particular formulas for $\varphi_2(x)$ and

$\varphi_3(x)$ on $[x_2, x_3]$ are found from (4.4). The function φ_3 has positive slope over $[x_2, x_3]$ and corresponds to the interval $[x_{i-1}, x_i]$ in (4.4). With $i = 3$ we get

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

while $\varphi_2(x)$ has negative slope over $[x_2, x_3]$ and corresponds to setting $i = 2$ in (4.4),

$$\varphi_2(x) = 1 - (x - x_2)/h.$$

We can now easily integrate,

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

The diagonal entry in the coefficient matrix becomes

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

The entry $A_{2,1}$ has an integral that is geometrically similar to the situation in Figure 4.12, so we get $A_{2,1} = h/6$.

Calculating a general row in the matrix. We can now generalize the calculation of matrix entries to a general row number i . The entry $A_{i,i-1} = \int_{\Omega} \varphi_i \varphi_{i-1} \, dx$ involves hat functions as depicted in Figure 4.13. Since the integral is geometrically identical to the situation with specific nodes 2 and 3, we realize that $A_{i,i-1} = A_{i,i+1} = h/6$ and $A_{i,i} = 2h/3$. However, we can compute the integral directly too:

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

The particular formulas for $\varphi_{i-1}(x)$ and $\varphi_i(x)$ on $[x_{i-1}, x_i]$ are found from (4.4): φ_i is the linear function with positive slope, corresponding

to the interval $[x_{i-1}, x_i]$ in (4.4), while ϕ_{i-1} has a negative slope so the definition in interval $[x_i, x_{i+1}]$ in (4.4) must be used.

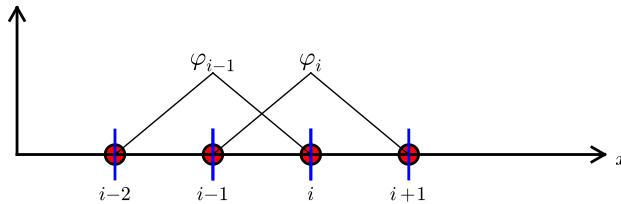


Fig. 4.13 Illustration of two neighboring linear (hat) functions with general node numbers.

The first and last row of the coefficient matrix lead to slightly different integrals:

$$A_{0,0} = \int_{\Omega} \varphi_0^2 dx = \int_{x_0}^{x_1} \left(1 - \frac{x - x_0}{h}\right)^2 dx = \frac{h}{3}.$$

Similarly, $A_{N,N}$ involves an integral over only one element and hence equals $h/3$.

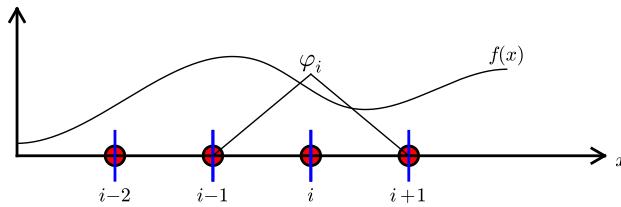


Fig. 4.14 Right-hand side integral with the product of a basis function and the given function to approximate.

The general formula for b_i , see Figure 4.14, is now easy to set up

$$b_i = \int_{\Omega} \varphi_i(x) f(x) dx = \int_{x_{i-1}}^{x_i} \frac{x - x_{i-1}}{h} f(x) dx + \int_{x_i}^{x_{i+1}} \left(1 - \frac{x - x_i}{h}\right) f(x) dx. \quad (4.6)$$

We remark that the above formula applies to internal nodes (living at the interface between two elements) and that for the nodes on the boundaries only one integral needs to be computed.

We need a specific $f(x)$ function to compute these integrals. With $f(x) = x(1 - x)$ and two equal-sized elements in $\Omega = [0, 1]$, one gets

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

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 4.15 (left). Doubling the number of elements to four leads to the improved approximation in the right part of Figure 4.15.

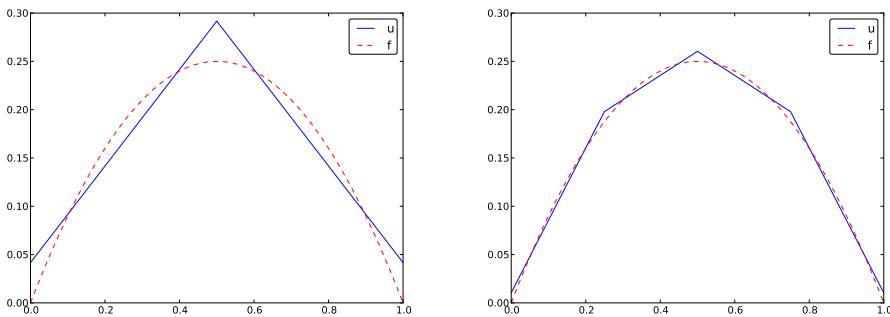


Fig. 4.15 Least squares approximation of a parabola using 2 (left) and 4 (right) P1 elements.

4.1.7 Assembly of elementwise computations

Our integral computations so far have been straightforward. However, with higher-degree polynomials and in higher dimensions (2D and 3D), integrating in the physical domain gets increasingly complicated. Instead, integrating over one element at a time, and transforming each element to a common standardized geometry in a new reference coordinate system, is technically easier. Almost all computer codes employ a finite element algorithm that calculates the linear system by integrating over one element at a time. We shall therefore explain this algorithm next. The amount of details might be overwhelming during a first reading, but once all those details are done right, one has a general finite element algorithm

that can be applied to all sorts of elements, in any space dimension, no matter how geometrically complicated the domain is.

The element matrix. We start by splitting the integral over Ω into a sum of contributions from each element:

$$A_{i,j} = \int_{\Omega} \varphi_i \varphi_j \, dx = \sum_e A_{i,j}^{(e)}, \quad A_{i,j}^{(e)} = \int_{\Omega^{(e)}} \varphi_i \varphi_j \, dx. \quad (4.7)$$

Now, $A_{i,j}^{(e)} \neq 0$, if and only if, i and j are nodes in element e (look at Figure 4.13 to realize this property, but the result also holds for all types of elements). Introduce $i = q(e, r)$ as the mapping of local node number r in element e to the global node number i . This is just a short mathematical notation for the expression `i=elements[e][r]` in a program. Let r and s be the local node numbers corresponding to the global node numbers $i = q(e, r)$ and $j = q(e, s)$. With $d + 1$ nodes per element, all the nonzero matrix entries in $A_{i,j}^{(e)}$ arise from the integrals involving basis functions with indices corresponding to the global node numbers in element number e :

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

These contributions can be collected in a $(d + 1) \times (d + 1)$ matrix known as the *element matrix*. Let $I_d = \{0, \dots, d\}$ be the valid indices of r and s . We introduce the notation

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

for the element matrix. For P1 elements ($d = 2$) we have

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

while P2 elements have a 3×3 element matrix:

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

Assembly of element matrices. Given the numbers $\tilde{A}_{r,s}^{(e)}$, we should, according to (4.7), add the contributions to the global coefficient matrix by

$$A_{q(e,r),q(e,s)} := A_{q(e,r),q(e,s)} + \tilde{A}_{r,s}^{(e)}, \quad r, s \in I_d. \quad (4.8)$$

This process of adding in elementwise contributions to the global matrix is called *finite element assembly* or simply *assembly*.

Figure 4.16 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 1 and 2 assembled, assuming that global nodes are numbered from left to right in the domain. With regularly numbered P3 elements, where the element matrices have size 4×4 , the assembly of elements 1 and 2 are sketched in Figure 4.17.

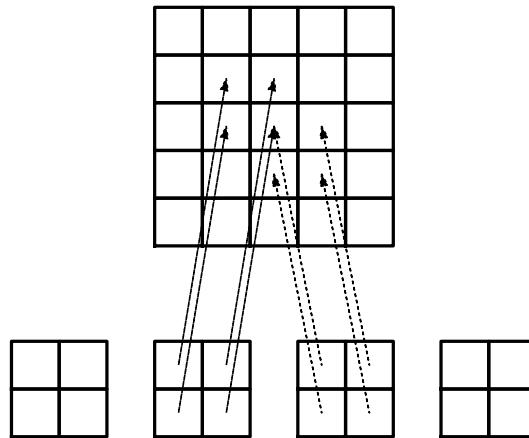


Fig. 4.16 Illustration of matrix assembly: regularly numbered P1 elements.

Assembly of irregularly numbered elements and nodes. After assembly of element matrices corresponding to regularly numbered elements and nodes are understood, it is wise to study the assembly process for irregularly numbered elements and nodes. Figure 4.5 shows a mesh where the `elements` array, or $q(e, r)$ mapping in mathematical notation, is given as

```
elements = [[2, 1], [4, 5], [0, 4], [3, 0], [5, 2]]
```

The associated assembly of element matrices 1 and 2 is sketched in Figure 4.18.

We have created [animations](#) to illustrate the assembly of P1 and P3 elements with regular numbering as well as P1 elements with irregular numbering. The reader is encouraged to develop a “geometric” understanding of how element matrix entries are added to the global matrix.

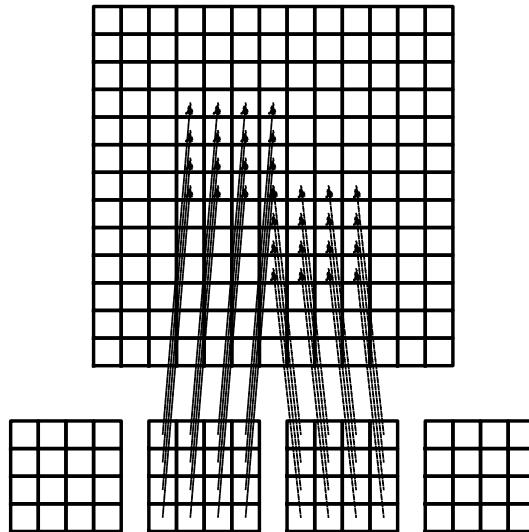


Fig. 4.17 Illustration of matrix assembly: regularly numbered P3 elements.

This understanding is crucial for hand computations with the finite element method.

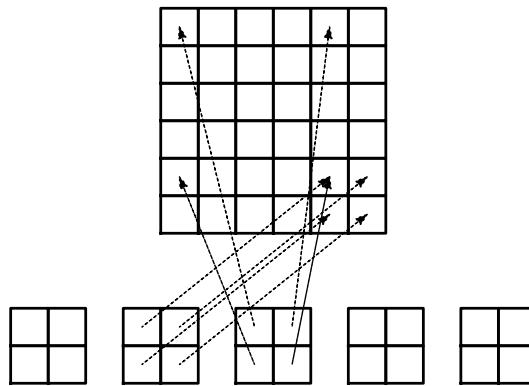


Fig. 4.18 Illustration of matrix assembly: irregularly numbered P1 elements.

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

$$b_i = \int_{\Omega} f(x) \varphi_i(x) dx = \sum_e b_i^{(e)}, \quad b_i^{(e)} = \int_{\Omega^{(e)}} f(x) \varphi_i(x) dx. \quad (4.9)$$

We observe that $b_i^{(e)} \neq 0$ if and only if global node i is a node in element e (look at Figure 4.14 to realize this property). With d nodes per element we can collect the $d + 1$ nonzero contributions $b_i^{(e)}$, for $i = q(e, r)$, $r \in I_d$, in an *element vector*

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

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

$$b_{q(e,r)} := b_{q(e,r)} + \tilde{b}_r^{(e)}, \quad r \in I_d. \quad (4.10)$$

4.1.8 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 $\Omega^{(e)} = [x_L, x_R]$ in the physical coordinate system, it turns out that it is considerably easier and more convenient to map the element domain $[x_L, x_R]$ to a standardized reference element domain $[-1, 1]$ and compute all integrals over the same 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, regular numbering of nodes and elements from left to right through the domain, we have $x_L = x_e$ and $x_R = x_{e+1}$ for P1 elements.

The coordinate transformation. Let $X \in [-1, 1]$ be the coordinate in the reference element. A linear mapping, also known as an affine mapping, from X to x can be written

$$x = \frac{1}{2}(x_L + x_R) + \frac{1}{2}(x_R - x_L)X. \quad (4.11)$$

This relation can alternatively be expressed as

$$x = x_m + \frac{1}{2}hX, \quad (4.12)$$

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

Formulas for the element matrix and vector entries. Integrating over the reference element is a matter of just changing the integration variable from x to X . Let

$$\tilde{\varphi}_r(X) = \varphi_{q(e,r)}(x(X)) \quad (4.13)$$

be the basis function associated with local node number r in the reference element. Switching from x to X as integration variable, using the rules from calculus, results in

$$\tilde{A}_{r,s}^{(e)} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}(x) \varphi_{q(e,s)}(x) dx = \int_{-1}^1 \tilde{\varphi}_r(X) \tilde{\varphi}_s(X) \frac{dx}{dX} dX. \quad (4.14)$$

In 2D and 3D, dx is transformed to $\det J dX$, where J is the Jacobian of the mapping from x to X . In 1D, $\det J dX = dx/dX = h/2$. To obtain a uniform notation for 1D, 2D, and 3D problems we therefore replace dx/dX by $\det J$ now. The integration over the reference element is then written as

$$\tilde{A}_{r,s}^{(e)} = \int_{-1}^1 \tilde{\varphi}_r(X) \tilde{\varphi}_s(X) \det J dX. \quad (4.15)$$

The corresponding formula for the element vector entries becomes

$$\tilde{b}_r^{(e)} = \int_{\Omega^{(e)}} f(x) \varphi_{q(e,r)}(x) dx = \int_{-1}^1 f(x(X)) \tilde{\varphi}_r(X) \det J dX. \quad (4.16)$$

Why reference elements?

The great advantage of using reference elements is that the formulas for the basis functions, $\tilde{\varphi}_r(X)$, are the same for all elements and independent of the element geometry (length and location in the mesh). The geometric information is “factored out” in the simple mapping formula and the associated $\det J$ quantity. Also, the integration domain is the same for all elements. All these features contribute to simplify computer codes and make them more general.

Formulas for local basis functions. 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

$$\tilde{\varphi}_0(X) = \frac{1}{2}(1 - X), \quad (4.17)$$

$$\tilde{\varphi}_1(X) = \frac{1}{2}(1 + X). \quad (4.18)$$

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

$$\tilde{\varphi}_0(X) = \frac{1}{2}(X - 1)X, \quad (4.19)$$

$$\tilde{\varphi}_1(X) = 1 - X^2, \quad (4.20)$$

$$\tilde{\varphi}_2(X) = \frac{1}{2}(X + 1)X. \quad (4.21)$$

In general,

$$\tilde{\varphi}_r(X) = \prod_{s=0, s \neq r}^d \frac{X - X_{(s)}}{X_{(r)} - X_{(s)}}, \quad (4.22)$$

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

4.1.9 Example on 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 $d = 1$ (P1 elements) and $f(x) = x(1 - x)$ on $\Omega = [0, 1]$. We have the general expressions (4.15) and (4.16) for $\tilde{A}_{r,s}^{(e)}$ and $\tilde{b}_r^{(e)}$. Writing these out for the choices (4.17) and (4.18), and using that $\det J = h/2$, we can do the following calculations of the element matrix entries:

$$\begin{aligned}\tilde{A}_{0,0}^{(e)} &= \int_{-1}^1 \tilde{\varphi}_0(X) \tilde{\varphi}_0(X) \frac{h}{2} dX \\ &= \int_{-1}^1 \frac{1}{2}(1-X) \frac{1}{2}(1-X) \frac{h}{2} dX = \frac{h}{8} \int_{-1}^1 (1-X)^2 dX = \frac{h}{3},\end{aligned}\quad (4.23)$$

$$\begin{aligned}\tilde{A}_{1,0}^{(e)} &= \int_{-1}^1 \tilde{\varphi}_1(X) \tilde{\varphi}_0(X) \frac{h}{2} dX \\ &= \int_{-1}^1 \frac{1}{2}(1+X) \frac{1}{2}(1-X) \frac{h}{2} dX = \frac{h}{8} \int_{-1}^1 (1-X^2) dX = \frac{h}{6},\end{aligned}\quad (4.24)$$

$$\tilde{A}_{0,1}^{(e)} = \tilde{A}_{1,0}^{(e)}, \quad (4.25)$$

$$\begin{aligned}\tilde{A}_{1,1}^{(e)} &= \int_{-1}^1 \tilde{\varphi}_1(X) \tilde{\varphi}_1(X) \frac{h}{2} dX \\ &= \int_{-1}^1 \frac{1}{2}(1+X) \frac{1}{2}(1+X) \frac{h}{2} dX = \frac{h}{8} \int_{-1}^1 (1+X)^2 dX = \frac{h}{3}.\end{aligned}\quad (4.26)$$

The corresponding entries in the element vector becomes using (4.12))

$$\begin{aligned}\tilde{b}_0^{(e)} &= \int_{-1}^1 f(x(X)) \tilde{\varphi}_0(X) \frac{h}{2} dX \\ &= \int_{-1}^1 (x_m + \frac{1}{2}hX)(1 - (x_m + \frac{1}{2}hX)) \frac{1}{2}(1-X) \frac{h}{2} dX \\ &= -\frac{1}{24}h^3 + \frac{1}{6}h^2x_m - \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m,\end{aligned}\quad (4.27)$$

$$\begin{aligned}\tilde{b}_1^{(e)} &= \int_{-1}^1 f(x(X)) \tilde{\varphi}_1(X) \frac{h}{2} dX \\ &= \int_{-1}^1 (x_m + \frac{1}{2}hX)(1 - (x_m + \frac{1}{2}hX)) \frac{1}{2}(1+X) \frac{h}{2} dX \\ &= -\frac{1}{24}h^3 - \frac{1}{6}h^2x_m + \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m.\end{aligned}\quad (4.28)$$

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 involve much more algebra, but `sympy` may help. For example, we can easily calculate (4.23), (4.24), and (4.27) by

```
>>> import sympy as sym
>>> x, x_m, h, X = sym.symbols('x x_m h X')
>>> sym.integrate(h/8*(1-X)**2, (X, -1, 1))
h/3
>>> sym.integrate(h/8*(1+X)*(1-X), (X, -1, 1))
h/6
>>> x = x_m + h/2*X
```

```
>>> b_0 = sym.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
```

4.2 Implementation

Based on the experience from the previous example, it makes sense to write some code to automate the analytical integration process for any choice of finite element basis functions. In addition, we can automate the assembly process and the solution of the linear system. Another advantage is that the code for these purposes document all details of all steps in the finite element computational machinery. The complete code can be found in the module file `fe_approx1D.py`.

4.2.1 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 sym
import numpy as np

def basis(d, point_distribution='uniform', symbolic=False):
    """
        Return all local basis function phi as functions of the
        local point X in a 1D element with d+1 nodes.
        If symbolic=True, return symbolic expressions, else
        return Python functions of X.
        point_distribution can be 'uniform' or 'Chebyshev'.
    """
    X = sym.symbols('X')
    if d == 0:
        phi_sym = [1]
    else:
        if point_distribution == 'uniform':
            if symbolic:
                # Compute symbolic nodes
                h = sym.Rational(1, d)  # node spacing
                nodes = [2*i*h - 1 for i in range(d+1)]
            else:
                nodes = np.linspace(-1, 1, d+1)
        elif point_distribution == 'Chebyshev':
            # Just numeric nodes
            nodes = Chebyshev_nodes(-1, 1, d)
```

```

phi_sym = [Lagrange_polynomial(X, r, nodes)
           for r in range(d+1)]
# Transform to Python functions
phi_num = [sym.lambdify([X], phi_sym[r], modules='numpy')
           for r in range(d+1)]
return phi_sym if symbolic else phi_num

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_sym` list to be symbolic expressions for $\tilde{\varphi}_r(X)$ with `X` as a `Symbol` object from `sympy`. Also note that the `Lagrange_polynomial` function (here simply copied from Section 3.3.2) works with both symbolic and numeric variables.

Now we can write the function that computes the element matrix with a list of symbolic expressions for φ_r (`phi = basis(d, symbolic=True)`):

```

def element_matrix(phi, Omega_e, symbolic=True):
    n = len(phi)
    A_e = sym.zeros(n, n)
    X = sym.Symbol('X')
    if symbolic:
        h = sym.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] = sym.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, symbolic=True)
>>> phi
[-X/2 + 1/2, X/2 + 1/2]
>>> 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, 0.01666666666666667]
[0.01666666666666667, 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 = sym.zeros(n, 1)
    # Make f a function of X
    X = sym.Symbol('X')
    if symbolic:
        h = sym.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] = sym.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 can be integrated in terms of X , cf. the formula $\tilde{b}_r^{(e)} = \int_{-1}^1 f(x(X))\tilde{\varphi}_r(X)\frac{h}{2} dX$.

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 or expression as when symbolic integration is successful. It may therefore be wise to introduce a fall back to the numerical integration. The symbolic integration can also spend considerable time before reaching an unsuccessful conclusion, so we may also introduce a parameter `symbolic` to turn symbolic integration on and off:

```
def element_vector(f, phi, Omega_e, symbolic=True):
    ...
    if symbolic:
        I = sym.integrate(f*phi[r]*detJ, (X, -1, 1))
    if not symbolic or isinstance(I, sym.Integral):
        h = Omega_e[1] - Omega_e[0] # Ensure h is numerical
        detJ = h/2
        integrand = sym.lambdify([X], f*phi[r]*detJ, 'mpmath')
        I = mpmath.quad(integrand, [-1, 1])
    b_e[r] = I
    ...
```

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.

4.2.2 Linear system assembly and solution

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

```
def assemble(nodes, elements, phi, f, symbolic=True):
    N_n, N_e = len(nodes), len(elements)
    if symbolic:
        A = sym.zeros(N_n, N_n)
        b = sym.zeros(N_n, 1)      # note: (N_n, 1) matrix
    else:
        A = np.zeros((N_n, N_n))
        b = np.zeros(N_n)
    for e in range(N_e):
        Omega_e = [nodes[elements[e][0]], nodes[elements[e][-1]]]

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

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

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

Given the coefficient matrix A and the right-hand side b , we can compute the coefficients $\{c_j\}_{j \in \mathcal{I}_s}$ 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, the solution procedure implied by `A.LUsolve` is symbolic. Otherwise, A and b are `numpy` arrays and 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.

4.2.3 Example on computing symbolic approximations

We can exemplify the use of `assemble` on the computational case from Section 4.1.6 with two P1 elements (linear basis functions) on the domain $\Omega = [0, 1]$. Let us first work with a symbolic element length:

```
>>> h, x = sym.symbols('h x')
>>> nodes = [0, h, 2*h]
>>> elements = [[0, 1], [1, 2]]
>>> phi = basis(d=1, symbolic=True)
>>> 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)]
```

4.2.4 Using interpolation instead of least squares

As an alternative to the least squares formulation, we may compute the c vector based on the interpolation method from Section 3.4.1, using finite element basis functions. Choosing the nodes as interpolation points, the method can be written as

$$u(x_i) = \sum_{j \in \mathcal{I}_s} c_j \varphi_j(x_i) = f(x_i), \quad i \in \mathcal{I}_s.$$

The coefficient matrix $A_{i,j} = \varphi_j(x_i)$ becomes the identity matrix because basis function number j vanishes at all nodes, except node i : $\varphi_j(x_i) = \delta_{ij}$. Therefore, $c_i = f(x_i)$.

The associated `sympy` calculations are

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

These expressions are much simpler than those based on least squares or projection in combination with finite element basis functions. However,

which of the two methods that is most appropriate for a given task is problem-dependent, so we need both methods in our toolbox.

4.2.5 Example on computing numerical approximations

The numerical computations corresponding to the symbolic ones in Section 4.2.3 (still done by `sympy` and the `assemble` function) go as follows:

```
>>> nodes = [0, 0.5, 1]
>>> elements = [[0, 1], [1, 2]]
>>> phi = basis(d=1, symbolic=True)
>>> x = sym.Symbol('x')
>>> f = x*(1-x)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=False)
>>> A
[ 0.1666666666666667, 0.0833333333333333, 0]
[0.0833333333333333, 0.3333333333333333, 0.0833333333333333]
[          0, 0.0833333333333333, 0.1666666666666667]
>>> b
[ 0.03125]
[0.1041666666666667]
[ 0.03125]
>>> c = A.LUsolve(b)
>>> c
[0.0416666666666666]
[ 0.2916666666666667]
[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` (`symbolic=True`) or by real numbers (`symbolic=False`):

```
nodes, elements = mesh_uniform(N_e=10, d=3, Omega=[0,1],
                               symbolic=True)
```

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`, $u(x) = \sum_j c_j \varphi_j(x)$ can be computed at a (large) number of points and plotted together with $f(x)$. The construction of the pointwise function u 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 in the local coordinate system, and the discrete (x, u) values on each element are stored in separate arrays that are finally concatenated to form a global array for x and for u . The details are found in the `u_glob` function in `fe_approx1D.py`.

4.2.6 The structure of the coefficient matrix

Let us first see how the global matrix looks 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:

$$A = \frac{h}{6} \begin{pmatrix} 2 & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 1 & 4 & 1 & \ddots & & & \vdots \\ 0 & 1 & 4 & 1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & 0 & 1 & 4 & 1 & \ddots \\ \vdots & & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & \ddots & 1 & 4 & 1 \\ 0 & \cdots & \cdots & \cdots & 0 & 1 & 2 \end{pmatrix} \quad (4.29)$$

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 $f(x)$. Numerical integration can give some insight into the nature of the right-hand side. For this purpose it is easier to look at the integration in x coordinates, which gives the general formula (4.6). For equal-sized elements of length h , we can apply the Trapezoidal rule at the global node points to arrive at

$$b_i = h \left(\frac{1}{2} \varphi_i(x_0)f(x_0) + \frac{1}{2} \varphi_i(x_N)f(x_N) + \sum_{j=1}^{N-1} \varphi_i(x_j)f(x_j) \right),$$

which leads to

$$b_i = \begin{cases} \frac{1}{2}hf(x_i), & i = 0 \text{ or } i = N, \\ hf(x_i), & 1 \leq i \leq N-1 \end{cases} \quad (4.30)$$

The reason for this simple formula is just 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

$$A^{(e)} = \frac{h}{30} \begin{pmatrix} 4 & 2 & -1 \\ 2 & 16 & 2 \\ -1 & 2 & 4 \end{pmatrix}, \quad (4.31)$$

and the following global matrix, assembled here from four elements:

$$A = \frac{h}{30} \begin{pmatrix} 4 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 16 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & 8 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 16 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & 8 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 16 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & 8 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 16 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 & 4 \end{pmatrix}. \quad (4.32)$$

In general, for i odd we have the nonzero elements

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

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

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

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 only one element contributes to the values in the global matrix.

4.2.7 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 4.19 shows how linear and quadratic elements approximate 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 sym
from fe_approx1D import approximate
x = sym.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 refine the mesh and the function becomes properly resolved.

4.2.8 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 $\int f(x(X))\tilde{\varphi}_r(X)h/2 dx$ before giving up, and the program then 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`.

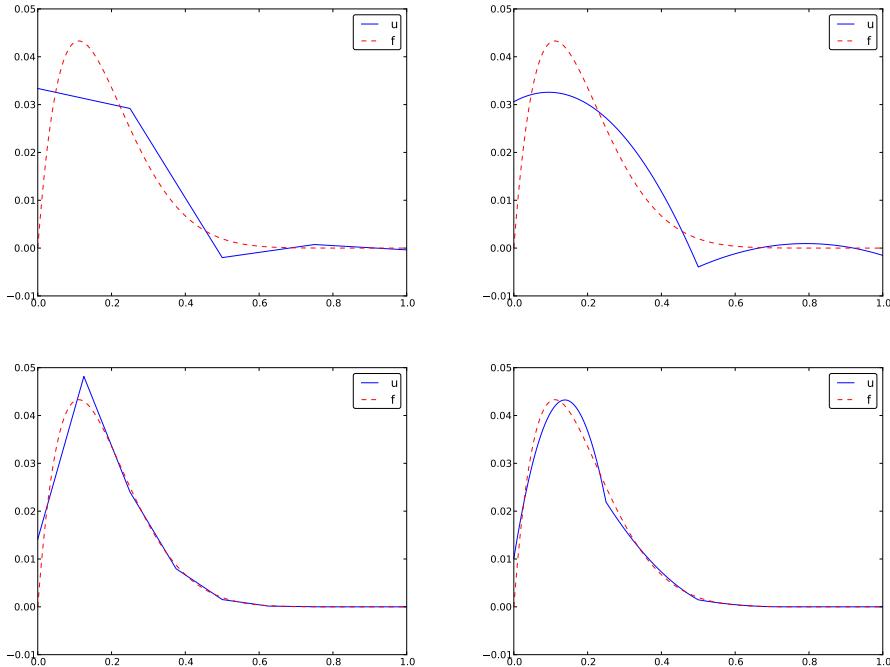


Fig. 4.19 Comparison of the finite element approximations: 4 P1 elements with 5 nodes (upper left), 2 P2 elements with 5 nodes (upper right), 8 P1 elements with 9 nodes (lower left), and 4 P2 elements with 9 nodes (lower right).

There is also another (potential) reason for slow code: the solution algorithm for the linear system performs much more work than necessary. Most of the matrix entries $A_{i,j}$ are zero, because $(\varphi_i, \varphi_j) = 0$ unless i and j are nodes in the same element. In 1D problems, we do not need to store or compute with these zeros when solving the linear system, but that requires solution methods adapted to the kind of matrices produced by the finite element approximations.

A matrix whose majority of entries are zeros, is known as a [sparse](#) matrix. Utilizing sparsity in software dramatically decreases the storage demands and the CPU-time needed to compute the solution of the linear system. This optimization is not very 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. One of the advantageous features of the finite element method is that it produces very sparse matrices. The reason is that the basis functions have local

support such that the product of two basis functions, as typically met in integrals, is mostly zero.

Using a numbering of nodes and elements from left to right over a 1D domain, the assembled coefficient matrix has only a few diagonals different from zero. More precisely, $2d + 1$ diagonals around the main diagonal are different from zero, where d is the order of the polynomial. 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 4.20 and 4.21 exemplify sparsity patterns.

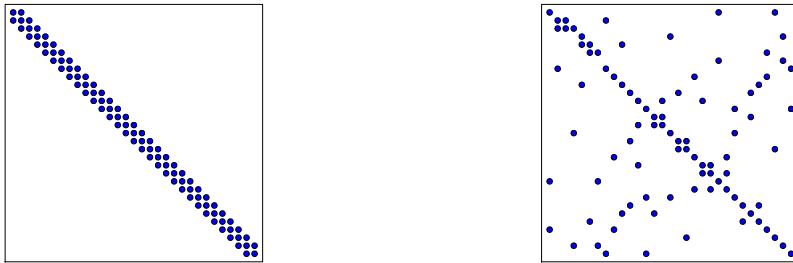


Fig. 4.20 Matrix sparsity pattern for left-to-right numbering (left) and random numbering (right) of nodes in P1 elements.



Fig. 4.21 Matrix sparsity pattern for left-to-right numbering (left) and random numbering (right) of nodes in P3 elements.

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

- `scipy.sparse.diags` for matrix defined via diagonals
- `scipy.sparse.dok_matrix` for matrix incrementally defined via index pairs (i, j)

The `dok_matrix` object is most convenient for finite element computations. This sparse matrix format is called DOK, which stands for

Dictionary Of Keys: the implementation is basically a dictionary (hash) with the entry indices (i, j) as keys.

Rather than declaring $A = np.zeros((N_n, N_n))$, a DOK sparse matrix is created by

```
import scipy.sparse
A = scipy.sparse.dok_matrix((N_n, N_n))
```

When there is any need to set or add some matrix entry i, j , just do

```
A[i,j] = entry
# or
A[i,j] += entry
```

The indexing creates the matrix entry on the fly, and only the nonzero entries in the matrix will be stored.

To solve a system with right-hand side b (one-dimensional numpy array) with a sparse coefficient matrix A , we must use some kind of a sparse linear system solver. The safest choice is a method based on sparse Gaussian elimination. One high-quality package for this purpose is [UMFPACK](#). It is interfaced from SciPy by

```
import scipy.sparse.linalg
c = scipy.sparse.linalg.spsolve(A.tocsr(), b, use_umfpack=True)
```

The call $A.tocsr()$ is not strictly needed (a warning is issued otherwise), but ensures that the solution algorithm can efficiently work with a copy of the sparse matrix in *Compressed Sparse Row* (CSR) format.

An advantage of the `scipy.sparse.diags` matrix over the DOK format is that the former allows vectorized assignment to the matrix. Vectorization is possible for approximation problems when all elements are of the same type. However, when solving differential equations, vectorization may be more difficult in particular because of boundary conditions. It also appears that the DOK sparse matrix format available in the `scipy.sparse` package is fast enough even for big 1D problems on today's laptops, so the need for improving efficiency occurs only in 2D and 3D problems, but then the complexity of the mesh favors the DOK format.

4.3 Comparison of finite elements and finite differences

The previous sections on approximating f by a finite element function u utilize the projection/Galerkin or least squares approaches to minimize

the approximation error. We may, alternatively, use the collocation/interpolation method as described in Section 4.2.4. Here we shall compare these three approaches with what one does in the finite difference method when representing a given function on a mesh.

4.3.1 Finite difference approximation of given functions

Approximating a given function $f(x)$ on a mesh in a finite difference context will typically just sample f at the mesh points. If u_i is the value of the approximate u at the mesh point x_i , we have $u_i = f(x_i)$. The collocation/interpolation method using finite element basis functions gives exactly the same representation, as shown Section 4.2.4,

$$u(x_i) = c_i = f(x_i).$$

How does a finite element Galerkin or least squares approximation differ from this straightforward interpolation of f ? This is the question to be addressed next. We now limit the scope to P1 elements since this is the element type that gives formulas closest to those arising in the finite difference method.

4.3.2 Interpretation of a finite element approximation in terms of finite difference operators

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

$$\sum_{j \in \mathcal{I}_s} c_j(\psi_i, \psi_j) = (f, \psi_i), \quad i \in \mathcal{I}_s.$$

In the finite element approximation we choose $\psi_i = \varphi_i$. With φ_i corresponding to P1 elements and a uniform mesh of element length h we have in Section 4.1.6 calculated the matrix with entries (φ_i, φ_j) . Equation number i reads

$$\frac{h}{6}(u_{i-1} + 4u_i + u_{i+1}) = (f, \varphi_i). \quad (4.33)$$

The first and last equation, corresponding to $i = 0$ and $i = N$ are slightly different, see Section 4.2.6.

The finite difference counterpart to (4.33) is just $u_i = f_i$ as explained in Section 4.3.1. To more easier compare this result to the finite element

approach to approximating functions, we can rewrite the left-hand side of (4.33) as

$$h\left(u_i + \frac{1}{6}(u_{i-1} - 2u_i + u_{i+1})\right). \quad (4.34)$$

Thinking in terms of finite differences, we can write this expression using finite difference operator notation:

$$[h(u + \frac{h^2}{6}D_x D_x u)]_i,$$

which is nothing but the standard discretization of (see also Appendix A.1)

$$h\left(u + \frac{h^2}{6}u''\right).$$

Before interpreting the approximation procedure as solving a differential equation, we need to work out what the right-hand side is in the context of P1 elements. Since φ_i is the linear function that is 1 at x_i and zero at all other nodes, only the interval $[x_{i-1}, x_{i+1}]$ contribute to the integral on the right-hand side. This integral is naturally split into two parts according to (4.4):

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

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

To proceed with the right-hand side, we can turn to numerical integration schemes. The Trapezoidal method for (f, φ_i) , based on sampling the integrand $f\varphi_i$ at the node points $x_i = ih$ gives

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

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

$$(f, \varphi_i) \approx h f(x_i), \quad (4.35)$$

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

$$(f, \varphi_i) \approx \frac{1}{2}hf(x_i), \quad i = 0, \quad i = N. \quad (4.36)$$

Simpson's rule with sample points also in the middle of the elements, at $x_{i+\frac{1}{2}} = (x_i + x_{i+1})/2$, can be written as

$$\int_{\Omega} g(x) dx \approx \frac{\tilde{h}}{3} \left(g(x_0) + 2 \sum_{j=1}^{N-1} g(x_j) + 4 \sum_{j=0}^{N-1} g(x_{j+\frac{1}{2}}) + f(x_{2N}) \right),$$

where $\tilde{h} = h/2$ is the spacing between the sample points. Our integrand is $g = f\varphi_i$. For all the node points, $\varphi_i(x_j) = \delta_{ij}$, and therefore $\sum_{j=1}^{N-1} f(x_j)\varphi_i(x_j) = f(x_i)$. At the midpoints, $\varphi_i(x_{i\pm\frac{1}{2}}) = 1/2$ and $\varphi_i(x_{j+\frac{1}{2}}) = 0$ for $j > 1$ and $j < i - 1$. Consequently,

$$\sum_{j=0}^{N-1} f(x_{j+\frac{1}{2}})\varphi_i(x_{j+\frac{1}{2}}) = \frac{1}{2}(f(x_{i-\frac{1}{2}}) + f(x_{i+\frac{1}{2}})).$$

When $1 \leq i \leq N - 1$ we then get

$$(f, \varphi_i) \approx \frac{h}{3}(f_{i-\frac{1}{2}} + f_i + f_{i+\frac{1}{2}}). \quad (4.37)$$

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

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

$$u + \frac{h^2}{6}u'' = f, \quad u'(0) = u'(L) = 0, \quad (4.38)$$

expressed with operator notation as

$$[u + \frac{h^2}{6}D_x D_x u = f]_i. \quad (4.39)$$

As $h \rightarrow 0$, the extra term proportional to u'' goes to zero, and the two methods converge to the same solution.

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

$$[u + \frac{h^2}{6} D_x D_x u = \bar{f}]_i, \quad (4.40)$$

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

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

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

4.3.3 Making finite elements behave as finite differences

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

Computations in physical space. We have already seen that applying the Trapezoidal rule to the right-hand side (f, φ_i) simply gives f sampled at x_i . Using the Trapezoidal rule on the matrix entries $A_{i,j} = (\varphi_i, \varphi_j)$ involves a sum

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

but $\varphi_i(x_k) = \delta_{ik}$ and $\varphi_j(x_k) = \delta_{jk}$. The product $\varphi_i \varphi_j$ is then different from zero only when sampled at x_i and $i = j$. The Trapezoidal approximation to the integral is then

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

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

$$c_i = f_i, \quad i \in \mathcal{I}_s.$$

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

Elementwise computations. Identical results to those above will arise if we perform elementwise computations. The idea is to use the Trapezoidal rule on the reference element for computing the element matrix and vector. When assembled, the same equations $c_i = f(x_i)$ arise. Exercise 4.10 encourages you to carry out the details.

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

4.4 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, corresponding to a $\varphi_i(x)$ function that is 1 on element number i and zero on all other elements. 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.

In order to get a richer space of finite element approximations, we need to revise the simple node and element concept presented so far and introduce a more powerful terminology. Much literature employs the definition of node and element introduced in the previous sections so it is important have this knowledge, besides being a good pedagogical background for understanding the extended element concept in the following.

4.4.1 Cells, vertices, and degrees of freedom

We now introduce *cells* as the subdomains $\Omega^{(e)}$ previously referred to as elements. The cell boundaries are uniquely defined in terms of *vertices*. This applies to cells in both 1D and higher dimensions. We also define a set of *degrees of freedom* (dof), 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 of the geometry (cell, vertices) from the space of basis functions. We will 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.

4.4.2 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 how basis functions from different elements are glued together across element interfaces. A common technique is to choose the basis functions such that $\tilde{\varphi}_i = 1$ for degree of freedom number i that is associated with nodal point x_i and $\tilde{\varphi}_i = 0$ for all other degrees of freedom. This technique ensures the desired continuity;
- a mapping between local and global degree of freedom numbers, here called the *dof map*;
- a geometric *mapping* of the reference cell onto the 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 .

The expansion of u over one cell is often used:

$$u(x) = \tilde{u}(X) = \sum_r c_r \tilde{\varphi}_r(X), \quad x \in \Omega^{(e)}, \quad X \in [-1, 1], \quad (4.41)$$

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 .

Our previous P1, P2, etc., elements are defined by introducing $d + 1$ equally spaced nodes in the reference cell, a polynomial space (P_d) containing a complete set of polynomials of order d , 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 (4.11) as before.

Notice

The extended finite element concept introduced above is quite general and has served as a successful recipe for implementing many finite element frameworks and for developing the theory behind. Here, we have seen several different examples but the exposition is most focused on 1D examples and the diversity is limited as many

of the different methods in 2D and 3D collapse to the same method in 1D. The curious reader is advised to for instance look into the numerous examples of finite elements implemented in FEniCS [23] to gain insight into the variety of methods that exists.

4.4.3 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 for P_d elements).

Consider the example from Section 4.1.1 where $\Omega = [0, 1]$ is divided into two cells, $\Omega^{(0)} = [0, 0.4]$ and $\Omega^{(1)} = [0.4, 1]$, as depicted in Figure 4.4. 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], [2, 3, 4]]
```

If we approximate f by piecewise constants, known as P0 elements, we simply introduce one point or node in an element, preferably $X = 0$, and define one degree of freedom, which is the function value at this node. Moreover, we set $\tilde{\varphi}_0(X) = 1$. The `cells` and `vertices` arrays remain the same, but `dof_map` is altered:

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

We use the `cells` and `vertices` lists to retrieve information on the geometry of a cell, while `dof_map` is the $q(e, r)$ mapping introduced earlier 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 drop the `nodes` and `elements` arrays and work exclusively with `cells`, `vertices`, and `dof_map`. The module `fe_approx1D_numint.py` now replaces the module `fe_approx1D` and offers similar functions that work with the new concepts:

```
from fe_approx1D_numint import *
x = sym.Symbol('x')
f = x*(1 - x)
N_e = 10
vertices, cells, dof_map = mesh_uniform(N_e, d=3, Omega=[0,1])
phi = [basis(len(dof_map[e])-1) for e in range(N_e)]
A, b = assemble(vertices, cells, dof_map, phi, f)
c = np.linalg.solve(A, b)
# Make very fine mesh and sample u(x) on this mesh for plotting
x_u, u = u_glob(c, vertices, cells, dof_map,
                  resolution_per_element=51)
plot(x_u, u)
```

These steps are offered in the `approximate` function, which we here apply to see how well four P0 elements (piecewise constants) can approximate a parabola:

```
from fe_approx1D_numint import *
x=sym.Symbol("x")
for N_e in 4, 8:
    approximate(x*(1-x), d=0, N_e=N_e, Omega=[0,1])
```

Figure 4.22 shows the result.

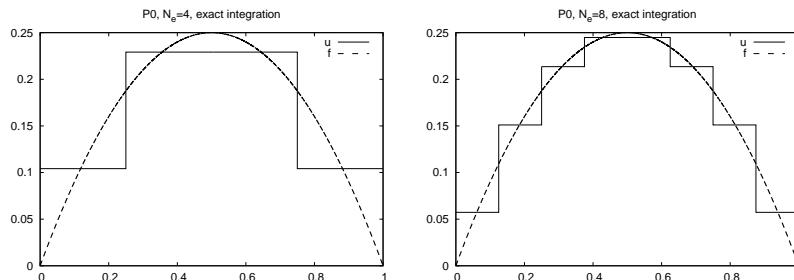


Fig. 4.22 Approximation of a parabola by 4 (left) and 8 (right) P0 elements.

4.4.4 Computing the error of the approximation

So far we have focused on computing the coefficients c_j in the approximation $u(x) = \sum_j c_j \varphi_j$ as well as on plotting u and f for visual comparison. A more quantitative comparison needs to investigate the error $e(x) = f(x) - u(x)$. We mostly want a single number to reflect the error and use a norm for this purpose, usually the L^2 norm

$$\|e\|_{L^2} = \left(\int_{\Omega} e^2 dx \right)^{1/2}.$$

Since the finite element approximation is defined for all $x \in \Omega$, and we are interested in how $u(x)$ deviates from $f(x)$ through all the elements, we can either integrate analytically or use an accurate numerical approximation. The latter is more convenient as it is a generally feasible and simple approach. The idea is to sample $e(x)$ at a large number of points in each element. The function `u_glob` in the `fe_approx1D_numint` module does this for $u(x)$ and returns an array `x` with coordinates and an array `u` with the u values:

```
x, u = u_glob(c, vertices, cells, dof_map,
                 resolution_per_element=101)
e = f(x) - u
```

Let us use the Trapezoidal method to approximate the integral. Because different elements may have different lengths, the `x` array may have a non-uniformly distributed set of coordinates. Also, the `u_glob` function works in an element by element fashion such that coordinates at the boundaries between elements appear twice. We therefore need to use a "raw" version of the Trapezoidal rule where we just add up all the trapezoids:

$$\int_{\Omega} g(x) dx \approx \sum_{j=0}^{n-1} \frac{1}{2}(g(x_j) + g(x_{j+1}))(x_{j+1} - x_j),$$

if x_0, \dots, x_n are all the coordinates in `x`. In vectorized Python code,

```
g_x = g(x)
integral = 0.5*np.sum((g_x[:-1] + g_x[1:])* (x[1:] - x[:-1]))
```

Computing the L^2 norm of the error, here named `E`, is now achieved by

```
e2 = e**2
E = np.sqrt(0.5*np.sum((e2[:-1] + e2[1:])* (x[1:] - x[:-1])))
```

How does the error depend on h and d ?

Theory and experiments show that the least squares or projection/-Galerkin method in combination with Pd elements of equal length h has an error

$$\|e\|_{L^2} = C|f^{(d+1)}|h^{d+1}, \quad (4.42)$$

where C is a constant depending on d and $\Omega = [0, L]$, but not on h , and the norm $|f^{(d+1)}|$ is defined through

$$|f^{(d+1)}|^2 = \int_0^L \left(\frac{d^{d+1} f}{dx^{d+1}} \right)^2 dx.$$

4.4.5 Example on cubic Hermite polynomials

The finite elements considered so far represent u as piecewise polynomials with discontinuous derivatives at the cell boundaries. Sometimes it is desirable to have continuous derivatives. A primary example 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 so-called cubic Hermite polynomials. The construction of such polynomials, as explained next, will further exemplify the concepts of a cell, vertex, degree of freedom, and dof map.

Given a reference cell $[-1, 1]$, we seek cubic polynomials with the values of the *function* and its *first-order derivative* at $X = -1$ and $X = 1$ as the four degrees of freedom. Let us number the degrees of freedom as

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

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

The four basis functions can be written in a general form

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

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

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

These four 4×4 linear equations can be solved, yielding the following formulas for the cubic basis functions:

$$\tilde{\varphi}_0(X) = 1 - \frac{3}{4}(X+1)^2 + \frac{1}{4}(X+1)^3 \quad (4.43)$$

$$\tilde{\varphi}_1(X) = -(X+1)\left(1 - \frac{1}{2}(X+1)\right)^2 \quad (4.44)$$

$$\tilde{\varphi}_2(X) = \frac{3}{4}(X+1)^2 - \frac{1}{2}(X+1)^3 \quad (4.45)$$

$$\tilde{\varphi}_3(X) = -\frac{1}{2}(X+1)\left(\frac{1}{2}(X+1)^2 - (X+1)\right) \quad (4.46)$$

$$(4.47)$$

The construction of the dof map needs a scheme for numbering the global degrees of freedom. A natural left-to-right numbering has the function value at vertex x_i as degree of freedom number $2i$ and the value of the derivative at x_i as degree of freedom number $2i+1$, $i = 0, \dots, N_e+1$.

4.5 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.

Numerical integration rules can be expressed in a common form,

$$\int_{-1}^1 g(X) dX \approx \sum_{j=0}^M w_j g(\bar{X}_j), \quad (4.48)$$

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

The very simplest method is the *Midpoint rule*,

$$\int_{-1}^1 g(X) dX \approx 2g(0), \quad \bar{X}_0 = 0, \quad w_0 = 2, \quad (4.49)$$

which integrates linear functions exactly.

4.5.1 Newton-Cotes rules

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

$$\int_{-1}^1 g(X) dX \approx g(-1) + g(1), \quad \bar{X}_0 = -1, \quad \bar{X}_1 = 1, \quad w_0 = w_1 = 1, \quad (4.50)$$

and *Simpson's rule*,

$$\int_{-1}^1 g(X) dX \approx \frac{1}{3} (g(-1) + 4g(0) + g(1)), \quad (4.51)$$

where

$$\bar{X}_0 = -1, \quad \bar{X}_1 = 0, \quad \bar{X}_2 = 1, \quad w_0 = w_2 = \frac{1}{3}, \quad w_1 = \frac{4}{3}. \quad (4.52)$$

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

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

4.5.2 Gauss-Legendre rules with optimized points

More accurate rules, for a given M , arise if the location of the integration points are optimized for polynomial integrands. The [Gauss-Legendre rules](#) (also known as Gauss-Legendre quadrature or Gaussian quadrature) constitute one such class of integration methods. Two widely applied Gauss-Legendre rules in this family have the choice

$$M = 1 : \quad \bar{X}_0 = -\frac{1}{\sqrt{3}}, \quad \bar{X}_1 = \frac{1}{\sqrt{3}}, \quad w_0 = w_1 = 1 \quad (4.53)$$

$$M = 2 : \quad \bar{X}_0 = -\sqrt{\frac{3}{5}}, \quad \bar{X}_1 = 0, \quad \bar{X}_2 = \sqrt{\frac{3}{5}}, \quad w_0 = w_2 = \frac{5}{9}, \quad w_1 = \frac{8}{9}. \quad (4.54)$$

These rules integrate 3rd and 5th degree polynomials exactly. In general, an M -point Gauss-Legendre rule integrates a polynomial of degree $2M + 1$ exactly. The code `numint.py` contains a large collection of Gauss-Legendre rules.

4.6 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 and use polynomials for approximating a function over a cell. Two popular cell shapes are triangles and quadrilaterals. It is common to denote finite elements on triangles and tetrahedrons as P while elements defined in terms of quadrilaterals and boxes are denoted by Q. Figures 4.23, 4.24, and 4.25 provide examples. P1 elements means linear functions ($a_0 + a_1x + a_2y$) over triangles, while Q1 elements have bilinear functions ($a_0 + a_1x + a_2y + a_3xy$) over rectangular cells. Higher-order elements can easily be defined.

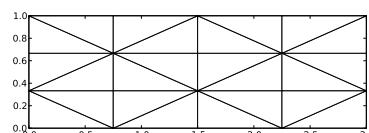
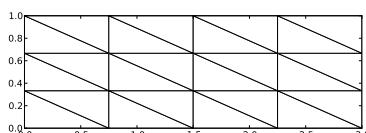


Fig. 4.23 Example on 2D P1 elements.

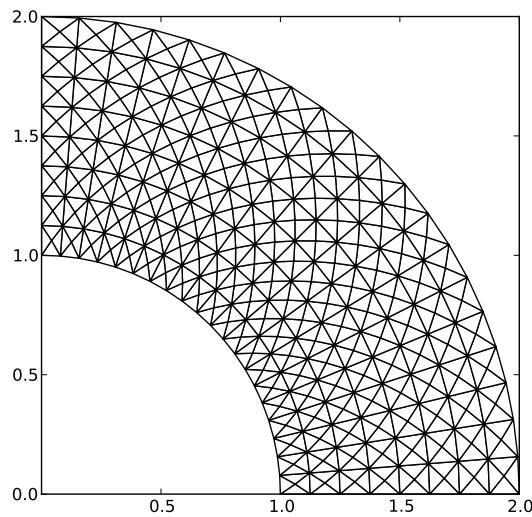


Fig. 4.24 Example on 2D P1 elements in a deformed geometry.

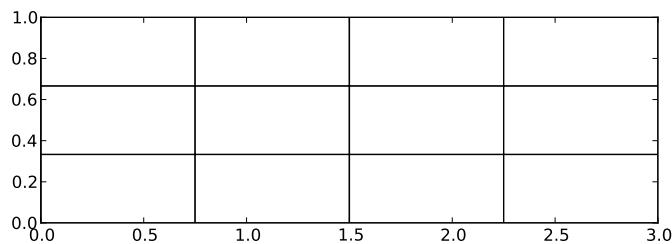


Fig. 4.25 Example on 2D Q1 elements.

4.6.1 Basis functions over triangles in the physical domain

Cells with triangular shape will be in main focus here. With the P1 triangular element, u is a linear function over each cell, as depicted in Figure 4.26, with discontinuous derivatives at the cell boundaries.

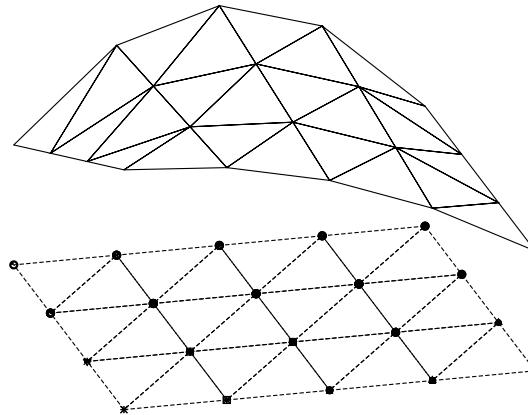


Fig. 4.26 Example on scalar function defined in terms of piecewise linear 2D functions defined on triangles.

We give the vertices of the cells global and local numbers as in 1D. The degrees of freedom in the P1 element are the function values at a set of nodes, which are the three vertices. The basis function $\varphi_i(x, y)$ is then 1 at the vertex with global vertex number i and zero at all other vertices. On an element, the three degrees of freedom uniquely determine the linear basis functions in that element, as usual. The global $\varphi_i(x, y)$ function is then a combination of the linear functions (planar surfaces) over all the neighboring cells that have vertex number i in common. Figure 4.27 tries to illustrate the shape of such a “pyramid”-like function.

Element matrices and vectors. As in 1D, we split the integral over Ω into a sum of integrals over cells. Also as in 1D, φ_i overlaps φ_j (i.e., $\varphi_i \varphi_j \neq 0$) if and only if i and j are vertices in the same cell. Therefore, the integral of $\varphi_i \varphi_j$ over an element is nonzero only when i and j run over the vertex numbers in the element. These nonzero contributions to the coefficient matrix are, as in 1D, collected in an element matrix. The

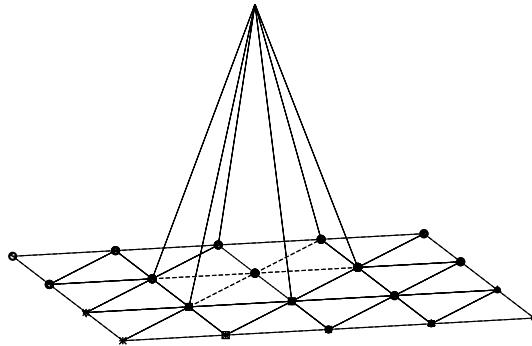


Fig. 4.27 Example on a piecewise linear 2D basis function over a patch of triangles.

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

4.6.2 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 need a mapping of local degrees of freedom numbers to global degrees of freedom numbers.

The reference cell in an (X, Y) coordinate system has vertices $(0, 0)$, $(1, 0)$, and $(0, 1)$, corresponding to local vertex numbers 0, 1, and 2, respectively. The P1 element has linear functions $\tilde{\varphi}_r(X, Y)$ as basis functions, $r = 0, 1, 2$. Since a linear function $\tilde{\varphi}_r(X, Y)$ in 2D is of the form $C_{r,0} + C_{r,1}X + C_{r,2}Y$, and hence has three parameters $C_{r,0}$, $C_{r,1}$, and $C_{r,2}$, we need three degrees of freedom. These are in general taken as the function values at a set of nodes. For the P1 element the set of nodes is the three vertices. Figure 4.28 displays the geometry of the element and the location of the nodes.

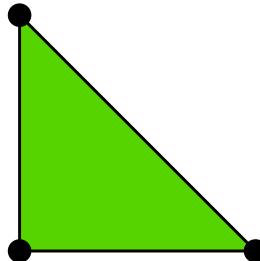


Fig. 4.28 2D P1 element.

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

$$\tilde{\varphi}_0(X, Y) = 1 - X - Y, \quad (4.55)$$

$$\tilde{\varphi}_1(X, Y) = X, \quad (4.56)$$

$$\tilde{\varphi}_2(X, Y) = Y \quad (4.57)$$

Higher-order approximations are obtained by increasing the polynomial order, adding additional nodes, and letting the degrees of freedom be function values at the nodes. Figure 4.29 shows the location of the six nodes in the P2 element.

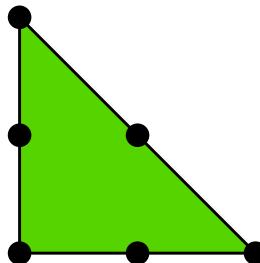


Fig. 4.29 2D P2 element.

A polynomial of degree p in X and Y has $n_p = (p + 1)(p + 2)/2$ terms and hence needs n_p nodes. The values at the nodes constitute n_p degrees of freedom. The location of the nodes for $\tilde{\varphi}_r$ up to degree 6 is displayed in Figure 4.30.

The generalization to 3D is straightforward: the reference element is a **tetrahedron** with vertices $(0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ in a

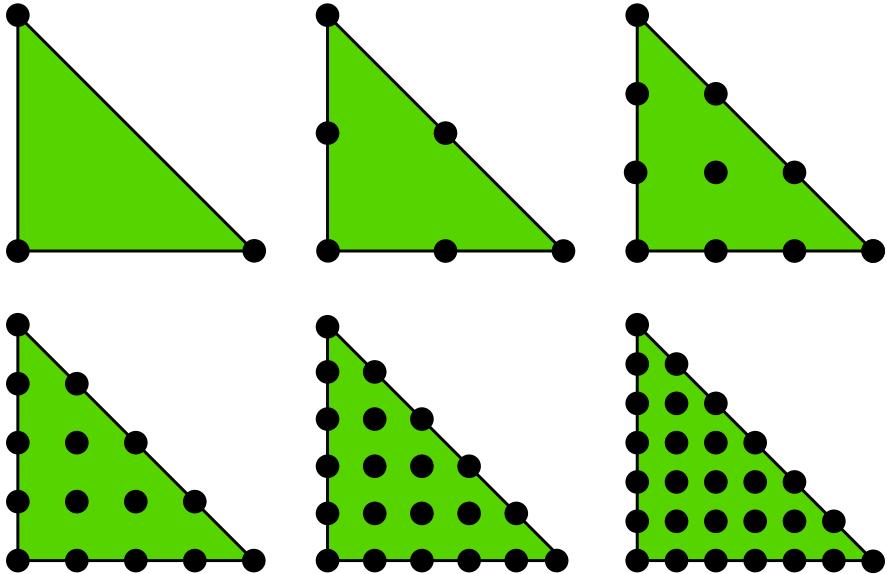


Fig. 4.30 2D P1, P2, P3, P4, P5, and P6 elements.

X, Y, Z reference coordinate system. The P1 element has its degrees of freedom as four nodes, which are the four vertices, see Figure 4.31. The P2 element adds additional nodes along the edges of the cell, yielding a total of 10 nodes and degrees of freedom, see Figure 4.32.

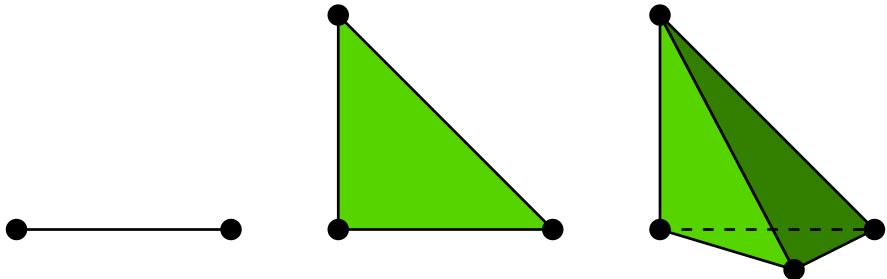


Fig. 4.31 P1 elements in 1D, 2D, and 3D.

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

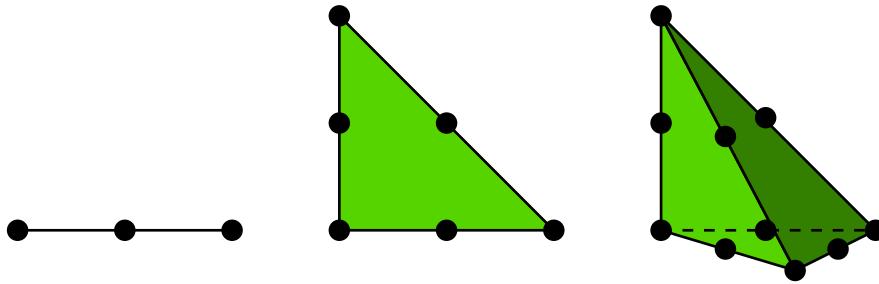


Fig. 4.32 P2 elements in 1D, 2D, and 3D.

Acknowledgment. Figures 4.28-4.32 are created by Anders Logg and taken from the FEniCS book: *Automated Solution of Differential Equations by the Finite Element Method*, edited by A. Logg, K.-A. Mardal, and G. N. Wells, published by Springer, 2012.

4.6.3 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 $\mathbf{x}_{q(e,r)}$ be the physical coordinates of local vertex number r in cell e . Furthermore, let \mathbf{X} be a point in the reference coordinate system corresponding to the point \mathbf{x} in the physical coordinate system. The affine mapping of any \mathbf{X} onto \mathbf{x} is then defined by

$$\mathbf{x} = \sum_r \tilde{\varphi}_r^{(1)}(\mathbf{X}) \mathbf{x}_{q(e,r)}, \quad (4.58)$$

where r runs over the local vertex numbers in the cell. The affine mapping essentially stretches, translates, and rotates the triangle. Straight or planar faces of the reference cell are therefore mapped onto straight or planar faces in the physical coordinate system. The mapping can be used for both P1 and higher-order elements, but note that the mapping itself always applies the P1 basis functions.

4.6.4 Isoparametric mapping of the reference cell

Instead of using the P1 basis functions in the mapping (4.58), we may use the basis functions of the actual P_d element:

$$\mathbf{x} = \sum_r \tilde{\varphi}_r(\mathbf{X}) \mathbf{x}_{q(e,r)}, \quad (4.59)$$

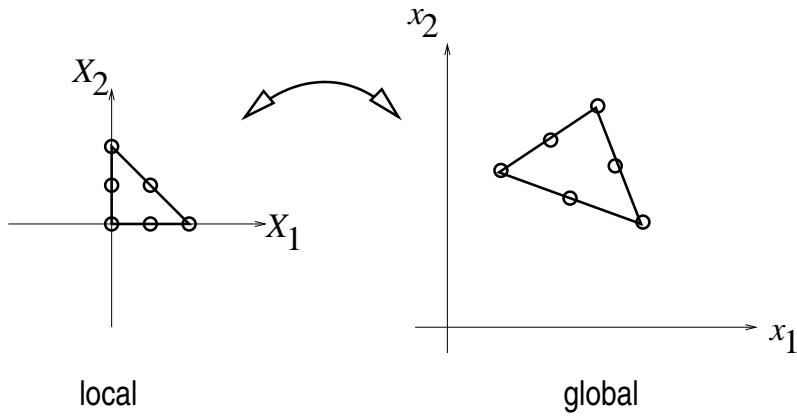


Fig. 4.33 Affine mapping of a P1 element.

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 (4.58), but for higher-order elements the mapping of the straight or planar faces of the reference cell will result in a *curved* face in the physical coordinate system. For example, when we use the basis functions of the triangular P2 element in 2D in (4.59), the straight faces of the reference triangle are mapped onto curved faces of parabolic shape in the physical coordinate system, see Figure 4.34.

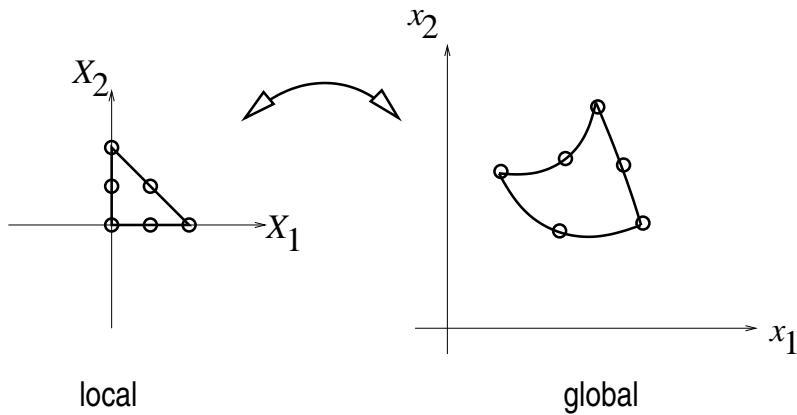


Fig. 4.34 Isoparametric mapping of a P2 element.

From (4.58) or (4.59) it is easy to realize that the vertices are correctly mapped. Consider a vertex with local number 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 $\mathbf{x} = \mathbf{x}_{q(e,s)}$, which is the coordinate of this vertex in the global coordinate system.

4.6.5 Computing integrals

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

$$\int_{\Omega^{(e)}} \varphi_i(\mathbf{x}) \varphi_j(\mathbf{x}) dx = \int_{\tilde{\Omega}^r} \tilde{\varphi}_i(\mathbf{X}) \tilde{\varphi}_j(\mathbf{X}) \det J dX, \quad (4.60)$$

$$\int_{\Omega^{(e)}} \varphi_i(\mathbf{x}) f(\mathbf{x}) dx = \int_{\tilde{\Omega}^r} \tilde{\varphi}_i(\mathbf{X}) f(\mathbf{x}(\mathbf{X})) \det J dX, \quad (4.61)$$

where dx means the infinitesimal area element $dxdy$ in 2D and $dxdydz$ in 3D, with a similar definition of dX . The quantity $\det J$ is the determinant of the Jacobian of the mapping $\mathbf{x}(\mathbf{X})$. In 2D,

$$J = \begin{bmatrix} \frac{\partial x}{\partial X} & \frac{\partial x}{\partial Y} \\ \frac{\partial y}{\partial X} & \frac{\partial y}{\partial Y} \end{bmatrix}, \quad \det J = \frac{\partial x}{\partial X} \frac{\partial y}{\partial Y} - \frac{\partial x}{\partial Y} \frac{\partial y}{\partial X}. \quad (4.62)$$

With the affine mapping (4.58), $\det J = 2\Delta$, where Δ is the area or volume of the cell in the physical coordinate system.

Remark. Observe that finite elements in 2D and 3D build on the same *ideas* and *concepts* as in 1D, but there is simply much more to compute because the specific mathematical formulas in 2D and 3D are more complicated and the book-keeping with dof maps also gets more complicated. The manual work is tedious, lengthy, and error-prone so automation by the computer is a must.

4.7 Implementation

Our previous programs for doing 1D approximation by finite element basis function had a focus on all the small details needed to compute the solution. When going to 2D and 3D, the basic algorithms are the same, but the amount of computational detail with basis functions, reference functions, mappings, numerical integration and so on, becomes overwhelming because of all the flexibility and choices of elements. For

this purpose, we *must*, except in the simplest cases with P1 elements, use some well-developed, existing computer library.

4.7.1 Example of approximation in 2D using FEniCS

Here we shall use **FEniCS**, which is a free, open source finite element package for advanced computations. The package can be programmed in C++ or Python. How it works is best illustrated by an example.

Mathematical problem. We want to approximate the function $f(x) = 2xy - x^2$ by P1 and P2 elements on $[0, 2] \times [-1, 1]$ using a division into 8×8 squares, which are then divided into rectangles and then into triangles.

The code. Observe that the code employs the basic concepts from 1D, but is capable of using *any* element in FEniCS on *any* mesh in *any* number of space dimensions (!).

```
from fenics import *

def approx(f, V):
    """Return Galerkin approximation to f in V."""
    u = TrialFunction(V)
    v = TestFunction(V)
    a = u*v*dx
    L = f*v*dx
    u = Function(V)
    solve(a == L, u)
    return u

def problem():
    f = Expression('2*x[0]*x[1] - pow(x[0], 2)', degree=2)
    mesh = RectangleMesh(Point(0, -1), Point(2, 1), 8, 8)

    V1 = FunctionSpace(mesh, 'P', 1)
    u1 = approx(f, V1)
    u1.rename('u1', 'u1')
    u1_error = errornorm(f, u1, 'L2')
    u1_norm = norm(u1, 'L2')

    V2 = FunctionSpace(mesh, 'P', 2)
    u2 = approx(f, V2)
    u2.rename('u2', 'u2')
    u2_error = errornorm(f, u2, 'L2')
    u2_norm = norm(u2, 'L2')

    print('L2 errors: e1=%g, e2=%g' % (u1_error, u2_error))
    print('L2 norms: n1=%g, n2=%g' % (u1_norm, u2_norm))
    # Simple plotting
```

```

import matplotlib.pyplot as plt
plot(f, title='f', mesh=mesh)
plt.show()
plot(u1, title='u1')
plt.show()
plot(u2, title='u2')
plt.show()

if __name__ == '__main__':
    problem()

```

Figure 4.35 shows the computed u_1 . The plots of u_2 and f are identical and therefore not shown. The plot shows that visually the approximation is quite close to f , but to quantify it more precisely we simply compute the error using the function `errorenorm`. The output of errors becomes

```

L2 errors: e1=0.01314,   e2=4.93418e-15
L2 norms:  n1=4.46217,   n2=4.46219

```

Hence, the second order approximation u_2 is able to reproduce f up to floating point precision, whereas the first order approximation u_1 has an error of slightly more than $\frac{1}{3}\%$.

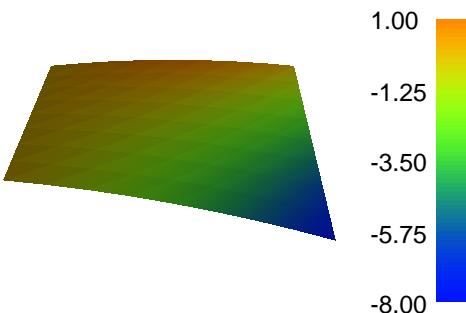


Fig. 4.35 Plot of the computed approximation using Lagrange elements of second order.

Dissection of the code. The function `approx` is a general solver function for any f and V . We define the unknown u in the variational form $a = a(u, v) = \int uv \, dx$ as a `TrialFunction` object and the test function v as a `TestFunction` object. Then we define the variational form through the integrand $u*v*dx$. The linear form L is similarly defined as $f*v*dx$. Here, f is an `Expression` object in FEniCS, i.e., a formula defined in terms of a C++ expression. This expression is in turn jit-compiled into

a Python object for fast evaluation. With `a` and `L` defined, we re-define `u` to be a finite element function `Function`, which is now the unknown scalar field to be computed by the simple expression `solve(a == L, u)`. We remark that the above function `approx` is implemented in FEniCS (in a slightly more general fashion) in the function `project`.

The `problem` function applies `approx` to solve a specific problem.

Integrating SymPy and FEniCS. The definition of f must be expressed in C++. This part requires two definitions: one of f and one of Ω , or more precisely: the mesh (discrete Ω divided into cells). The definition of f is here expressed in C++ (it will be compiled for fast evaluation), where the independent coordinates are given by a C/C++ vector `x`. This means that x is `x[0]`, y is `x[1]`, and z is `x[2]`. Moreover, `x[0]**2` must be written as `pow(x[0], 2)` in C/C++.

Fortunately, we can easily integrate SymPy and `Expression` objects, because SymPy can take a formula and translate it to C/C++ code, and then we can require a Python code to numerically evaluate the formula. Here is how we can specify `f` in SymPy and use it in FEniCS as an `Expression` object:

```
>>> import sympy as sym
>>> x, y = sym.symbols('x[0] x[1]')
>>> f = 2*x*y - x**2
>>> print(f)
-x[0]**2 + 2*x[0]*x[1]
>>> f = sym.printing.ccode(f)    # Translate to C code
>>> print(f)
-pow(x[0], 2) + 2*x[0]*x[1]
>>> import fenics as fe
>>> f = fe.Expression(f, degree=2)
```

Here, the function `ccode` generates C code and we use `x` and `y` as placeholders for `x[0]` and `x[1]`, which represent the coordinate of a general point `x` in any dimension. The output of `ccode` can then be used directly in `Expression`.

4.7.2 Refined code with curve plotting

Interpolation and projection. The operation of defining `a`, `L`, and solving for a `u` is so common that it has been implemented in the FEniCS function `project`:

```
u = project(f, V)
```

So, there is no need for our `approx` function!

If we want to do interpolation (or collocation) instead, we simply do

```
u = interpolate(f, V)
```

Plotting the solution along a line. Having `u` and `f` available as finite element functions (Function objects), we can easily plot the solution along a line since FEniCS has functionality for evaluating a Function at arbitrary points *inside the domain*. For example, here is the code for plotting u and f along a line $x = \text{const}$ or $y = \text{const}$.

```
import numpy as np
import matplotlib.pyplot as plt

def comparison_plot2D(
    u, f,           # Function expressions in x and y
    value=0.5,      # x or y equals this value
    variation='y',  # independent variable
    n=100,          # no of intervals in plot
    tol=1E-8,        # tolerance for points inside the domain
    plottitle='',   # heading in plot
    filename='tmp', # stem of filename
):
    """
    Plot u and f along a line in x or y dir with n intervals
    and a tolerance of tol for points inside the domain.
    """
    v = np.linspace(-1+tol, 1-tol, n+1)
    # Compute points along specified line:
    points = np.array([(value, v_)
                       if variation == 'y' else (v_, value)
                       for v_ in v])
    u_values = [u(point) for point in points] # eval. Function
    f_values = [f(point) for point in points]
    plt.figure()
    plt.plot(v, u_values, 'r-', v, f_values, 'b--')
    plt.legend(['u', 'f'], loc='upper left')
    if variation == 'y':
        plt.xlabel('y'); plt.ylabel('u, f')
    else:
        plt.xlabel('x'); plt.ylabel('u, f')
    plt.title(plottitle)
    plt.savefig(filename + '.pdf')
    plt.savefig(filename + '.png')
```

Integrating plotting and computations. It is now very easy to give some graphical impression of the approximations for various kinds of 2D elements. Basically, to solve the problem of approximating $f = 2xy - x^2$ on $\Omega = [-1, 1] \times [0, 2]$ by P2 elements on a 2×2 mesh, we want to integrate the function above with following type of computations:

```

import fenics as fe
f = fe.Expression('2*x[0]*x[1] - pow(x[0], 2)', degree=2)
mesh = fe.RectangleMesh(fe.Point(1,-1), fe.Point(2,1), 2, 2)
V = fe.FunctionSpace(mesh, 'P', 2)
u = fe.project(f, V)
err = fe.errornorm(f, u, 'L2')
print(err)

```

However, we can now easily compare different type of elements and mesh resolutions:

```

import fenics as fe
import sympy as sym
x, y = sym.symbols('x[0] x[1]')

def problem(f, nx=8, ny=8, degrees=[1,2]):
    """
    Plot u along x=const or y=const for Lagrange elements,
    of given degrees, on a nx times ny mesh. f is a SymPy expression.
    """
    f = sym.printing.ccode(f)
    f = fe.Expression(f, degree=2)
    mesh = fe.RectangleMesh(
        fe.Point(-1, 0), fe.Point(1, 2), 2, 2)
    for degree in degrees:
        if degree == 0:
            # The P0 element is specified like this in FEniCS
            V = fe.FunctionSpace(mesh, 'DG', 0)
        else:
            # The Lagrange Pd family of elements, d=1,2,3,...
            V = fe.FunctionSpace(mesh, 'P', degree)
        u = fe.project(f, V)
        u_error = fe.errornorm(f, u, 'L2')
        print('||u-f||=%g' % u_error, degree)
        comparison_plot2D(
            u, f,
            n=50,
            value=0.4, variation='x',
            plottitle='Approximation by P%d elements' % degree,
            filename='approx_fenics_by_P%d' % degree,
            tol=1E-3)
        #fe.plot(u, title='Approx by P%d' % degree)

    if __name__ == '__main__':
        # x and y are global SymPy variables
        f = 2*x*y - x**16
        f = 2*x*y - x**2
        problem(f, nx=2, ny=2, degrees=[0, 1, 2])
        plt.show()

```

(We note that this code issues a lot of warnings from the `u(point)` evaluations.)

We show in Figure 4.36 how f is approximated by P0, P1, and P2 elements on a very coarse 2×2 mesh consisting of 8 cells.

We have also added the result obtained by P2 elements.

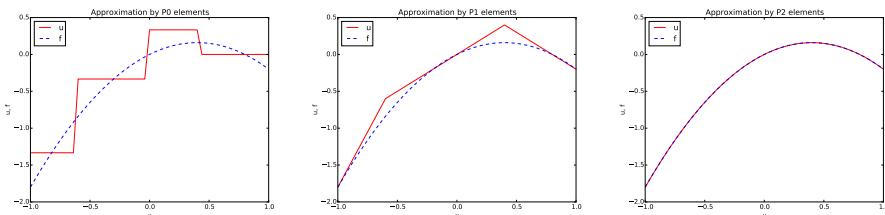


Fig. 4.36 Comparison of P0, P1, and P2 approximations (left to right) along a line in a 2D mesh.

Questions

There are two striking features in the figure:

1. The P2 solution is exact. Why?
2. The P1 solution does not seem to be a least squares approximation. Why?

With this code, found in the file `approx_fenics.py`, we can easily run lots of experiments with the Lagrange element family. Just write the SymPy expression and choose the mesh resolution!

4.8 Exercises

Problem 4.1: Define nodes and elements

Consider a domain $\Omega = [0, 2]$ divided into the three elements $[0, 1]$, $[1, 1.2]$, and $[1.2, 2]$.

For P1 and P2 elements, set up the list of coordinates and nodes (`nodes`) and the numbers of the nodes that belong to each element (`elements`) in two cases: 1) nodes and elements numbered from left to right, and 2) nodes and elements numbered from right to left.

Solution. We can write up figure sketches and the data structure in code:

```

# P1 elements
# Left to right numbering
"""
elements: |--0--|--1--|--2--|
nodes:    0     1     2     3
"""

nodes    = [0, 1, 1.2, 2]
elements = [[0,1], [1,2], [2,3]]


# Right to left numbering
"""
elements: |--2--|--1--|--0--|
nodes:    3     2     1     0
"""

nodes    = [2, 1.2, 1, 0]
elements = [[1,0], [2,1], [3,2]]


# P2 elements

# Left to right numbering
"""
elements: |--0--|--1--|--2--|
nodes:    0     1     2     3     4     5     6
"""

nodes = [0, 0.5, 1, 1.1, 1.6, 2]
elements = [[0,1,2], [2,3,4], [4,5,6]]


# Right to left numbering
"""
elements: |--2--|--1--|--0--|
nodes:    6     5     4     3     2     1     0
"""

nodes = [2, 1.6, 1.2, 1.1, 1, 0.5, 0]
elements = [[2,1,0], [4,3,2], [6,5,4]]

```

Filename: fe_numberings1.

Problem 4.2: Define vertices, cells, and dof maps

Repeat Problem 4.1, but define the data structures `vertices`, `cells`, and `dof_map` instead of `nodes` and `elements`.

Solution. Written in Python, the solution becomes

```
# P1 elements
```

```
# Left to right numbering
"""
elements: |--0---|---1---|---2---|
vertices: 0      1      2      3
dofs:     0      1      2      3
"""
# elements:  0  1  2
# vertices: 0  1  2  3

vertices = [0, 1, 1.2, 2]
cells    = [[0,1], [1,2], [2,3]]
dof_map  = [[0,1], [1,2], [2,3]]


# Right to left numbering
"""
elements: |---2---|---1---|---0---|
vertices: 3      2      1      0
dofs:     3      2      1      0
"""
# elements:  0  1  2
# vertices: 3  2  1  0

vertices = [2, 1.2, 1, 0]
cells    = [[1,0], [2,1], [3,2]]
dof_map  = [[1,0], [2,1], [3,2]]


# P2 elements

# Left to right numbering
# elements:  0  1  2
"""
elements: |---0---|---1---|---2---|
vertices: 0      1      2      3
dofs:     0      1      2      3      4      5      6
"""
# elements:  0  1  2
# vertices: 0  1  2  3
# dof_map:  [0,1,2], [2,3,4], [4,5,6]

vertices = [0, 1, 1.2, 2]
cells    = [[0,1], [1,2], [2,3]]
dof_map  = [[0,1,2], [2,3,4], [4,5,6]]


# Right to left numbering
# elements:  2  1  0
"""
elements: |---2---|---1---|---0---|
vertices: 3      2      1      0
dofs:     6      5      4      3      2      1      0
"""
# elements:  2  1  0
# vertices: 3  2  1  0
# dof_map:  [2,1,0], [4,3,2], [6,5,4]
```

Filename: fe_numberings2.

Problem 4.3: Construct matrix sparsity patterns

Problem 4.1 describes a element mesh with a total of five elements, but with two different element and node orderings. For each of the two orderings, make a 5×5 matrix and fill in the entries that will be nonzero.

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

Solution. If we create an empty matrix, we can run through all elements and then over all local node pairs and mark that the corresponding entry (i, j) in the global matrix is a nonzero entry. The `elements` data structure is sufficient. Below is a program that fills matrix entries with an X and prints the matrix sparsity pattern.

```
def sparsity_pattern(elements, N_n):
    import numpy as np
    matrix = np.zeros((N_n, N_n), dtype=str)
    matrix[:, :] = '0'
    for e in elements:
        for i in e:
            for j in e:
                matrix[i, j] = 'X'
    matrix = matrix.tolist()
    matrix = '\n'.join([' '.join([matrix[i][j]
                                  for j in range(len(matrix[i]))])
                      for i in range(len(matrix))])
    return matrix

print('\nP1 elements, left-to-right numbering')
N_n = 4
elements = [[0,1], [1,2], [2,3]]
print(sparsity_pattern(elements, N_n))

print('\nP1 elements, right-to-left numbering')
elements = [[1,0], [2,1], [3,2]]
print(sparsity_pattern(elements, N_n))

print('\nP2 elements, left-to-right numbering')
N_n = 7
elements = [[0,1,2], [2,3,4], [4,5,6]]
print(sparsity_pattern(elements, N_n))

print('\nP1 elements, right-to-left numbering')
elements = [[2,1,0], [4,3,2], [6,5,4]]
print(sparsity_pattern(elements, N_n))
```

The output becomes

```
P1 elements, left-to-right numbering
X X 0 0
X X X 0
0 X X X
0 0 X X

P1 elements, right-to-left numbering
X X 0 0
X X X 0
0 X X X
0 0 X X

P2 elements, left-to-right numbering
X X X 0 0 0 0
X X X 0 0 0 0
X X X X X 0 0
0 0 X X X 0 0
0 0 X X X X X
0 0 0 0 X X X
0 0 0 0 X X X
```

Filename: `fe_sparsity_pattern.`

Problem 4.4: Perform symbolic finite element computations

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

Solution. Here are suitable `sympy` commands:

```
import sympy as sym
# Mesh: |-----|-----|
#      0      pi/2     pi
#
# Basis functions:
#
#   phi_0   phi_1   phi_2
#   \       / \     /
#   \       / \     /
```

```

#      \ /   \ /
#      \/\   \/
#      |-----|-----|
#      0       pi/2     pi

x = sym.Symbol('x')
A = sym.zeros(3,3)
f = sym.sin

phi_0 = 1 - (2*x)/sym.pi
phi_1l = 2*x/sym.pi      # left part of phi_1
phi_1r = 2 - (2*x)/sym.pi # right part of phi_1
phi_2 = x/(sym.pi/2) - 1
node_0 = 0
node_1 = sym.pi/2
node_2 = sym.pi

# Diagonal terms
A[0,0] = sym.integrate(phi_0**2, (x, node_0, node_1))
A[1,1] = sym.integrate(phi_1l**2, (x, node_0, node_1)) + \
          sym.integrate(phi_1r**2, (x, node_1, node_2))
A[2,2] = sym.integrate(phi_2**2, (x, node_1, node_2))

# Off-diagonal terms
A[0,1] = sym.integrate(phi_0*phi_1l, (x, node_0, node_1))
A[1,0] = A[0,1]

A[1,2] = sym.integrate(phi_1r*phi_2, (x, node_1, node_2))
A[2,1] = A[1,2]

print('A:\n', A) # Can compare with general matrix, h=pi/2

b = sym.zeros(3,1)

b[0] = sym.integrate(phi_0*f(x), (x, node_0, node_1))
b[1] = sym.integrate(phi_1l*f(x), (x, node_0, node_1)) + \
          sym.integrate(phi_1r*f(x), (x, node_1, node_2))
b[2] = sym.integrate(phi_2*f(x), (x, node_1, node_2))

print('b:\n', b)

c = A.LUsolve(b)
print('c:\n', c)

for i in range(len(c)):
    print('c[%d]=%g' % (i, c[i].evalf()))
print('u(pi/2)=%g' % c[1])

# For reports
print(sym.latex(A))
print(sym.latex(b))
print(sym.latex(c))

```

Running the program, we get the matrix system

$$\begin{bmatrix} \frac{\pi}{6} & \frac{\pi}{12} & 0 \\ \frac{\pi}{12} & \frac{\pi}{3} & \frac{\pi}{12} \\ 0 & \frac{\pi}{12} & \frac{\pi}{6} \end{bmatrix} \begin{bmatrix} \frac{1}{\pi} \left(-\frac{24}{\pi} + 8 \right) \\ \frac{-28 + \frac{168}{\pi}}{\pi} \\ \frac{1}{\pi} \left(-\frac{54}{\pi} + 8 \right) \end{bmatrix} = \begin{bmatrix} -\frac{2}{\pi} + 1 \\ \frac{4}{\pi} \\ -\frac{2}{\pi} + 1 \end{bmatrix}$$

The solution at the midpoint is 1.15847, i.e., 16% error.

Filename: `fe_sin_P1`.

Problem 4.5: Approximate a steep function by P1 and P2 elements

Given

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

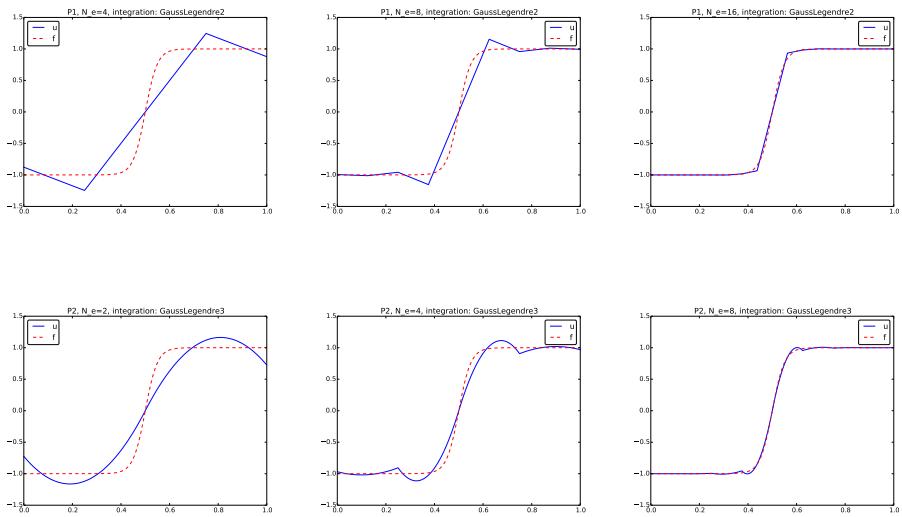
use the Galerkin or least squares method with finite elements to find an approximate function $u(x)$. Choose $s = 20$ and try $N_e = 4, 8, 16$ P1 elements and $N_e = 2, 4, 8$ P2 elements. Integrate $f\varphi_i$ numerically.

Hint. You can automate the computations by calling the `approximate` method in the `fe_approx1D_numint` module.

Solution. The set of calls to `approximate` becomes

```
from fe_approx1D_numint import approximate
from sympy import tanh, Symbol
x = Symbol('x')

steepness = 20
arg = steepness*(x-0.5)
approximate(tanh(arg), symbolic=False, numint='GaussLegendre2',
            d=1, N_e=4, filename='fe_p1_tanh_4e')
approximate(tanh(arg), symbolic=False, numint='GaussLegendre2',
            d=1, N_e=8, filename='fe_p1_tanh_8e')
approximate(tanh(arg), symbolic=False, numint='GaussLegendre2',
            d=1, N_e=16, filename='fe_p1_tanh_16e')
approximate(tanh(arg), symbolic=False, numint='GaussLegendre3',
            d=2, N_e=2, filename='fe_p2_tanh_2e')
approximate(tanh(arg), symbolic=False, numint='GaussLegendre3',
            d=2, N_e=4, filename='fe_p2_tanh_4e')
approximate(tanh(arg), symbolic=False, numint='GaussLegendre3',
            d=2, N_e=8, filename='fe_p2_tanh_8e')
```



Filename: fe_tanh_P1P2.

Problem 4.6: Approximate a steep function by P3 and P4 elements

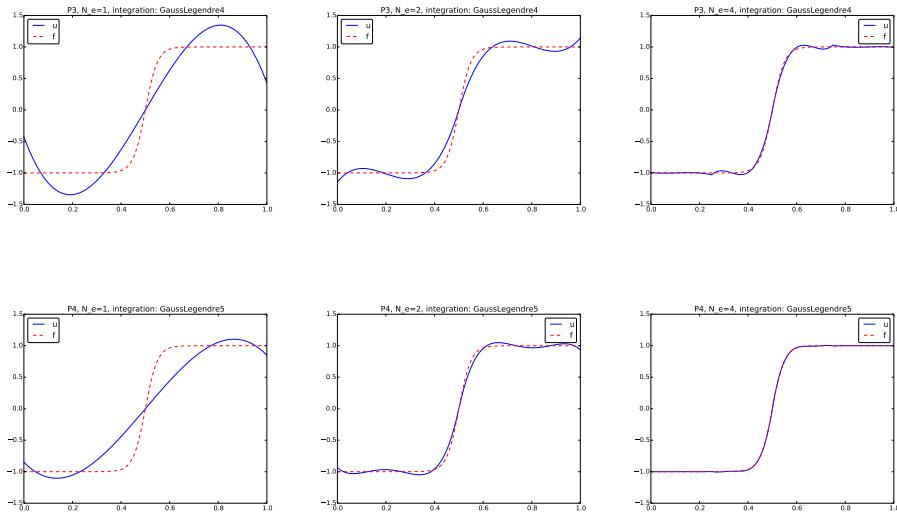
a) Solve Problem 4.5 using $N_e = 1, 2, 4$ P3 and P4 elements.

Solution. We can easily adopt the code from Exercise 4.5:

```
from fe_approx1D_numint import approximate, u_glob
from sympy import tanh, Symbol, lambdify
x = Symbol('x')

steepness = 20
arg = steepness*(x-0.5)

approximate(tanh(arg), symbolic=False, numint='GaussLegendre4',
            d=3, N_e=1, filename='fe_p3_tanh_1e')
approximate(tanh(arg), symbolic=False, numint='GaussLegendre4',
            d=3, N_e=2, filename='fe_p3_tanh_2e')
approximate(tanh(arg), symbolic=False, numint='GaussLegendre4',
            d=3, N_e=4, filename='fe_p3_tanh_4e')
approximate(tanh(arg), symbolic=False, numint='GaussLegendre5',
            d=4, N_e=1, filename='fe_p4_tanh_1e')
approximate(tanh(arg), symbolic=False, numint='GaussLegendre5',
            d=4, N_e=2, filename='fe_p4_tanh_2e')
approximate(tanh(arg), symbolic=False, numint='GaussLegendre5',
            d=4, N_e=4, filename='fe_p4_tanh_4e')
```



b) How will an interpolation method work in this case with the same number of nodes?

Solution. The coefficients arising from the interpolation method are trivial to compute since $c_i = f(x_i)$, where x_i are the global nodes. The function `u_glob` in the `fe_approx1D_numint` module can be used to compute appropriate arrays for plotting the resulting finite element function. We create plots where the finite element approximation is shown along with $f(x)$ and the interpolation points. Since `u_glob` requires the `vertices`, `cells`, and `dof_map` data structures, we must compute these for the values of number of elements (N_e) and the polynomial degree (d).

```
# Interpolation method
import numpy as np
import matplotlib.pyplot as plt
f = lambdify([x], tanh(arg), modules='numpy')

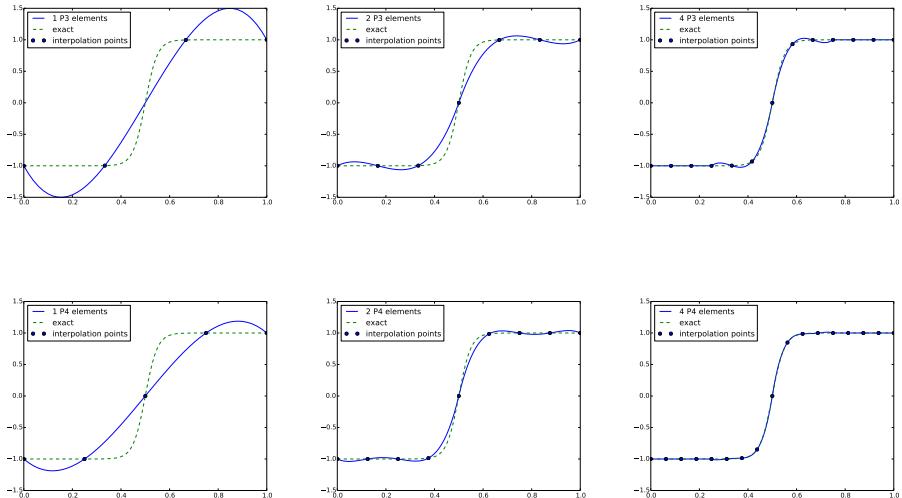
# Compute exact f on a fine mesh
x_fine = np.linspace(0, 1, 101)
f_fine = f(x_fine)

for d in 3, 4:
    for N_e in 1, 2, 4:
        h = 1.0/N_e # element length
        vertices = [i*h for i in range(N_e+1)]
        cells = [[e, e+1] for e in range(N_e)]
        dof_map = [[i*e + j for i in range(d+1)] for e in range(N_e)]
        N_n = d*N_e + 1 # Number of nodes
        x_nodes = np.linspace(0, 1, N_n) # Node coordinates
        U = f(x_nodes) # Interpolation method samples node values
```

```

x, u, _ = u_glob(U, vertices, cells, dof_map,
                   resolution_per_element=51)
plt.figure()
plt.plot(x, u, '-.', x_fine, f_fine, '--',
          x_nodes, U, 'bo')
plt.legend(['%d P%d elements' % (N_e, d),
            'exact', 'interpolation points'],
           loc='upper left')
plt.savefig('tmp_%d_P%d.pdf' % (N_e, d))
plt.savefig('tmp_%d_P%d.png' % (N_e, d))
plt.show()

```



Filename: fe_tanh_P3P4.

Exercise 4.7: Investigate the approximation error in finite elements

The theory (4.42) from Section 4.4.4 predicts that the error in the P^d approximation of a function should behave as h^{d+1} , where h is the length of the element. Use experiments to verify this asymptotic behavior (i.e., for small enough h). Choose three examples: $f(x) = Ae^{-\omega x}$ on $[0, 3/\omega]$, $f(x) = A \sin(\omega x)$ on $\Omega = [0, 2\pi/\omega]$ for constant A and ω , and $f(x) = \sqrt{x}$ on $[0, 1]$.

Hint 1. Run a series of experiments: (h_i, E_i) , $i = 0, \dots, m$, where E_i is the L^2 norm of the error corresponding to element length h_i . Assume an error model $E = Ch^r$ and compute r from two successive experiments:

$$r_i = \ln(E_{i+1}/E_i) / \ln(h_{i+1}/h_i), \quad i = 0, \dots, m-1.$$

Hopefully, the sequence r_0, \dots, r_{m-1} converges to the true r , and r_{m-1} can be taken as an approximation to r . Run such experiments for different d for the different $f(x)$ functions.

Hint 2. The `approximate` function in `fe_approx1D_numint.py` is handy for calculating the numerical solution. This function returns the finite element solution as the coefficients $\{c_i\}_{i \in \mathcal{I}_s}$. To compute u , use `u_glob` from the same module. Use the Trapezoidal rule to integrate the L^2 error:

```
xc, u = u_glob(c, vertices, cells, dof_map)
e = f_func(xc) - u
L2_error = 0
e2 = e**2
for i in range(len(xc)-1):
    L2_error += 0.5*(e2[i+1] + e2[i])*(xc[i+1] - xc[i])
L2_error = np.sqrt(L2_error)
```

The reason for this Trapezoidal integration is that `u_glob` returns coordinates `xc` and corresponding `u` values where some of the coordinates (the cell vertices) coincides, because the solution is computed in one element at a time, using all local nodes. Also note that there are many coordinates in `xc` per cell such that we can accurately compute the error inside each cell.

Solution. Here is an appropriate program:

```
from fe_approx1D_numint import approximate, mesh_uniform, u_glob
from sympy import sqrt, exp, sin, Symbol, lambdify, simplify
import numpy as np
from math import log

x = Symbol('x')
A = 1
w = 1

cases = {'sqrt': {'f': sqrt(x), 'Omega': [0,1]}, 
          'exp': {'f': A*exp(-w*x), 'Omega': [0, 3.0/w]}, 
          'sin': {'f': A*sin(w*x), 'Omega': [0, 2*np.pi/w]}}

results = []
d_values = [1, 2, 3, 4]

for case in cases:
    f = cases[case]['f']
    f_func = lambdify([x], f, modules='numpy')
    Omega = cases[case]['Omega']
```

```

results[case] = {}
for d in d_values:
    results[case][d] = {'E': [], 'h': [], 'r': []}
    for N_e in [4, 8, 16, 32, 64, 128]:
        try:
            c = approximate(
                f, symbolic=False,
                numint='GaussLegendre%d' % (d+1),
                d=d, N_e=N_e, Omega=Omega,
                filename='tmp_%s_d%d_e%d' % (case, d, N_e))
        except np.linalg.LinAlgError as e:
            print(str(e))
            continue
        vertices, cells, dof_map = mesh_uniform(
            N_e, d, Omega, symbolic=False)
        xc, u, _ = u_glob(c, vertices, cells, dof_map, 51)
        e = f_func(xc) - u
        # Trapezoidal integration of the L2 error over the
        # xc/u patches
        e2 = e**2
        L2_error = 0
        for i in range(len(xc)-1):
            L2_error += 0.5*(e2[i+1] + e2[i])*(xc[i+1] - xc[i])
        L2_error = np.sqrt(L2_error)
        h = (Omega[1] - Omega[0])/float(N_e)
        results[case][d]['E'].append(L2_error)
        results[case][d]['h'].append(h)

        # Compute rates
        h = results[case][d]['h']
        E = results[case][d]['E']
        for i in range(len(h)-1):
            r = log(E[i+1]/E[i])/log(h[i+1]/h[i])
            results[case][d]['r'].append(round(r, 2))

print(results)
for case in results:
    for d in sorted(results[case]):
        print('case=%s d=%d, r: %s' % \
              (case, d, results[case][d]['r']))

```

The output becomes

```

case=sqrt d=1, r: [1.0, 1.0, 1.0, 1.0, 1.0]
case=sqrt d=2, r: [1.0, 1.0, 1.0, 1.0, 1.0]
case=sqrt d=3, r: [1.0, 1.0, 1.0, 1.0, 1.0]
case=sqrt d=4, r: [1.0, 1.0, 1.0, 1.0, 1.0]
case=exp d=1, r: [2.01, 2.01, 2.0, 2.0, 2.0]
case=exp d=2, r: [2.81, 2.89, 2.94, 2.97, 2.98]
case=exp d=3, r: [3.98, 4.0, 4.0, 4.0, 4.0]
case=exp d=4, r: [4.87, 4.93, 4.96, 4.98, 4.99]
case=sin d=1, r: [2.15, 2.06, 2.02, 2.0, 2.0]
case=sin d=2, r: [2.68, 2.83, 2.93, 2.97, 2.99]
case=sin d=3, r: [4.06, 4.04, 4.01, 4.0, 4.0]
case=sin d=4, r: [4.79, 4.9, 4.96, 4.98, 4.99]

```

showing that the convergence rate stabilizes quite quickly at $N_e = 128$ cells. While the theory predicts the rate as $d + 1$, this is only fulfilled for the exponential and sine functions, while the square root functions gives a rate 1 regardless of d . The reason is that the estimate (4.42) contains the integral of the derivatives of f over $[0, 1]$. For $f = \sqrt{x}$, we have $f' = \frac{1}{2}x^{-1/2}$, $f'' = -\frac{1}{4}x^{-3/2}$, and all integrals of f'' and higher derivatives are infinite on $[0, L]$. Our experiments show that the method still converges, but f is not smooth enough that higher-order elements give superior convergence rates.

Filename: `Pd_approx_error`.

Problem 4.8: Approximate a step function by finite elements

Approximate the step function

$$f(x) = \begin{cases} 0 & \text{if } 0 \leq x < 1/2, \\ 1 & \text{if } 1/2 \leq x \geq 1/2 \end{cases}$$

by 2, 4, 8, and 16 elements and P1, P2, P3, and P4. Compare approximations visually.

Hint. This f can also be expressed in terms of the Heaviside function $H(x)$: $f(x) = H(x - 1/2)$. Therefore, f can be defined by

```
f = sym.Heaviside(x - sym.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`.

An alternative is to perform hand calculations. This is an instructive task, but in practice only feasible for few elements and P1 and P2 elements. It is better to copy the functions `element_matrix`, `element_vector`, `assemble`, and `approximate` from the `fe_approx1D_numint.py` file and edit these functions such that they can compute approximations with `f` given as a Python function and not a symbolic expression. Also assume that `phi` computed by the `basis` function is a Python callable function. Remove all instances of the `symbolic` variable and associated code.

Solution. The modifications of `element_matrix`, `element_vector`, `assemble`, and `approximate` from the `fe_approx1D_numint.py` file are listed below.

```

from fe_approx1D_numint import mesh_uniform, u_glob
from fe_approx1D import basis
import numpy as np

def element_matrix(phi, Omega_e, numint):
    n = len(phi)
    A_e = np.zeros((n, n))
    h = Omega_e[1] - Omega_e[0]
    detJ = h/2 # dx/dX
    for r in range(n):
        for s in range(r, n):
            for j in range(len(numint[0])):
                Xj, wj = numint[0][j], numint[1][j]
                A_e[r,s] += phi[r](Xj)*phi[s](Xj)*detJ*wj
    A_e[s,r] = A_e[r,s]
    return A_e

def element_vector(f, phi, Omega_e, numint):
    n = len(phi)
    b_e = np.zeros(n)
    h = Omega_e[1] - Omega_e[0]
    detJ = h/2
    for r in range(n):
        for j in range(len(numint[0])):
            Xj, wj = numint[0][j], numint[1][j]
            xj = (Omega_e[0] + Omega_e[1])/2 + h/2*Xj # mapping
            b_e[r] += f(xj)*phi[r](Xj)*detJ*wj
    return b_e

def assemble(vertices, cells, dof_map, phi, f, numint):
    N_n = len(list(set(np.array(dof_map).ravel())))
    N_e = len(cells)
    A = np.zeros((N_n, N_n))
    b = np.zeros(N_n)
    for e in range(N_e):
        Omega_e = [vertices[cells[e][0]], vertices[cells[e][1]]]
        A_e = element_matrix(phi[e], Omega_e, numint)
        b_e = element_vector(f, phi[e], Omega_e, numint)
        #print('element', e)
        #print(b_e)
        for r in range(len(dof_map[e])):
            for s in range(len(dof_map[e])):
                A[dof_map[e][r],dof_map[e][s]] += A_e[r,s]
            b[dof_map[e][r]] += b_e[r]
    return A, b

def approximate(f, d, N_e, numint, Omega=[0,1], filename='tmp'):
    """
    Compute the finite element approximation, using Lagrange
    elements of degree d, to a Python function f on a domain
    Omega. N_e is the number of elements.
    numint is the name of the numerical integration rule
    (Trapezoidal, Simpson, GaussLegendre2, GaussLegendre3,

```

```
GaussLegendre4, etc.). numint=None implies exact
integration.
"""
from math import sqrt
numint_name = numint # save name
if numint == 'Trapezoidal':
    numint = [[-1, 1], [1, 1]]
elif numint == 'Simpson':
    numint = [[-1, 0, 1], [1./3, 4./3, 1./3]]
elif numint == 'Midpoint':
    numint = [[0], [2]]
elif numint == 'GaussLegendre2':
    numint = [[-1/sqrt(3), 1/sqrt(3)], [1, 1]]
elif numint == 'GaussLegendre3':
    numint = [[-sqrt(3./5), 0, sqrt(3./5)],
               [5./9, 8./9, 5./9]]
elif numint == 'GaussLegendre4':
    numint = [[-0.86113631, -0.33998104, 0.33998104,
               0.86113631],
               [ 0.34785485, 0.65214515, 0.65214515,
                 0.34785485]]
elif numint == 'GaussLegendre5':
    numint = [[-0.90617985, -0.53846931, -0. ,
               0.53846931, 0.90617985],
               [ 0.23692689, 0.47862867, 0.56888889,
                 0.47862867, 0.23692689]]
elif numint is not None:
    print('Numerical rule %s is not supported '\
          'for numerical computing' % numint)
    sys.exit(1)

vertices, cells, dof_map = mesh_uniform(N_e, d, Omega)

# phi is a list where phi[e] holds the basis in cell no e
# (this is required by assemble, which can work with
# meshes with different types of elements).
# len(dof_map[e]) is the number of nodes in cell e,
# and the degree of the polynomial is len(dof_map[e])-1
phi = [basis(len(dof_map[e])-1) for e in range(N_e)]

A, b = assemble(vertices, cells, dof_map, phi, f,
                 numint=numint)

print('cells:', cells)
print('vertices:', vertices)
print('dof_map:', dof_map)
print('A:\n', A)
print('b:\n', b)
c = np.linalg.solve(A, b)
print('c:\n', c)

if filename is not None:
    title = 'P%d, N_e=%d' % (d, N_e)
```

```

title += ', integration: %s' % numint_name
x_u, u, _ = u_glob(np.asarray(c), vertices, cells, dof_map,
                     resolution_per_element=51)
x_f = np.linspace(Omega[0], Omega[1], 10001) # mesh for f
import scitools.std as plt
plt.plot(x_u, u, '--',
          x_f, f(x_f), '---')
plt.legend(['u', 'f'])
plt.title(title)
plt.savefig(filename + '.pdf')
plt.savefig(filename + '.png')
return c

```

With a purely numerical version of the `approximate` function, we can easily investigate the suggested approximations in this exercise:

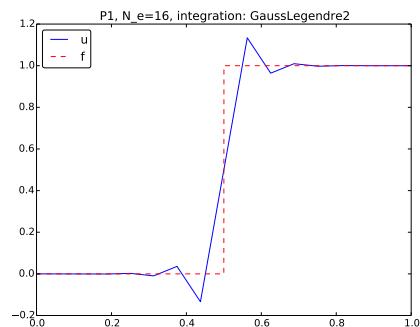
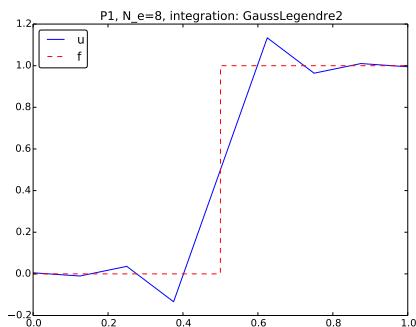
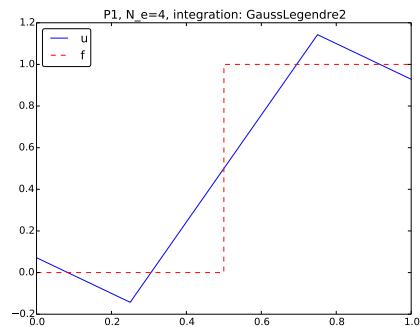
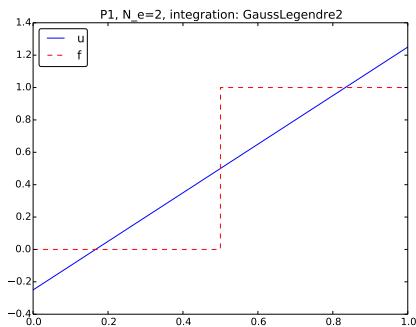
```

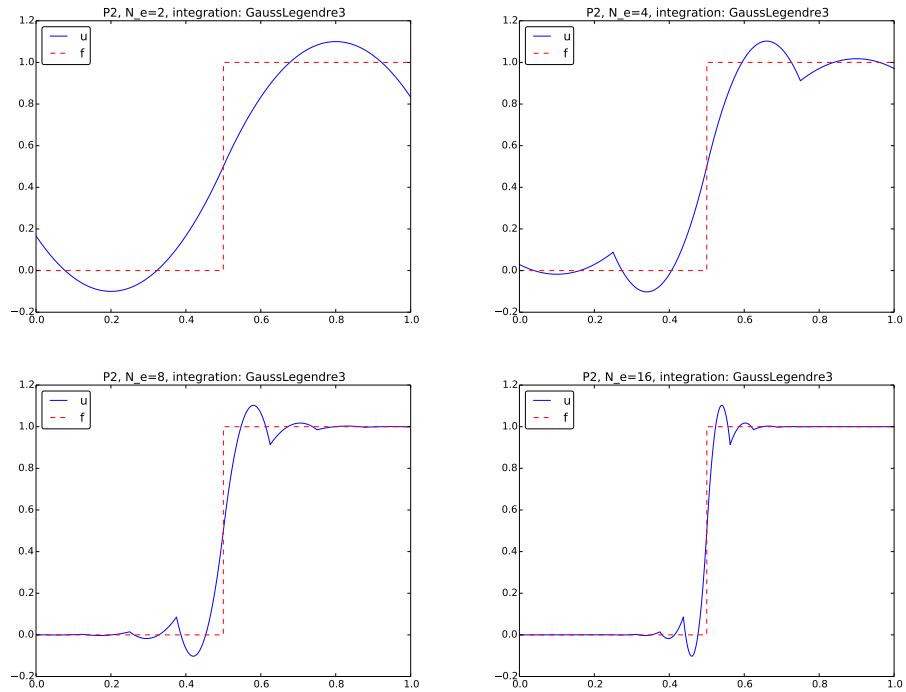
def exercise():
    def f(x):
        if isinstance(x, (float,int)):
            return 0 if x < 0.5 else 1
        elif isinstance(x, np.ndarray):
            return np.where(x < 0.5, 0, 1)

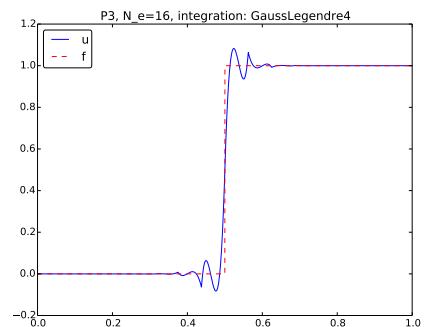
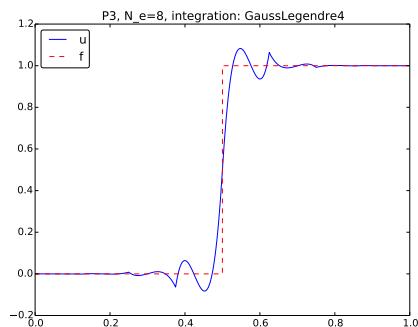
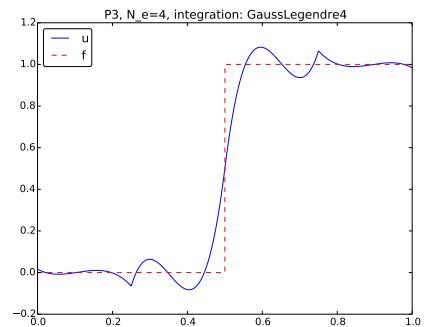
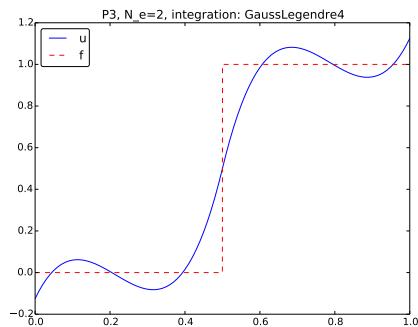
    N_e_values = [2, 4, 8, 16]
    for d in 1, 2, 3, 4:
        for N_e in N_e_values:
            approximate(f, numint='GaussLegendre%d' % (d+1),
                        d=d, N_e=N_e,
                        filename='fe_Heaviside_P%d_%de' % (d, N_e))
    for ext in 'pdf', 'png':
        cmd = 'doconce combine_images '
        cmd += ext + ' -2 '
        cmd += ' '.join(['fe_Heaviside_P%d_%de' % (d, N_e)
                         for N_e in N_e_values])
        cmd += ' fe_Heaviside_P%d' % d
        print(cmd)
        os.system(cmd)

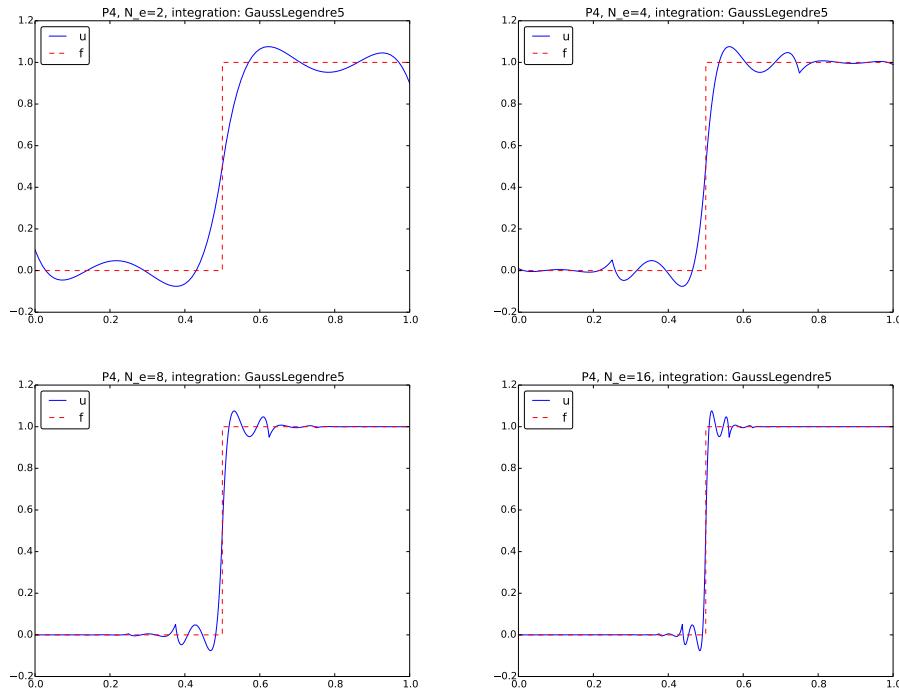
```

Running this function reveals that even finite elements (and not only sines, as demonstrated in Exercise 3.8) give oscillations around a discontinuity.









Remarks. It is of extreme importance to use a Gauss-Legendre numerical integration rule that matches the degree of polynomials in the basis. Using a rule with fewer points may lead to very strange results.

Filename: `fe_Heaviside_P1P2`.

Exercise 4.9: 2D approximation with orthogonal functions

a) 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.

Solution. We 1) remove the `j` loop in the `least_squares` function and set `j = i`, 2) make `A` a vector (i.e., $(N + 1, 1)$ matrix as `b` and `c`), 3) solve for `c[i, 0]` as soon as `A[i, 0]` and `b[i, 0]` are computed.

```
import sympy as sym
import mpmath
```

```
def least_squares_orth(f, psi, Omega, symbolic=True,
                      print_latex=False):
    """
    Given a function f(x,y) on a rectangular domain
    Omega=[[xmin,xmax],[ymin,ymax]],
    return the best approximation to f(x,y) in the space V
    spanned by the functions in the list psi.
    This function assumes that psi are orthogonal on Omega.
    """
    # Modification of least_squares function: drop the j loop,
    # set j=i, compute c on the fly in the i loop.

    N = len(psi) - 1
    # Note that A, b, c becomes (N+1)x(N+1), use 1st column
    A = sym.zeros(N+1)
    b = sym.zeros(N+1)
    c = sym.zeros(N+1)
    x, y = sym.symbols('x y')
    print('...evaluating matrix...', A.shape, b.shape, c.shape)
    for i in range(N+1):
        j = i
        print('(%d,%d)' % (i, j))

        integrand = psi[i]*psi[j]
        if symbolic:
            I = sym.integrate(integrand,
                               (x, Omega[0][0], Omega[0][1]),
                               (y, Omega[1][0], Omega[1][1]))
        if not symbolic or isinstance(I, sym.Integral):
            # Could not integrate symbolically, use numerical int.
            print('numerical integration of', integrand)
            integrand = sym.lambdify([x,y], integrand, 'mpmath')
            I = mpmath.quad(integrand,
                             [Omega[0][0], Omega[0][1]],
                             [Omega[1][0], Omega[1][1]])
        A[i,0] = I

        integrand = psi[i]*f
        if symbolic:
            I = sym.integrate(integrand,
                               (x, Omega[0][0], Omega[0][1]),
                               (y, Omega[1][0], Omega[1][1]))
        if not symbolic or isinstance(I, sym.Integral):
            # Could not integrate symbolically, use numerical int.
            print('numerical integration of', integrand)
            integrand = sym.lambdify([x,y], integrand, 'mpmath')
            I = mpmath.quad(integrand,
                             [Omega[0][0], Omega[0][1]],
                             [Omega[1][0], Omega[1][1]])
        b[i,0] = I
        c[i,0] = b[i,0]/A[i,0]
    print()
    print('A:\n', A, '\nb:\n', b)
```

```

c = [c[i,0] for i in range(c.shape[0])] # make list
print('coeff:', c)

# c is a sympy Matrix object, numbers are in c[i,0]
u = sum(c[i]*psi[i] for i in range(len(psi)))
print('approximation:', u)
print('f:', sym.expand(f))
if print_latex:
    print(sym.latex(A, mode='plain'))
    print(sym.latex(b, mode='plain'))
    print(sym.latex(c, mode='plain'))
return u, c

```

b) 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

$$\varphi_i(x, y) = \sin((p+1)\pi x) \sin((q+1)\pi y), \quad i = q(N_x + 1) + p,$$

where $p = 0, \dots, N_x$ and $q = 0, \dots, N_y$.

Hint. Get ideas from the function `least_squares_orth` in Section 3.3.3 and file `approx1D.py`.

Solution. A function for computing the basis functions may look like this:

```

def sine_basis(Nx, Ny):
    """
    Compute basis sin((p+1)*pi*x)*sin((q+1)*pi*y),
    p=0,...,Nx, q=0,...,Ny.
    """
    x, y = sym.symbols('x y')
    psi = []
    for q in range(0, Ny+1):
        for p in range(0, Nx+1):
            r = sym.sin((p+1)*sym.pi*x)*sym.sin((q+1)*sym.pi*y)
            psi.append(r)
    return psi

```

Application of this basis to approximate the given function is coded in the following function:

```

def demo(N):
    """
    Find the approximation of f by the least squares method.
    The basis is sin((p+1)*pi*x)sin((q+1)*pi*y) where
    0<p<=N, p<q<=N.

```

```

"""
x, y = sym.symbols('x y')
f = x*(1-x)*y*(1-y)*sym.exp(-x-y)

psi = sine_basis(N, N)

Omega = [[0,1], [0,1]]
u, c = least_squares_orth(f, psi, Omega, symbolic=False)
from approx2D import comparison_plot
comparison_plot(f, u, Omega, title='N=%d' % N)
print(c)

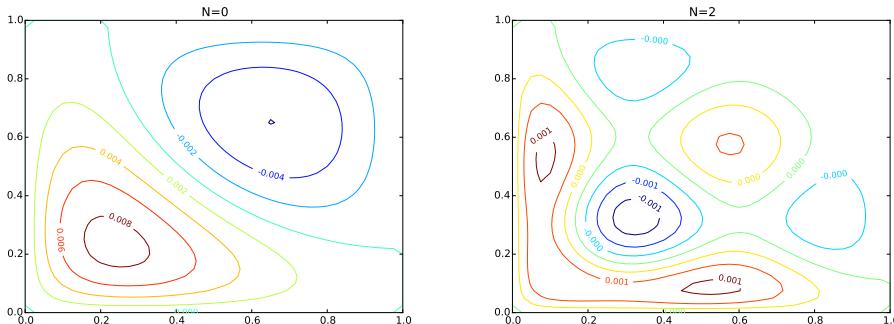
if __name__ == '__main__':
    #test_least_squares_orth()
    demo(N=2)

```

A lesson learned is that `symbolic=False` is important, otherwise `sympy` consumes a lot of CPU time on trying to integrate symbolically.

The figure below shows the error in the approximation for $N = 0$ (left) and $N = 2$ (right). The coefficients for $N = 2$ decay rapidly:

```
[0.025, 0.0047, 0.0014, 0.0047, 0.0009, 0.0003, 0.0014, 0.0003,
 8.2e-5]
```



c) Make a unit test for the `least_squares_orth` function.

Solution. Let us use the basis in b), fix the coefficients of some function f , and check that the computed approximation, with the same basis, has the same coefficients (this test employs the principle that if $f \in V$, then $u = f$).

```

def test_least_squares_orth():
    # Use sine functions
    x, y = sym.symbols('x y')
    N = 2 # (N+1)**2 = 9 basis functions
    psi = sine_basis(N, N)

```

```

f_coeff = [0]*len(psi)
f_coeff[3] = 2
f_coeff[4] = 3
f = sum(f_coeff[i]*psi[i] for i in range(len(psi)))
# Check that u exactly reproduces f
u, c = least_squares_orth(f, psi, Omega=[[0,1], [0,1]],
                           symbolic=False)
import numpy as np
diff = np.abs(np.array(c) - np.array(f_coeff)).max()
print('diff:', diff)
tol = 1E-15
assert diff < tol

```

Filename: approx2D_ls_orth.

Exercise 4.10: Use the Trapezoidal rule and P1 elements

Consider the approximation of some $f(x)$ on an interval Ω using the least squares or Galerkin methods with P1 elements. Derive the element matrix and vector using the Trapezoidal rule (4.50) for calculating integrals on the reference element. Assemble the contributions, assuming a uniform cell partitioning, and show that the resulting linear system has the form $c_i = f(x_i)$ for $i \in \mathcal{I}_s$.

Solution. The Trapezoidal rule for integrals on $[-1, 1]$ is given by (4.50). The expressions for the entries in the element matrix are given by (4.15) in Section 4.1.8:

$$\begin{aligned}\tilde{A}_{r,s}^{(e)} &= \int_{-1}^1 \tilde{\varphi}_r(X) \tilde{\varphi}_s(X) \det J \, dX \\ &\approx \frac{h}{2} (\tilde{\varphi}_r(-1)\tilde{\varphi}_s(-1) + \tilde{\varphi}_r(1)\tilde{\varphi}_s(1)) .\end{aligned}$$

We know that if $\tilde{\varphi}_r(\pm 1)$ is 0 or 1, so evaluating the formula above for $r, s = 0, 1$ gives

$$\tilde{A}^{(e)} = \frac{h}{2} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} .$$

As usual, h is the length of the element in physical coordinates.

The element vector in the reference element is given by (4.16):

$$\begin{aligned}\tilde{b}_r^{(e)} &= \int_{-1}^1 f(x(X))\tilde{\varphi}_r(X) \det J \, dX \\ &\approx \frac{h}{2}(f(x(-1))\tilde{\varphi}_r(-1) + f(x(1))\tilde{\varphi}_r(1)).\end{aligned}$$

Evaluating the formula for $r = 0, 1$ leads to

$$\tilde{b}^{(e)} = \frac{h}{2} \begin{pmatrix} f(x_L) \\ f(x_R) \end{pmatrix},$$

where x_L and x_R are the x coordinates of the local points $X = -1$ and $X = 1$, respectively.

With a uniform mesh with nodes $x_i = ih$, the element matrix and vectors assemble to a coefficient matrix

$$\frac{h}{2} \text{diag}(1, 2, \dots, 2, 1),$$

and right-hand side vector

$$\frac{h}{2}(f(x_0), 2f(x_1), \dots, 2f(x_{N_n-1}), f(x_{N_n})).$$

The factors $h/2$ and 2 cancel, so we are left with the solution of the system as

$$c_i = f(x_i).$$

Filename: `fe_P1_trapez.`

Exercise 4.11: Compare P1 elements and interpolation

We shall approximate the function

$$f(x) = 1 + \epsilon \sin(2\pi nx), \quad x \in \Omega = [0, 1],$$

where $n \in \mathbb{Z}$ and $\epsilon \geq 0$.

- a)** Plot $f(x)$ for $n = 1, 2, 3$ and find the wavelength of the function.
- b)** We want to use N_P elements per wavelength. Show that the number of elements is then nN_P .
- c)** The critical quantity for accuracy is the number of elements per wavelength, not the element size in itself. It therefore suffices to study an f with just one wavelength in $\Omega = [0, 1]$. Set $\epsilon = 0.5$.

Run the least squares or projection/Galerkin method for $N_P = 2, 4, 8, 16, 32$. Compute the error $E = \|u - f\|_{L^2}$.

Hint 1. Use the `fe_approx1D_numint` module to compute u and use the technique from Section 4.4.4 to compute the norm of the error.

Hint 2. Read up on the Nyquist–Shannon sampling theorem.

d) Repeat the set of experiments in the above point, but use interpolation/collocation based on the node points to compute $u(x)$ (recall that c_i is now simply $f(x_i)$). Compute the error $E = \|u - f\|_{L^2}$. Which method seems to be most accurate?

Filename: `fe_P1_vs_interp`.

Exercise 4.12: Implement 3D computations with global basis functions

Extend the `approx2D.py` code to 3D by applying ideas from Section 3.6.4. Construct some 3D problem to make a test function for the implementation.

Hint. Drop symbolic integration since it is in general too slow for 3D problems. Also use `scipy.integrate.nquad` instead of `mpmath.quad` for numerical integration, since it is much faster.

Solution. We take a copy of `approx2D.py` and drop the `comparison_plot` function since plotting in 3D is much more complicated (could make a special version with curves through lines in the 3D domain, for instance). Furthermore, we remove the lines with symbolic integration and replace the calls to `mpmath.quad` by calls to `scipy.integrate.nquad`. The resulting function becomes

```
import sympy as sym
import numpy as np
import scipy.integrate

def least_squares(f, psi, Omega):
    """
    Given a function f(x,y,z) on a rectangular domain
    Omega=[[xmin,xmax],[ymin,ymax],[zmin,zmax]],
    return the best approximation to f in the space V
    spanned by the functions in the list psi.
    f and psi are symbolic (sympy) expressions, but will
    be converted to numeric functions for faster integration.
    """
    N = len(psi) - 1
```

```

A = np.zeros((N+1, N+1))
b = np.zeros(N+1)
x, y, z = sym.symbols('x y z')
f = sym.lambdify([x, y, z], f, modules='numpy')
psi_sym = psi[:] # take a copy, needed for forming u later
psi = [sym.lambdify([x, y, z], psi[i]) for i in range(len(psi))]

print('...evaluating matrix...')
for i in range(N+1):
    for j in range(i, N+1):
        print('(%d,%d)' % (i, j))

        integrand = lambda x, y, z: psi[i](x,y,z)*psi[j](x,y,z)
        I, err = scipy.integrate.nquad(
            integrand,
            [[Omega[0][0], Omega[0][1]],
             [Omega[1][0], Omega[1][1]],
             [Omega[2][0], Omega[2][1]]])
        A[i,j] = A[j,i] = I
    integrand = lambda x, y, z: psi[i](x,y,z)*f(x,y,z)
    I, err = scipy.integrate.nquad(
        integrand,
        [[Omega[0][0], Omega[0][1]],
         [Omega[1][0], Omega[1][1]],
         [Omega[2][0], Omega[2][1]]])
    b[i] = I
print()
c = np.linalg.solve(A, b)
if N <= 10:
    print('A:\n', A, '\nb:\n', b)
    print('coeff:', c)
u = sum(c[i]*psi_sym[i] for i in range(len(psi_sym)))
print('approximation:', u)
return u, c

```

As test example, we can use the basis

$$\psi_{p,q,r} = \sin((p+1)\pi x) \sin((q+1)\pi y) \sin((r+1)\pi z),$$

for $p = 1, \dots, N_x$, $q = 1, \dots, N_y$, $r = 1, \dots, N_z$. We choose f as some prescribed combination of these functions and check that the computed u is exactly equal to f .

```

def sine_basis(Nx, Ny, Nz):
    """
    Compute basis sin((p+1)*pi*x)*sin((q+1)*pi*y)*sin((r+1)*pi*z),
    p=0,...,Nx, q=0,...,Ny, r=0,...,Nz.
    """
    x, y, z = sym.symbols('x y z')
    psi = []
    for r in range(0, Nz+1):
        for q in range(0, Ny+1):
            for p in range(0, Nx+1):

```

```

        s = sym.sin((p+1)*sym.pi*x)*\
            sym.sin((q+1)*sym.pi*y)*sym.sin((r+1)*sym.pi*z)
        psi.append(s)
    return psi

def test_least_squares():
    # Use sine functions
    x, y, z = sym.symbols('x y z')
    N = 1 # (N+1)**3 = 8 basis functions
    psi = sine_basis(N, N, N)
    f_coeff = [0]*len(psi)
    f_coeff[3] = 2
    f_coeff[4] = 3
    f = sum(f_coeff[i]*psi[i] for i in range(len(psi)))
    # Check that u exactly reproduces f
    u, c = least_squares(f, psi, Omega=[[0,1], [0,1], [0,1]])
    diff = np.abs(np.array(c) - np.array(f_coeff)).max()
    print('diff:', diff)
    tol = 1E-15
    assert diff < tol

```

Filename: approx3D.

Exercise 4.13: Use Simpson's rule and P2 elements

Redo Exercise 4.10, but use P2 elements and Simpson's rule based on sampling the integrands at the nodes in the reference cell.

Solution. Simpson's rule for integrals on $[-1, 1]$ is given by (4.51). The expressions for the entries in the element matrix are given by (4.15):

$$\begin{aligned}\tilde{A}_{r,s}^{(e)} &= \int_{-1}^1 \tilde{\varphi}_r(X) \tilde{\varphi}_s(X) \det J \, dX \\ &\approx \frac{1}{3} \frac{h}{2} (\tilde{\varphi}_r(-1) \tilde{\varphi}_s(-1) + 4\tilde{\varphi}_r(0) \tilde{\varphi}_s(0) + \tilde{\varphi}_r(1) \tilde{\varphi}_s(1)).\end{aligned}$$

The expressions for $\tilde{\varphi}_r(X)$ are given by (4.17)-(4.18). Evaluating the formula for $r, s = 0, 1, 2$ gives the element matrix

$$\tilde{A}^{(e)} = \frac{h}{6} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

As usual, h is the length of the element in physical coordinates.

The element vector in the reference element is given by (4.16):

$$\begin{aligned}\tilde{b}_r^{(e)} &= \int_{-1}^1 f(x(X))\tilde{\varphi}_r(X) \det J \, dX \\ &\approx \frac{1}{3} \frac{h}{2} (f(x(-1))\tilde{\varphi}_r(-1) + 4f(x(0))\tilde{\varphi}_r(0) + f(x(1))\tilde{\varphi}_r(1)).\end{aligned}$$

Evaluating the formula for $r = 0, 1, 2$ leads to

$$\tilde{b}^{(e)} = \frac{h}{2} \left(\frac{f(x_L)}{4f(x_c)f(x_R)} \right),$$

where x_L , x_c , and x_R are the x coordinates of the local points $X = -1$, $X = 0$, and $X = 1$, respectively. These correspond to the nodes in the element.

With a uniform mesh with nodes $x_i = ih$, the element matrix and vectors assemble to a coefficient matrix

$$\frac{h}{6} \text{diag}(1, 4, 2, 4, 2, 4, \dots, 2, 4, 1),$$

and right-hand side vector

$$\frac{h}{6} (f(x_0), 4f(x_1), 2f(x_2), 4f(x_3), 2f(x_4), \dots, 2f(x_{N_n-2}), 4f(x_{N_n-1}), f(x_{N_n})).$$

The factors $h/6$, 2 and 4 all cancel, so we are left with the solution of the system as

$$c_i = f(x_i).$$

Filename: `fe_P2_simpson.`

Exercise 4.14: Make a 3D code for Lagrange elements of arbitrary order

Extend the code from Section 4.7.2 to 3D.

The finite element method is a very flexible approach for solving partial differential equations. Its two most attractive features are the ease of handling domains of complex shape in two and three dimensions and the variety of polynomials (with different properties and orders) that are available. The latter feature typically leads to errors proportional to h^{d+1} , where h is the element length and d is the polynomial degree. When the solution is sufficiently smooth, the ability to use larger d creates methods that are much more computationally efficient than standard finite difference methods (and equally efficient finite difference methods are technically much harder to construct).

However, before we attack finite element methods, with localized basis functions, it can be easier from a pedagogical point of view to study approximations by global functions because the mathematics in this case gets simpler.

5.1 Basic principles for approximating differential equations

The finite element method is usually applied for discretization in space, and therefore spatial problems will be our focus in the coming sections. Extensions to time-dependent problems usually employs finite difference approximations in time.

The coming sections address at how global basis functions and the least squares, Galerkin, and collocation principles can be used to solve differential equations.

5.1.1 Differential equation models

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

$$\mathcal{L}(u) = 0, \quad x \in \Omega. \quad (5.1)$$

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

$$\mathcal{L}(u) = \frac{d^2u}{dx^2} - f(x), \quad (5.2)$$

$$\mathcal{L}(u) = \frac{d}{dx} \left(\alpha(x) \frac{du}{dx} \right) + f(x), \quad (5.3)$$

$$\mathcal{L}(u) = \frac{d}{dx} \left(\alpha(u) \frac{du}{dx} \right) - au + f(x), \quad (5.4)$$

$$\mathcal{L}(u) = \frac{d}{dx} \left(\alpha(u) \frac{du}{dx} \right) + f(u, x). \quad (5.5)$$

Both $\alpha(x)$ and $f(x)$ are considered as specified functions, while a is a prescribed parameter. Differential equations corresponding to (5.2)-(5.3) arise in diffusion phenomena, such as stationary (time-independent) transport of heat in solids and flow of viscous fluids between flat plates. The form (5.4) arises when transient diffusion or wave phenomena are discretized in time by finite differences. The equation (5.5) appears in chemical models when diffusion of a substance is combined with chemical reactions. Also in biology, (5.5) plays an important role, both for spreading of species and in models involving generation and propagation of electrical signals.

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

$$\mathcal{B}_0(u) = 0, \quad x = 0, \quad \mathcal{B}_1(u) = 0, \quad x = L \quad (5.6)$$

There are three common choices of boundary conditions:

$$\mathcal{B}_i(u) = u - g, \quad \text{Dirichlet condition} \quad (5.7)$$

$$\mathcal{B}_i(u) = -\alpha \frac{du}{dx} - g, \quad \text{Neumann condition} \quad (5.8)$$

$$\mathcal{B}_i(u) = -\alpha \frac{du}{dx} - H(u - g), \quad \text{Robin condition} \quad (5.9)$$

Here, g and H are specified quantities.

From now on we shall use $u_e(x)$ as symbol for the *exact* solution, fulfilling

$$\mathcal{L}(u_e) = 0, \quad x \in \Omega, \quad (5.10)$$

while $u(x)$ is our notation for an *approximate* solution of the differential equation.

Remark on notation

In the literature about the finite element method, it is common to use u as the exact solution and u_h as the approximate solution, where h is a discretization parameter. However, the vast part of the present text is about the approximate solutions, and having a subscript h attached all the time is cumbersome. Of equal importance is the close correspondence between implementation and mathematics that we strive to achieve in this text: when it is natural to use u and not u_h in code, we let the mathematical notation be dictated by the code's preferred notation. In the relatively few cases where we need to work with the exact solution of the PDE problem we call it u_e in mathematics and u_e in the code (the function for computing u_e is named `u_exact`).

5.1.2 Simple model problems and their solutions

A common model problem used much in the forthcoming examples is

$$-u''(x) = f(x), \quad x \in \Omega = [0, L], \quad u(0) = 0, \quad u(L) = D. \quad (5.11)$$

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

$$-u''(x) = f(x), \quad x \in \Omega = [0, L], \quad u'(0) = C, \quad u(L) = D. \quad (5.12)$$

A third variant has a variable coefficient,

$$-(\alpha(x)u'(x))' = f(x), \quad x \in \Omega = [0, L], \quad u'(0) = C, \quad u(L) = D. \quad (5.13)$$

The solution u to the model problem (5.11) can be determined as

$$\begin{aligned} u'(x) &= - \int_0^x f(x) + c_0, \\ u(x) &= \int_0^x u'(x) + c_1, \end{aligned}$$

where c_0 and c_1 are determined by the boundary conditions such that $u'(0) = C$ and $u(L) = D$.

Computing the solution is easily done using `sympy`. Some common code is defined first:

```
import sympy as sym
x, L, C, D, c_0, c_1, = sym.symbols('x L C D c_0 c_1')
```

The following function computes the solution symbolically for the model problem (5.11):

```
def model1(f, L, D):
    """Solve -u'' = f(x), u(0)=0, u(L)=D."""
    # Integrate twice
    u_x = - sym.integrate(f, (x, 0, x)) + c_0
    u = sym.integrate(u_x, (x, 0, x)) + c_1
    # Set up 2 equations from the 2 boundary conditions and solve
    # with respect to the integration constants c_0, c_1
    r = sym.solve([u.subs(x, 0)-0, # x=0 condition
                  u.subs(x,L)-D], # x=L condition
                  [c_0, c_1])        # unknowns
    # Substitute the integration constants in the solution
    u = u.subs(c_0, r[c_0]).subs(c_1, r[c_1])
    u = sym.simplify(sym.expand(u))
    return u
```

Calling `model1(2, L, D)` results in the solution

$$u(x) = \frac{1}{L}x \left(D + L^2 - Lx \right) \quad (5.14)$$

The model problem (5.12) can be solved by

```
def model2(f, L, C, D):
    """Solve -u' = f(x), u'(0)=C, u(L)=D."""
    u_x = - sym.integrate(f, (x, 0, x)) + c_0
    u = sym.integrate(u_x, (x, 0, x)) + c_1
    r = sym.solve([sym.diff(u,x).subs(x, 0)-C, # x=0 cond.
                  u.subs(x,L)-D], # x=L cond.
                  [c_0, c_1])
    u = u.subs(c_0, r[c_0]).subs(c_1, r[c_1])
    u = sym.simplify(sym.expand(u))
    return u
```

to yield

$$u(x) = -x^2 + Cx - CL + D + L^2, \quad (5.15)$$

if $f(x) = 2$. Model (5.13) requires a bit more involved code,

```
def model3(f, a, L, C, D):
    """Solve -(a*u)' = f(x), u(0)=C, u(L)=D."""
    au_x = - sym.integrate(f, (x, 0, x)) + c_0
    u = sym.integrate(au_x/a, (x, 0, x)) + c_1
    r = sym.solve([u.subs(x, 0)-C,
                  u.subs(x,L)-D],
                  [c_0, c_1])
    u = u.subs(c_0, r[c_0]).subs(c_1, r[c_1])
    u = sym.simplify(sym.expand(u))
    return u

def demo():
    f = 2
    u = model1(f, L, D)
    print('model1:', u, u.subs(x, 0), u.subs(x, L))
    print(sym.latex(u, mode='plain'))
    u = model2(f, L, C, D)
    #f = x
    #u = model2(f, L, C, D)
    print('model2:', u, sym.diff(u, x).subs(x, 0), u.subs(x, L))
    print(sym.latex(u, mode='plain'))
    u = model3(0, 1+x**2, L, C, D)
    print('model3:', u, u.subs(x, 0), u.subs(x, L))
    print(sym.latex(u, mode='plain'))

if __name__ == '__main__':
    demo()
```

With $f(x) = 0$ and $\alpha(x) = 1 + x^2$ we get

$$u(x) = \frac{C \tan^{-1}(L) - C \tan^{-1}(x) + D \tan^{-1}(x)}{\tan^{-1}(L)}$$

5.1.3 Forming the residual

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

$$V = \text{span}\{\psi_0(x), \dots, \psi_N(x)\},$$

which means that u can always be expressed as a linear combination of the basis functions $\{\psi_j\}_{j \in \mathcal{I}_s}$, with \mathcal{I}_s as the index set $\{0, \dots, N\}$:

$$u(x) = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x).$$

The coefficients $\{c_j\}_{j \in \mathcal{I}_s}$ are unknowns to be computed.

(Later, in Section 6.2, we will see that if we specify boundary values of u different from zero, we must look for an approximate solution $u(x) = B(x) + \sum_j c_j \psi_j(x)$, where $\sum_j c_j \psi_j \in V$ and $B(x)$ is some function for incorporating the right boundary values. Because of $B(x)$, u will not necessarily lie in V . This modification does not imply any difficulties.)

We need principles for deriving $N+1$ equations to determine the $N+1$ unknowns $\{c_i\}_{i \in \mathcal{I}_s}$. When approximating a given function f by $u = \sum_j c_j \varphi_j$, a key idea is to minimize the square norm of the approximation error $e = u - f$ or (equivalently) demand that e is orthogonal to V . Working with e is not so useful here since the approximation error in our case is $e = u_e - u$ and u_e is unknown. The only general indicator we have on the quality of the approximate solution is to what degree u fulfills the differential equation. Inserting $u = \sum_j c_j \psi_j$ into $\mathcal{L}(u)$ reveals that the result is not zero, because u in general is an approximation and not identical to u_e . The nonzero result,

$$R = \mathcal{L}(u) = \mathcal{L}\left(\sum_j c_j \psi_j\right), \quad (5.16)$$

is called the *residual* and measures the error in fulfilling the governing equation.

Various principles for determining $\{c_j\}_{j \in \mathcal{I}_s}$ try to minimize R in some sense. Note that R varies with x and the $\{c_j\}_{j \in \mathcal{I}_s}$ parameters. We may write this dependence explicitly as

$$R = R(x; c_0, \dots, c_N). \quad (5.17)$$

Below, we present three principles for making R small: a least squares method, a projection or Galerkin method, and a collocation or interpolation method.

5.1.4 The least squares method

The least-squares method aims to find $\{c_i\}_{i \in \mathcal{I}_s}$ such that the square norm of the residual

$$\|R\| = (R, R) = \int_{\Omega} R^2 dx \quad (5.18)$$

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

$$(f, g) = \int_{\Omega} f(x)g(x) dx, \quad (5.19)$$

the least-squares method can be defined as

$$\min_{c_0, \dots, c_N} E = (R, R). \quad (5.20)$$

Differentiating with respect to the free parameters $\{c_i\}_{i \in \mathcal{I}_s}$ gives the $N + 1$ equations

$$\int_{\Omega} 2R \frac{\partial R}{\partial c_i} dx = 0 \Leftrightarrow (R, \frac{\partial R}{\partial c_i}) = 0, \quad i \in \mathcal{I}_s. \quad (5.21)$$

5.1.5 The Galerkin method

The least-squares principle is equivalent to demanding the error to be orthogonal to the space V when approximating a function f by $u \in V$. With a differential equation we do not know the true error so we must instead require the residual R to be orthogonal to V . This idea implies seeking $\{c_i\}_{i \in \mathcal{I}_s}$ such that

$$(R, v) = 0, \quad \forall v \in V. \quad (5.22)$$

This is the Galerkin method for differential equations.

The above abstract statement can be made concrete by choosing a concrete basis. For example, the statement is equivalent to R being

orthogonal to the $N + 1$ basis functions $\{\psi_i\}$ spanning V (and this is the most convenient way to express (5.22)):

$$(R, \psi_i) = 0, \quad i \in \mathcal{I}_s, \quad (5.23)$$

resulting in $N + 1$ equations for determining $\{c_i\}_{i \in \mathcal{I}_s}$.

5.1.6 The method of weighted residuals

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

$$(R, v) = 0, \quad \forall v \in W. \quad (5.24)$$

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

$$(R, w_i) = 0, \quad i \in \mathcal{I}_s. \quad (5.25)$$

The result is $N + 1$ equations for $\{c_i\}_{i \in \mathcal{I}_s}$.

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

Variational formulation of the continuous problem

Statements like (5.22), (5.23), (5.24), or (5.25)) are known as **weak formulations** or *variational formulations*. These equations are in this text primarily used for a numerical approximation $u \in V$, where V is a *finite-dimensional* space with dimension $N + 1$. However, we may also let the exact solution u_e fulfill a variational formulation $(\mathcal{L}(u_e), v) = 0 \quad \forall v \in V$, but the exact solution lies in general in a space with infinite dimensions (because an infinite number of parameters are needed to specify the solution). The variational formulation for u_e in an infinite-dimensional space V is a mathematical way of stating the problem and acts as an alternative to the usual (strong) formulation of a differential equation with initial and/or boundary conditions.

Much of the literature on finite element methods takes a differential equation problem and first transforms it to a variational formulation in an infinite-dimensional space V , before searching for an approximate solution in a finite-dimensional subspace of V . However, we prefer the more intuitive approach with an approximate solution u in a finite-dimensional space V inserted in the differential equation, and then the resulting residual is demanded to be orthogonal to V .

Remark on terminology

The terms weak or variational formulations often refer to a statement like (5.22) or (5.24) after *integration by parts* has been performed (the integration by parts technique is explained in Section 5.1.11). The result after integration by parts is what is obtained after taking the *first variation* of a minimization problem (see Section 5.2.4). However, in this text we use variational formulation as a common term for formulations which, in contrast to the differential equation $R = 0$, instead demand that an average of R is zero: $(R, v) = 0$ for all v in some space.

5.1.7 The method of weighted residual and the truncation error

In finite difference methods the concept *truncation error* is often used to analyze schemes. In our case the continuous problem is:

$$\mathcal{L}u = 0,$$

and the corresponding discrete problem is:

$$\mathcal{L}_h u_h = 0.$$

The truncation error is defined in terms of the *discrete operator* \mathcal{L}_h applied to the *continuous solution* u as follows:

$$t_h = \mathcal{L}_h u.$$

A finite difference method is consistent if $\|t_h\| \rightarrow 0$ as $h \rightarrow 0$.

To relate the truncation error to the weighted residual, we multiply by a test function v and integrate

$$(t_h, w) = (\mathcal{L}_h u, v) = (\mathcal{L} u, v) = (\mathcal{L} u_h, v) = (R, v) = 0 \quad \forall v \in W.$$

Hence, both the truncation error and the residual are orthogonal to our test space W and hence consistent.

5.1.8 Test and trial functions

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

We see that in the method of weighted residuals the test and trial spaces are different and so are the test and trial functions. In the Galerkin method the test and trial spaces are the same (so far).

5.1.9 The collocation method

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

$$R(x_i; c_0, \dots, c_N) = 0, \quad i \in \mathcal{I}_s. \quad (5.26)$$

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

$$\int_{\Omega} f(x) \delta(x - x_i) dx = f(x_i), \quad x_i \in \Omega. \quad (5.27)$$

Intuitively, we may think of $\delta(x - x_i)$ as a very peak-shaped function around $x = x_i$ with an integral $\int_{-\infty}^{\infty} \delta(x - x_i) dx$ that evaluates to unity. Mathematically, it can be shown that $\delta(x - x_i)$ is the limit of a Gaussian

function centered at $x = x_i$ with a standard deviation that approaches zero. Using this latter model, we can roughly visualize delta functions as done in Figure 5.1. Because of (5.27), we can let $w_i = \delta(x - x_i)$ be weighting functions in the method of weighted residuals, and (5.25) becomes equivalent to (5.26).

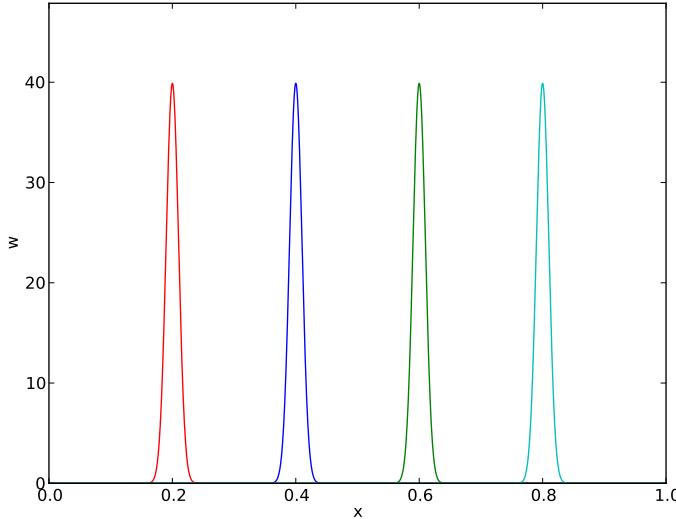


Fig. 5.1 Approximation of delta functions by narrow Gaussian functions.

The subdomain collocation method. The idea of this approach is to demand the integral of R to vanish over $N + 1$ subdomains Ω_i of Ω :

$$\int_{\Omega_i} R \, dx = 0, \quad i \in \mathcal{I}_s. \quad (5.28)$$

This statement can also be expressed as a weighted residual method

$$\int_{\Omega} R w_i \, dx = 0, \quad i \in \mathcal{I}_s, \quad (5.29)$$

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

5.1.10 Examples on using the principles

Let us now apply global basis functions to illustrate the different principles for making the residual R small.

The model problem. We consider the differential equation problem

$$-u''(x) = f(x), \quad x \in \Omega = [0, L], \quad u(0) = 0, \quad u(L) = 0. \quad (5.30)$$

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

$$\psi_i(x) = \sin\left((i+1)\pi\frac{x}{L}\right), \quad i \in \mathcal{I}_s. \quad (5.31)$$

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

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

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

$$\int_0^L \sin\left((i+1)\pi\frac{x}{L}\right) \sin\left((j+1)\pi\frac{x}{L}\right) dx = \begin{cases} \frac{1}{2}L & i = j \\ 0 & i \neq j \end{cases} \quad (5.32)$$

provided i and j are integers.

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

$$\begin{aligned} R(x; c_0, \dots, c_N) &= u''(x) + f(x), \\ &= \frac{d^2}{dx^2} \left(\sum_{j \in \mathcal{I}_s} c_j \psi_j(x) \right) + f(x), \\ &= \sum_{j \in \mathcal{I}_s} c_j \psi_j''(x) + f(x). \end{aligned} \quad (5.33)$$

The least squares method. The equations (5.21) in the least squares method require an expression for $\partial R / \partial c_i$. We have

$$\frac{\partial R}{\partial c_i} = \frac{\partial}{\partial c_i} \left(\sum_{j \in \mathcal{I}_s} c_j \psi_j''(x) + f(x) \right) = \sum_{j \in \mathcal{I}_s} \frac{\partial c_j}{\partial c_i} \psi_j''(x) = \psi_i''(x). \quad (5.34)$$

The governing equations for the unknown parameters $\{c_j\}_{j \in \mathcal{I}_s}$ are then

$$(\sum_j c_j \psi_j'' + f, \psi_i'') = 0, \quad i \in \mathcal{I}_s, \quad (5.35)$$

which can be rearranged as

$$\sum_{j \in \mathcal{I}_s} (\psi_i'', \psi_j'') c_j = -(f, \psi_i''), \quad i \in \mathcal{I}_s. \quad (5.36)$$

This is nothing but a linear system

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s.$$

The entries in the coefficient matrix are given by

$$\begin{aligned} A_{i,j} &= (\psi_i'', \psi_j'') \\ &= \pi^4 (i+1)^2 (j+1)^2 L^{-4} \int_0^L \sin((i+1)\pi \frac{x}{L}) \sin((j+1)\pi \frac{x}{L}) \, dx \end{aligned}$$

The orthogonality of the sine functions simplify the coefficient matrix:

$$A_{i,j} = \begin{cases} \frac{1}{2} L^{-3} \pi^4 (i+1)^4 & i = j \\ 0 & i \neq j \end{cases} \quad (5.37)$$

The right-hand side reads

$$b_i = -(f, \psi_i'') = (i+1)^2 \pi^2 L^{-2} \int_0^L f(x) \sin((i+1)\pi \frac{x}{L}) \, dx \quad (5.38)$$

Since the coefficient matrix is diagonal we can easily solve for

$$c_i = \frac{2L}{\pi^2(i+1)^2} \int_0^L f(x) \sin((i+1)\pi \frac{x}{L}) \, dx. \quad (5.39)$$

With the special choice of $f(x) = 2$, the coefficients can be calculated in `sympy` by

```
import sympy as sym
```

```
i, j = sym.symbols('i j', integer=True)
x, L = sym.symbols('x L')
f = 2
a = 2*L/(sym.pi**2*(i+1)**2)
c_i = a*sym.integrate(f*sym.sin((i+1)*sym.pi*x/L), (x, 0, L))
c_i = sym.simplify(c_i)
print(c_i)
```

The answer becomes

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

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

$$u(x) = \sum_{k=0}^{N/2} \frac{8L^2}{\pi^3 (2k+1)^3} \sin\left((2k+1)\pi \frac{x}{L}\right). \quad (5.40)$$

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

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

The Galerkin method. The Galerkin principle (5.22) applied to (5.30) consists of inserting our special residual (5.33) in (5.22)

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

or

$$(u'', v) = -(f, v), \quad \forall v \in V. \quad (5.41)$$

This is the variational formulation, based on the Galerkin principle, of our differential equation. The $\forall v \in V$ requirement is equivalent to demanding the equation $(u'', v) = -(f, v)$ to be fulfilled for all basis functions $v = \psi_i$, $i \in \mathcal{I}_s$, see (5.22) and (5.23). We therefore have

$$\left(\sum_{j \in \mathcal{I}_s} c_j \psi_j'', \psi_i \right) = -(f, \psi_i), \quad i \in \mathcal{I}_s. \quad (5.42)$$

This equation can be rearranged to a form that explicitly shows that we get a linear system for the unknowns $\{c_j\}_{j \in \mathcal{I}_s}$:

$$\sum_{j \in \mathcal{I}_s} (\psi_i, \psi_j'') c_j = (f, \psi_i), \quad i \in \mathcal{I}_s. \quad (5.43)$$

For the particular choice of the basis functions (5.31) we get in fact the same linear system as in the least squares method because $\psi'' = -(i+1)^2\pi^2 L^{-2}\psi$. Consequently, the solution $u(x)$ becomes identical to the one produced by the least squares method.

The collocation method. For the collocation method (5.26) we need to decide upon a set of $N + 1$ collocation points in Ω . A simple choice is to use uniformly spaced points: $x_i = i\Delta x$, where $\Delta x = L/N$ in our case ($N \geq 1$). However, these points lead to at least two rows in the matrix consisting of zeros (since $\psi_i(x_0) = 0$ and $\psi_i(x_N) = 0$), thereby making the matrix singular and non-invertible. This forces us to choose some other collocation points, e.g., random points or points uniformly distributed in the interior of Ω . Demanding the residual to vanish at these points leads, in our model problem (5.30), to the equations

$$-\sum_{j \in \mathcal{I}_s} c_j \psi_j''(x_i) = f(x_i), \quad i \in \mathcal{I}_s, \quad (5.44)$$

which is seen to be a linear system with entries

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

in the coefficient matrix and entries $b_i = 2$ for the right-hand side (when $f(x) = 2$).

The special case of $N = 0$ can sometimes be of interest. A natural choice is then the midpoint $x_0 = L/2$ of the domain, resulting in $A_{0,0} = -\psi_0''(x_0) = \pi^2 L^{-2}$, $f(x_0) = 2$, and hence $c_0 = 2L^2/\pi^2$.

Comparison. In the present model problem, with $f(x) = 2$, the exact solution is $u(x) = x(L-x)$, while for $N = 0$ the Galerkin and least squares method result in $u(x) = 8L^2\pi^{-3}\sin(\pi x/L)$ and the collocation method leads to $u(x) = 2L^2\pi^{-2}\sin(\pi x/L)$. We can quickly use `sympy` to verify that the maximum error occurs at the midpoint $x = L/2$ and find what the errors are. First we set up the error expressions:

```
>>> import sympy as sym
>>> # Computing with Dirichlet conditions: -u''=2 and sines
>>> x, L = sym.symbols('x L')
>>> e_Galerkin = x*(L-x) - 8*L**2*sym.pi**(-3)*sym.sin(sym.pi*x/L)
>>> e_colloc = x*(L-x) - 2*L**2*sym.pi**(-2)*sym.sin(sym.pi*x/L)
```

If the derivative of the errors vanish at $x = L/2$, the errors reach their maximum values here (the errors vanish at the boundary points).

```
>>> dedx_Galerkin = sym.diff(e_Galerkin, x)
>>> dedx_Galerkin.subs(x, L/2)
0
>>> dedx_colloc = sym.diff(e_colloc, x)
>>> dedx_colloc.subs(x, L/2)
0
```

Finally, we can compute the maximum error at $x = L/2$ and evaluate the expressions numerically with three decimals:

```
>>> sym.simplify(e_Galerkin.subs(x, L/2).evalf(n=3))
-0.00812*L**2
>>> sym.simplify(e_colloc.subs(x, L/2).evalf(n=3))
0.0473*L**2
```

The error in the collocation method is about 6 times larger than the error in the Galerkin or least squares method.

5.1.11 Integration by parts

A problem arises if we want to apply popular finite element functions to solve our model problem (5.30) by the standard least squares, Galerkin, or collocation methods: the piecewise polynomials $\psi_i(x)$ have discontinuous derivatives at the cell boundaries which makes it problematic to compute the second-order derivative. This fact actually makes the least squares and collocation methods less suitable for finite element approximation of the unknown function. (By rewriting the equation $-u'' = f$ as a system of two first-order equations, $u' = v$ and $-v' = f$, the least squares method can be applied. Also, differentiating discontinuous functions can actually be handled by distribution theory in mathematics.) The Galerkin method and the method of weighted residuals can, however, be applied together with finite element basis functions if we use *integration by parts* as a means for transforming a second-order derivative to a first-order one.

Consider the model problem (5.30) and its Galerkin formulation

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

Using integration by parts in the Galerkin method, we can “move” a derivative of u onto v :

$$\begin{aligned}
\int_0^L u''(x)v(x) \, dx &= - \int_0^L u'(x)v'(x) \, dx + [vu']_0^L \\
&= - \int_0^L u'(x)v'(x) \, dx + u'(L)v(L) - u'(0)v(0).
\end{aligned} \tag{5.45}$$

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

$$\begin{aligned}
\int_0^L \psi_i(x)\psi_j''(x) \, dx &= - \int_0^L \psi_i'(x)\psi_j'(x) \, dx + [\psi_i\psi_j']_0^L \\
&= - \int_0^L \psi_i'(x)\psi_j'(x) \, dx + \psi_i(L)\psi_j'(L) - \psi_i(0)\psi_j'(0).
\end{aligned} \tag{5.46}$$

Integration by parts serves to reduce the order of the derivatives and to make the coefficient matrix symmetric since $(\psi_i', \psi_j') = (\psi_j', \psi_i')$. The symmetry property depends on the type of terms that enter the differential equation. As will be seen later in Section 6.3, integration by parts also provides a method for implementing boundary conditions involving u' .

With the choice (5.31) of basis functions we see that the “boundary terms” $\psi_i(L)\psi_j'(L)$ and $\psi_i(0)\psi_j'(0)$ vanish since $\psi_i(0) = \psi_i(L) = 0$. We therefore end up with the following alternative Galerkin formulation:

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

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

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

5.1.12 Boundary function

So far we have assumed zero Dirichlet boundary conditions, typically $u(0) = u(L) = 0$, and we have demanded that $\psi_i(0) = \psi_i(L) = 0$ for $i \in \mathcal{I}_s$. What about a boundary condition like $u(L) = D \neq 0$? This condition immediately faces a problem: $u = \sum_j c_j \varphi_j(L) = 0$ since all $\varphi_i(L) = 0$.

We remark that we faced exactly the same problem in Section 3.3.2 where we considered Fourier series approximations of functions that were non-zero at the boundaries. We will use the same trick as we did earlier to get around this problem.

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

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x).$$

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

$$u(L) = B(L) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(L) = B(L) = D.$$

The idea is that for any boundary where u is known we demand ψ_i to vanish and construct a function $B(x)$ to attain the boundary value of u . There are no restrictions on how $B(x)$ varies with x in the interior of the domain, so this variation needs to be constructed in some way. Exactly how we decide the variation to be, is not important.

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

$$u(x) = \frac{x}{L} D + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x), \quad (5.47)$$

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

The particular shape of the $B(x)$ function is not important as long as its boundary values are correct. For example, $B(x) = D(x/L)^p$ for any power p will work fine in the above example. Another choice could be $B(x) = D \sin(\pi x/(2L))$.

As a more general example, consider a domain $\Omega = [a, b]$ where the boundary conditions are $u(a) = U_a$ and $u(b) = U_b$. A class of possible $B(x)$ functions is

$$B(x) = U_a + \frac{U_b - U_a}{(b - a)^p} (x - a)^p, \quad p > 0. \quad (5.48)$$

Real applications will most likely use the simplest version, $p = 1$, but here such a p parameter was included to demonstrate that there are many choices of $B(x)$ in a problem. Fortunately, there is a general, unique technique for constructing $B(x)$ when we use finite element basis functions for V .

How to deal with nonzero Dirichlet conditions

The general procedure of incorporating Dirichlet boundary conditions goes as follows. Let $\partial\Omega_E$ be the part(s) of the boundary $\partial\Omega$ of the domain Ω where u is specified. Set $\psi_i = 0$ at the points in $\partial\Omega_E$ and seek u as

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x), \quad (5.49)$$

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

Remark. With the $B(x)$ term, u does not in general lie in $V = \text{span}\{\psi_0, \dots, \psi_N\}$ anymore. Moreover, when a prescribed value of u at the boundary, say $u(a) = U_a$ is different from zero, it does not make sense to say that u lies in a vector space, because this space does not obey the requirements of addition and scalar multiplication. For example, $2u$ does not lie in the space since its boundary value is $2U_a$, which is incorrect. It only makes sense to split u in two parts, as done above, and have the unknown part $\sum_j c_j \psi_j$ in a proper function space.

5.2 Computing with global polynomials

The next example uses global polynomials and shows that if our solution, modulo boundary conditions, lies in the space spanned by these polynomials, then the Galerkin method recovers the exact solution.

5.2.1 Computing with Dirichlet and Neumann conditions

Let us perform the necessary calculations to solve

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

using a global polynomial basis $\psi_i \sim x^i$. The requirements on ψ_i is that $\psi_i(1) = 0$, because u is specified at $x = 1$, so a proper set of polynomial basis functions can be

$$\psi_i(x) = (1-x)^{i+1}, \quad i \in \mathcal{I}_s.$$

A suitable $B(x)$ function to handle the boundary condition $u(1) = D$ is $B(x) = Dx$. The variational formulation becomes

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

From inserting $u = B + \sum_j c_j \psi_j$ and choosing $v = \psi_i$ we get

$$\sum_{j \in \mathcal{I}_s} (\psi'_j, \psi'_i) c_j = (2, \psi_i) - (B', \psi'_i) - C\psi_i(0), \quad i \in \mathcal{I}_s.$$

The entries in the linear system are then

$$\begin{aligned} A_{i,j} &= (\psi'_j, \psi'_i) = \int_0^1 \psi'_i(x) \psi'_j(x) dx = \int_0^1 (i+1)(j+1)(1-x)^{i+j} dx \\ &= \frac{(i+1)(j+1)}{i+j+1}, \\ b_i &= (2, \psi_i) - (D, \psi'_i) - C\psi_i(0) \\ &= \int_0^1 (2\psi_i(x) - D\psi'_i(x)) dx - C\psi_i(0) \\ &= \int_0^1 \left(2(1-x)^{i+1} + D(i+1)(1-x)^i\right) dx - C \\ &= \frac{(D-C)(i+2)+2}{i+2} = D - C + \frac{2}{i+2}. \end{aligned}$$

Relevant `sympy` commands to help calculate these expressions are

```
from sympy import *
x, C, D = symbols('x C D')
i, j = symbols('i j', integer=True, positive=True)
psi_i = (1-x)**(i+1)
psi_j = psi_i.subs(i, j)
integrand = diff(psi_i, x)*diff(psi_j, x)
```

```

integrand = simplify(integrand)
A_ij = integrate(integrand, (x, 0, 1))
A_ij = simplify(A_ij)
print('A_ij:', A_ij)
f = 2
b_i = integrate(f*psi_i, (x, 0, 1)) - \
    integrate(diff(D*x, x)*diff(psi_i, x), (x, 0, 1)) - \
    C*psi_i.subs(x, 0)
b_i = simplify(b_i)
print('b_i:', b_i)

```

The output becomes

```

A_ij: (i + 1)*(j + 1)/(i + j + 1)
b_i: ((-C + D)*(i + 2) + 2)/(i + 2)

```

We can now choose some N and form the linear system, say for $N = 1$:

```

N = 1
A = zeros(N+1, N+1)
b = zeros(N+1)
print('fresh b:', b)
for r in range(N+1):
    for s in range(N+1):
        A[r,s] = A_ij.subs(i, r).subs(j, s)
    b[r,0] = b_i.subs(i, r)

```

The system becomes

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

The solution (`c = A.LUsolve(b)`) becomes $c_0 = 2 - C + D$ and $c_1 = -1$, resulting in

$$u(x) = 1 - x^2 + D + C(x - 1), \quad (5.50)$$

We can form this u in `sympy` and check that the differential equation and the boundary conditions are satisfied:

```

u = sum(c[r,0]*psi_i.subs(i, r) for r in range(N+1)) + D*x
print('u:', simplify(u))
print("u'':", simplify(diff(u, x, x)))
print('BC x=0:', simplify(diff(u, x).subs(x, 0)))
print('BC x=1:', simplify(u.subs(x, 1)))

```

The output becomes

```

u: C*x - C + D - x**2 + 1
u'': -2
BC x=0: C
BC x=1: D

```

The complete `sympy` code is found in `u_xx_2_CD.py`.

The exact solution is found by integrating twice and applying the boundary conditions, either by hand or using `sympy` as shown in Section 5.1.2. It appears that the numerical solution coincides with the exact one. This result is to be expected because if $(u_e - B) \in V$, $u = u_e$, as proved next.

5.2.2 When the numerical method is exact

We have some variational formulation: find $(u - B) \in V$ such that $a(u, v) = L(u) \forall v \in V$. The exact solution also fulfills $a(u_e, v) = L(v)$, but normally $(u_e - B)$ lies in a much larger (infinite-dimensional) space. Suppose, nevertheless, that $u_e - B = E$, where $E \in V$. That is, apart from Dirichlet conditions, u_e lies in our finite-dimensional space V which we use to compute u . Writing also u on the same form $u = B + F$, $F \in V$, we have

$$\begin{aligned} a(u_e, v) &= a(B + E, v) &= L(v) \quad \forall v \in V, \\ a(u, v) &= a(B + F, v) &= L(v) \quad \forall v \in V. \end{aligned}$$

Since these are two variational statements in the same space, we can subtract them and use the bilinear property of $a(\cdot, \cdot)$:

$$\begin{aligned} a(B + E, v) - a(B + F, v) &= L(v) - L(v) \\ a(B + E - (B + F), v) &= 0 \\ a(E - F, v) &= 0 \end{aligned}$$

If $a(E - F, v) = 0$ for all v in V , then $E - F$ must be zero everywhere in the domain, i.e., $E = F$. Or in other words: $u = u_e$. This proves that the exact solution is recovered if $u_e - B$ lies in V , i.e., can be expressed as $\sum_{j \in \mathcal{I}_s} d_j \psi_j$ where $\{\psi_j\}_{j \in \mathcal{I}_s}$ is a basis for V . The method will then compute the solution $c_j = d_j$, $j \in \mathcal{I}_s$.

The case treated in Section 5.2.1 is of the type where $u_e - B$ is a quadratic function that is 0 at $x = 1$, and therefore $(u_e - B) \in V$, and the method finds the exact solution.

5.2.3 Abstract notation for variational formulations

We have seen that variational formulations end up with a formula involving u and v , such as (u', v') and a formula involving v and known functions, such as (f, v) . A widely used notation is to introduce an abstract variational statement written as

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

where $a(u, v)$ is a so-called *bilinear form* involving all the terms that contain both the test and trial function, while $L(v)$ is a *linear form* containing all the terms without the trial function. For example, the statement

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

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

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

where we have the definitions

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

The term *linear* means that

$$L(\alpha_1 v_1 + \alpha_2 v_2) = \alpha_1 L(v_1) + \alpha_2 L(v_2)$$

for two test functions v_1 and v_2 , and scalar parameters α_1 and α_2 . Similarly, the term *bilinear* means that $a(u, v)$ is linear in both its arguments:

$$\begin{aligned} a(\alpha_1 u_1 + \alpha_2 u_2, v) &= \alpha_1 a(u_1, v) + \alpha_2 a(u_2, v), \\ a(u, \alpha_1 v_1 + \alpha_2 v_2) &= \alpha_1 a(u, v_1) + \alpha_2 a(u, v_2). \end{aligned}$$

In nonlinear problems these linearity properties do not hold in general and the abstract notation is then

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

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

$$\sum_{j \in \mathcal{I}_s} a(\psi_j, \psi_i) c_j = L(\psi_i), \quad i \in \mathcal{I}_s,$$

which is a linear system

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s,$$

where

$$A_{i,j} = a(\psi_j, \psi_i), \quad b_i = L(\psi_i).$$

In many problems, $a(u, v)$ is symmetric such that $a(\psi_j, \psi_i) = a(\psi_i, \psi_j)$. In those cases the coefficient matrix becomes symmetric, $A_{i,j} = A_{j,i}$, a property that can simplify solution algorithms for linear systems and make them more stable. The property also reduces memory requirements and the computational work.

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

5.2.4 Variational problems and minimization of functionals

Example. Many physical problems can be modeled as partial differential equations and as minimization problems. For example, the deflection $u(x)$ of an elastic string subject to a transversal force $f(x)$ is governed by the differential equation problem

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

Equivalently, the deflection $u(x)$ is the function v that minimizes the potential energy $F(v)$ in a string,

$$F(v) = \frac{1}{2} \int_0^L ((v')^2 - fv) \, dx.$$

That is, $F(u) = \min_{v \in V} F(v)$. The quantity $F(v)$ is called a functional: it takes one or more functions as input and produces a number. Loosely speaking, we may say that a functional is “a function of functions”. Functionals very often involve integral expressions as above.

A range of physical problems can be formulated either as a differential equation or as a minimization of some functional. Quite often, the differential equation arises from Newton's 2nd law of motion while the functional expresses a certain kind of energy.

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

The general minimization problem. The relation between a differential equation and minimization of a functional can be expressed in a general mathematical way using our abstract notation for a variational form: $a(u, v) = L(v)$. It can be shown that the variational statement

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

is equivalent to minimizing the functional

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

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

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

Derivation. To see this, we write $F(u) \leq F(\eta)$, $\forall \eta \in V$ instead and set $\eta = u + \epsilon v$, where $v \in V$ is an arbitrary function in V . For any given arbitrary v , we can view $F(v)$ as a function $g(\epsilon)$ and find the extrema of g , which is a function of one variable. We have

$$F(\eta) = F(u + \epsilon v) = \frac{1}{2}a(u + \epsilon v, u + \epsilon v) - L(u + \epsilon v).$$

From the linearity of a and L we get

$$\begin{aligned} g(\epsilon) &= F(u + \epsilon v) \\ &= \frac{1}{2}a(u + \epsilon v, u + \epsilon v) - L(u + \epsilon v) \\ &= \frac{1}{2}a(u, u + \epsilon v) + \frac{1}{2}\epsilon a(v, u + \epsilon v) - L(u) - \epsilon L(v) \\ &= \frac{1}{2}a(u, u) + \frac{1}{2}\epsilon a(u, v) + \frac{1}{2}\epsilon a(v, u) + \frac{1}{2}\epsilon^2 a(v, v) - L(u) - \epsilon L(v). \end{aligned}$$

If we now assume that a is symmetric, $a(u, v) = a(v, u)$, we can write

$$g(\epsilon) = \frac{1}{2}a(u, u) + \epsilon a(u, v) + \frac{1}{2}\epsilon^2 a(v, v) - L(u) - \epsilon L(v).$$

The extrema of g is found by searching for ϵ such that $g'(\epsilon) = 0$:

$$g'(\epsilon) = a(u, v) - L(v) + \epsilon a(v, v) = 0.$$

This linear equation in ϵ has a solution $\epsilon = (a(u, v) - L(u))/a(v, v)$ if $a(v, v) > 0$. But recall that $a(u, v) = L(v)$ for any v , so we must have $\epsilon = 0$. Since the reasoning above holds for any $v \in V$, the function $\eta = u + \epsilon v$ that makes $F(\eta)$ extreme must have $\epsilon = 0$, i.e., $\eta = u$, the solution of $a(u, v) = L(v)$ for any v in V .

Looking at $g''(\epsilon) = a(v, v)$, we realize that $\epsilon = 0$ corresponds to a unique minimum if $a(v, v) > 0$.

The equivalence of a variational form $a(u, v) = L(v) \forall v \in V$ and the minimization problem $F(u) \leq F(v) \forall v \in V$ requires that 1) a is bilinear and L is linear, 2) $a(u, v)$ is symmetric: $a(u, v) = a(v, u)$, and 3) that $a(v, v) > 0$.

Minimization of the discretized functional. Inserting $v = \sum_j c_j \psi_j$ turns minimization of $F(v)$ into minimization of a quadratic function of the parameters c_0, \dots, c_N :

$$\bar{F}(c_0, \dots, c_N) = \sum_{j \in \mathcal{I}_s} \sum_{i \in \mathcal{I}_s} a(\psi_i, \psi_j) c_i c_j - \sum_{j \in \mathcal{I}_s} L(\psi_j) c_j$$

of $N + 1$ parameters.

Minimization of \bar{F} implies

$$\frac{\partial \bar{F}}{\partial c_i} = 0, \quad i \in \mathcal{I}_s.$$

After some algebra one finds

$$\sum_{j \in \mathcal{I}_s} a(\psi_i, \psi_j) c_j = L(\psi_i), \quad i \in \mathcal{I}_s,$$

which is the same system as the one arising from $a(u, v) = L(v)$.

Calculus of variations. A branch of applied mathematics, called [calculus of variations](#), deals with the technique of minimizing functionals to derive differential equations. The technique involves taking the *variation* (a kind of derivative) of functionals, which have given name to terms like variational form, variational problem, and variational formulation.

5.3 Examples on variational formulations

The following sections derive variational formulations for some prototype differential equations in 1D, and demonstrate how we with ease can handle variable coefficients, mixed Dirichlet and Neumann boundary conditions, first-order derivatives, and nonlinearities.

5.3.1 Variable coefficient

Consider the problem

$$-\frac{d}{dx} \left(\alpha(x) \frac{du}{dx} \right) = f(x), \quad x \in \Omega = [0, L], \quad u(0) = C, \quad u(L) = D. \quad (5.51)$$

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

Let us first deal with the boundary conditions. We seek

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_i(x).$$

Since the Dirichlet conditions demand

$$\psi_i(0) = \psi_i(L) = 0, \quad i \in \mathcal{I}_s,$$

the function $B(x)$ must fulfill $B(0) = C$ and $B(L) = D$. This we are guaranteed that $u(0) = C$ and $u(L) = D$. How B varies in between $x = 0$ and $x = L$ is not of importance. One possible choice is

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

which follows from (5.48) with $p = 1$.

We seek $(u - B) \in V$. As usual,

$$V = \text{span}\{\psi_0, \dots, \psi_N\}.$$

Note that any $v \in V$ has the property $v(0) = v(L) = 0$.

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

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

Galerkin's method is

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

or written with explicit integrals,

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

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

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

The boundary term vanishes since $v(0) = v(L) = 0$. The variational formulation is then

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

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

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

The corresponding abstract notation reads

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

with

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

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

$$(\alpha B' + \alpha \sum_{j \in \mathcal{I}_s} c_j \psi'_j, \psi'_i) = (f, \psi_i), \quad i \in \mathcal{I}_s.$$

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

$$\sum_{j \in \mathcal{I}_s} (\alpha \psi'_j, \psi'_i) c_j = (f, \psi_i) + (\alpha(D - C)L^{-1}, \psi'_i), \quad i \in \mathcal{I}_s.$$

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

$$A_{i,j} = (\alpha\psi'_j, \psi'_i) = \int_{\Omega} \alpha(x)\psi'_j(x), \psi'_i(x) \, dx,$$

$$b_i = (f, \psi_i) + (\alpha(D - C)L^{-1}, \psi'_i) = \int_{\Omega} \left(f(x)\psi_i(x) + \alpha(x)\frac{D - C}{L}\psi'_i(x) \right) \, dx.$$

5.3.2 First-order derivative in the equation and boundary condition

The next problem to formulate in terms of a variational form reads

$$-u''(x) + bu'(x) = f(x), \quad x \in \Omega = [0, L], \quad u(0) = C, \quad u'(L) = E. \quad (5.52)$$

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

The expansion for u becomes

$$u = C + \sum_{j \in \mathcal{I}_s} c_j \psi_i(x).$$

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

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

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

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

Now, $v(0) = 0$ so

$$[u'v]_0^L = u'(L)v(L) = Ev(L),$$

because $u'(L) = E$. Thus, integration by parts allows us to take care of the Neumann condition in the boundary term.

Natural and essential boundary conditions

A common mistake is to forget a boundary term like $[u'v]_0^L$ in the integration by parts. Such a mistake implies that we actually impose the condition $u' = 0$ unless there is a Dirichlet condition (i.e., $v = 0$) at that point! This fact has great practical consequences, because it is easy to forget the boundary term, and that implicitly set a boundary condition!

Since homogeneous Neumann conditions can be incorporated without “doing anything” (i.e., omitting the boundary term), and non-homogeneous Neumann conditions can just be inserted in the boundary term, such conditions are known as *natural boundary conditions*. Dirichlet conditions require more essential steps in the mathematical formulation, such as forcing all $\varphi_i = 0$ on the boundary and constructing a $B(x)$, and are therefore known as *essential boundary conditions*.

The final variational form reads

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

In the abstract notation we have

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

with the particular formulas

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

The associated linear system is derived by inserting $u = B + \sum_j c_j \psi_j$ and replacing v by ψ_i for $i \in \mathcal{I}_s$. Some algebra results in

$$\sum_{j \in \mathcal{I}_s} \underbrace{((\psi'_j, \psi'_i) + (b\psi'_j, \psi_i))}_{A_{i,j}} c_j = \underbrace{(f, \psi_i) + E\psi_i(L)}_{b_i}.$$

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

$$(b\psi'_j, \psi_i) = \int_{\Omega} b\psi'_j \psi_i \, dx \neq \int_{\Omega} b\psi'_i \psi_j \, dx = (\psi'_i, b\psi_j).$$

5.3.3 Nonlinear coefficient

Finally, we show that the techniques used above to derive variational forms apply to nonlinear differential equation problems as well. Here is a model problem with a nonlinear coefficient $\alpha(u)$ and a nonlinear right-hand side $f(u)$:

$$-(\alpha(u)u')' = f(u), \quad x \in [0, L], \quad u(0) = 0, \quad u'(L) = E. \quad (5.53)$$

Our space V has basis $\{\psi_i\}_{i \in \mathcal{I}_s}$, and because of the condition $u(0) = 0$, we must require $\psi_i(0) = 0$, $i \in \mathcal{I}_s$.

Galerkin's method is about inserting the approximate u , multiplying the differential equation by $v \in V$, and integrate,

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

The integration by parts does not differ from the case where we have $\alpha(x)$ instead of $\alpha(u)$:

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

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

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

or alternatively written more compactly as

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

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

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

with

$$F(u; v) = (\alpha(u)u', v') - (f(u), v) - \alpha(u(L))v(L)E.$$

By inserting $u = \sum_j c_j \psi_j$ and $v = \psi_i$ in $F(u; v)$, we get a *nonlinear system of algebraic equations* for the unknowns c_i , $i \in \mathcal{I}_s$. Such systems must be solved by constructing a sequence of linear systems whose solutions hopefully converge to the solution of the nonlinear system. Frequently applied methods are Picard iteration and Newton's method.

5.4 Implementation of the algorithms

Our hand calculations can benefit greatly by symbolic computing, as shown earlier, so it is natural to extend our approximation programs based on `sympy` to the problem domain of variational formulations.

5.4.1 Extensions of the code for approximation

The user must prepare a function `integrand_lhs(psi, i, j)` for returning the integrand of the integral that contributes to matrix entry (i, j) on the left-hand side. The `psi` variable is a Python dictionary holding the basis functions and their derivatives in symbolic form. More precisely, `psi[q]` is a list of

$$\left\{ \frac{d^q \psi_0}{dx^q}, \dots, \frac{d^q \psi_{N_n-1}}{dx^q} \right\}.$$

Similarly, `integrand_rhs(psi, i)` returns the integrand for entry number i in the right-hand side vector.

Since we also have contributions to the right-hand side vector (and potentially also the matrix) from boundary terms without any integral, we introduce two additional functions, `boundary_lhs(psi, i, j)` and `boundary_rhs(psi, i)` for returning terms in the variational formulation that are not to be integrated over the domain Ω . Examples, to be shown later, will explain in more detail how these user-supplied functions may look like.

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

```
import sympy as sym

def solver(integrand_lhs, integrand_rhs, psi, Omega,
           boundary_lhs=None, boundary_rhs=None):
    N = len(psi[0]) - 1
    A = sym.zeros(N+1, N+1)
```

```

b = sym.zeros(N+1, 1)
x = sym.Symbol('x')
for i in range(N+1):
    for j in range(i, N+1):
        integrand = integrand_lhs(psi, i, j)
        I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
        if boundary_lhs is not None:
            I += boundary_lhs(psi, i, j)
        A[i,j] = A[j,i] = I # assume symmetry
        integrand = integrand_rhs(psi, i)
        I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
        if boundary_rhs is not None:
            I += boundary_rhs(psi, i)
        b[i,0] = I
c = A.LUsolve(b)
u = sum(c[i,0]*psi[0][i] for i in range(len(psi[0])))
return u, c

```

5.4.2 Fallback to numerical methods

Not surprisingly, symbolic solution of differential equations, discretized by a Galerkin or least squares method with global basis functions, is of limited interest beyond the simplest problems, because symbolic integration might be very time consuming or impossible, not only in `sympy` but also in `WolframAlpha` (which applies the perhaps most powerful symbolic integration software available today: Mathematica). Numerical integration as an option is therefore desirable.

The extended `solver` function below tries to combine symbolic and numerical integration. The latter can be enforced by the user, or it can be invoked after a non-successful symbolic integration (being detected by an `Integral` object as the result of the integration in `sympy`). Note that for a numerical integration, symbolic expressions must be converted to Python functions (using `lambdify`), and the expressions cannot contain other symbols than `x`. The real `solver` routine in the `varform1D.py` file has error checking and meaningful error messages in such cases. The `solver` code below is a condensed version of the real one, with the purpose of showing how to automate the Galerkin or least squares method for solving differential equations in 1D with global basis functions:

```

def solver(integrand_lhs, integrand_rhs, psi, Omega,
           boundary_lhs=None, boundary_rhs=None, symbolic=True):
    N = len(psi[0]) - 1
    A = sym.zeros(N+1, N+1)
    b = sym.zeros(N+1, 1)
    x = sym.Symbol('x')

```

```

for i in range(N+1):
    for j in range(i, N+1):
        integrand = integrand_lhs(psi, i, j)
        if symbolic:
            I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
            if isinstance(I, sym.Integral):
                symbolic = False # force num.int. hereafter
        if not symbolic:
            integrand_ = sym.lambdify([x], integrand, 'mpmath')
            I = mpmath.quad(integrand_, [Omega[0], Omega[1]])
        if boundary_lhs is not None:
            I += boundary_lhs(psi, i, j)
        A[i,j] = A[j,i] = I
    integrand = integrand_rhs(psi, i)
    if symbolic:
        I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
        if isinstance(I, sym.Integral):
            symbolic = False
    if not symbolic:
        integrand_ = sym.lambdify([x], integrand, 'mpmath')
        I = mpmath.quad(integrand_, [Omega[0], Omega[1]])
    if boundary_rhs is not None:
        I += boundary_rhs(psi, i)
    b[i,0] = I
c = A.LUsolve(b)
u = sum(c[i,0]*psi[0][i] for i in range(len(psi[0])))
return u, c

```

5.4.3 Example with constant right-hand side

To demonstrate the code above, we address

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

with b as a (symbolic) constant. A possible basis for the space V is $\psi_i(x) = x^{i+1}(1-x)$, $i \in \mathcal{I}_s$. Note that $\psi_i(0) = \psi_i(1) = 0$ as required by the Dirichlet conditions. We need a $B(x)$ function to take care of the known boundary values of u . Any function $B(x) = 1 - x^p$, $p \in \mathbb{R}$, is a candidate, and one arbitrary choice from this family is $B(x) = 1 - x^3$. The unknown function is then written as

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x).$$

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

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

and with $u = B + \sum_j c_j \psi_j$ we get the linear system

$$\sum_{j \in \mathcal{I}_s} \left(\int_0^1 \psi_i' \psi_j' \, dx \right) c_j = \int_0^1 (f \psi_i - B' \psi_i') \, dx, \quad i \in \mathcal{I}_s. \quad (5.54)$$

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

```
import sympy as sym
x, b = sym.symbols("x b")
f = b
B = 1 - x**3
dBdx = sym.diff(B, x)

# Compute basis functions and their derivatives
N = 3
psi = {0: [x**((i+1)*(1-x)) for i in range(N+1)]}
psi[1] = [sym.diff(psi[i], x) for psi_i in psi[0]]

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

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

Omega = [0, 1]

from varform1D import solver
u_bar, _ = solver(integrand_lhs, integrand_rhs, psi, Omega,
                   verbose=True, symbolic=True)
u = B + u_bar
print("solution u:", sym.simplify(sym.expand(u)))
```

The printout of `u` reads $-b*x**2/2 + b*x/2 - x + 1$. Note that expanding `u`, before simplifying, is necessary in the present case to get a compact, final expression with `sympy`. Doing `expand` before `simplify` is a common strategy for simplifying expressions in `sympy`. However, a non-expanded `u` might be preferable in other cases - this depends on the problem in question.

The exact solution $u_e(x)$ can be derived by some `sympy` code that closely follows the examples in Section 5.1.2. The idea is to integrate $-u'' = b$ twice and determine the integration constants from the boundary conditions:

```
C1, C2 = sym.symbols('C1 C2')      # integration constants
f1 = sym.integrate(f, x) + C1
```

```

f2 = sym.integrate(f1, x) + C2
# Find C1 and C2 from the boundary conditions u(0)=0, u(1)=1
s = sym.solve([u_e.subs(x,0) - 1, u_e.subs(x,1) - 0], [C1, C2])
# Form the exact solution
u_e = -f2 + s[C1]*x + s[C2]
print('analytical solution:', u_e)
print('error:', sym.simplify(sym.expand(u - u_e)))

```

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

We can play around with the code and test that with $f = Kx^p$, for some constants K and p , the solution is a polynomial of degree $p + 2$, and $N = p + 1$ guarantees that the approximate solution is exact.

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

5.5 Approximations may fail: convection-diffusion

In the previous examples we have obtained reasonable approximations of the continuous solution with several different approaches. In this section we will consider a convection-diffusion equation where many methods will fail. The failure is purely numerical and it is often tied to the resolution. The current example is perhaps the prime example of numerical instabilities in the context of numerical solution algorithms for PDEs. Consider the equation

$$-\epsilon u_{xx} - u_x = 0, \quad \in (0, 1), \quad (5.55)$$

$$u(0) = 1, \quad (5.56)$$

$$u(1) = 0. \quad (5.57)$$

The PDE problem describes a convection-diffusion problem where the convection is modeled by the first order term $-u_x$ and diffusion is described by the second order term $-\epsilon u_{xx}$. In many applications $\epsilon \ll 1$ and the dominating term is $-u_x$. The sign of $-u_x$ is not important, the same problem occurs for u_x . The sign only determine the direction of the convection.

For $\epsilon = 0$, the solution satisfies

$$u(x) - u(1) = \int_1^x (-u_x)(-\mathrm{d}x) = 0,$$

which means that $u(x) = u(1)$. Clearly only the boundary condition at $x = 1$ is required and the solution is constant throughout the domain.

If $0 < \epsilon \ll 1$ such that the term $-u_x$ is dominating, the solution is similar to the solution for $\epsilon = 0$ in the interior. However, the second order term $-\epsilon u_{xx}$ makes the problem a second order problem and two boundary conditions are required, one condition at each side. The boundary condition at $x = 0$ forces the solution to be zero at that point and this creates a sharp gradient close to $x = 0$. For this reason, the problem is called a *singular perturbation problem* as the problem changes fundamentally in the sense that different boundary conditions are required in the limiting case $\epsilon = 0$.

The solution of the above problem is

$$u(x) = \frac{e^{-x/\epsilon} - 1}{e^{-1/\epsilon} - 1}. \quad (5.58)$$

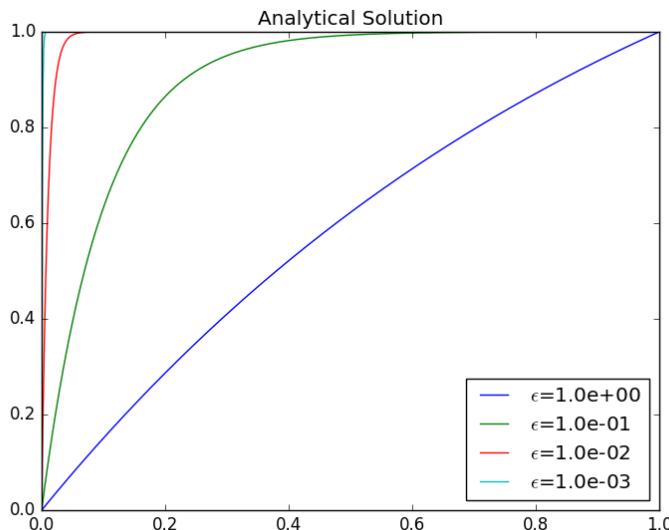


Fig. 5.2 Analytical solution to the convection-diffusion problem for varying ϵ .

The solution is plotted in Figure 5.2 for different values of ϵ . Clearly, as ϵ decrease the exponential function represents a sharper and sharper

gradient. From a physical or engineering point of view, the equation (5.55) represents the simplest problem involving a common phenomenon of boundary layers. Boundary layers are common in all kinds of fluid flow and is a main problem when discretizing such equations. Boundary layers have the characteristics of the solution (5.58), that is; a sharp local exponential gradient. In fluid flow the parameter ϵ is often related to the inverse of the Reynolds number which frequently in engineering is significantly larger than 10^3 as it was here. In these applications the boundary layer is extremely thin and the gradient extremely sharp.

In this chapter we will not embark on the fascinating and complex issue of boundary layer theory but only consider the numerical issues related to this phenomenon. Let us as earlier therefore consider an approximate solution on the following form

$$u(x) = \hat{u}(x) + B(x) = \sum_{j=1}^{N-1} c_j \psi_j(x) + B(x) \quad (5.59)$$

As earlier $\{\psi_j(x)\}_{j=1}^{N-1}$ are zero at the boundary $x = 0$ and $x = 1$ and the boundary conditions are accounted for by the function $B(x)$. Let

$$B(x) = c_0(1 - x) + c_N x. \quad (5.60)$$

Then we fixate $c_0 = 0$ and $c_N = 1$ which makes $B(x) = x$. To determine $\{c_j\}_{j=1}^{N-1}$ we consider the homogeneous Dirichlet problem where we solve for $\hat{u} = u - B$. The homogeneous Dirichlet problem reads

$$-\epsilon \hat{u}_{xx} + \hat{u}_x = 1, \quad \in (0, 1), \quad (5.61)$$

$$\hat{u}(0) = 0, \quad (5.62)$$

$$\hat{u}(1) = 0.$$

The Galerkin formulation of (5.62) is obtained as

$$\int_0^1 (-\epsilon \hat{u}'' + \hat{u}' - 1) \psi_j \, dx.$$

Integration by parts leads to

$$\int_0^1 \epsilon \hat{u}' \psi'_i + \hat{u}' \psi_i - 1 \psi_i \, dx.$$

In other words, we need to solve the linear system $\sum_j A_{i,j} c_j = b_i$ where

$$A_{i,j} = \int_0^1 \epsilon \psi'_j \psi'_i + \psi'_j \psi_i \, dx,$$

$$b_i = \int_0^1 \psi_j \, dx.$$

A sketch of a corresponding code where we also plot the behavior of the solution with respect to different ϵ goes as follows.

```
import matplotlib.pyplot as plt
N = 8
psi = series(x, series_type, N) # Lagrange, Bernstein, sin, ...
eps_values =[1.0, 0.1, 0.01, 0.001]
for eps in eps_values:
    A = sym.zeros(N-1, N-1)
    b = sym.zeros(N-1)

    for i in range(0, N-1):
        integrand = f*psi[i]
        integrand = sym.lambdify([x], integrand, 'mpmath')
        b[i,0] = mpmath.quad(integrand, [Omega[0], Omega[1]])
        for j in range(0, N-1):
            integrand = eps*sym.diff(psi[i], x)*\
                sym.diff(psi[j], x) - sym.diff(psi[i], x)*psi[j]
            integrand = sym.lambdify([x], integrand, 'mpmath')
            A[i,j] = mpmath.quad(integrand, [Omega[0], Omega[1]])

    c = A.LUsolve(b)
    u = sum(c[r,0]*psi[r] for r in range(N-1)) + x

    U = sym.lambdify([x], u, modules='numpy')
    x_ = numpy.arange(Omega[0], Omega[1], 1/((N+1)*100.0))
    U_ = U(x_)
    plt.plot(x_, U_)
```

The numerical solutions for different ϵ is shown in Figure 5.3 and 5.4 for $N = 8$ and $N = 16$, respectively. From these figures we can make two observations. The first observation is that the numerical solution contains non-physical oscillations that grows as ϵ decreases. These oscillations are so strong that for $N = 8$, the numerical solutions do not resemble the true solution at all for ϵ less than 1/10. The true solution is always in the interval $[0, 1]$ while the numerical solution has values larger than 2 for $\epsilon = 1/100$ and larger than 10 for $\epsilon = 1/1000$. The second observation is that the numerical solutions appear to improve as N increases. While the numerical solution is outside the interval $[0, 1]$ for ϵ less than 1/10 the magnitude of the oscillations clearly has decreased. Both Lagrange and Bernstein approximations have similar problems, but the computations using Bernstein polynomials are significantly more efficient and are therefore shown.

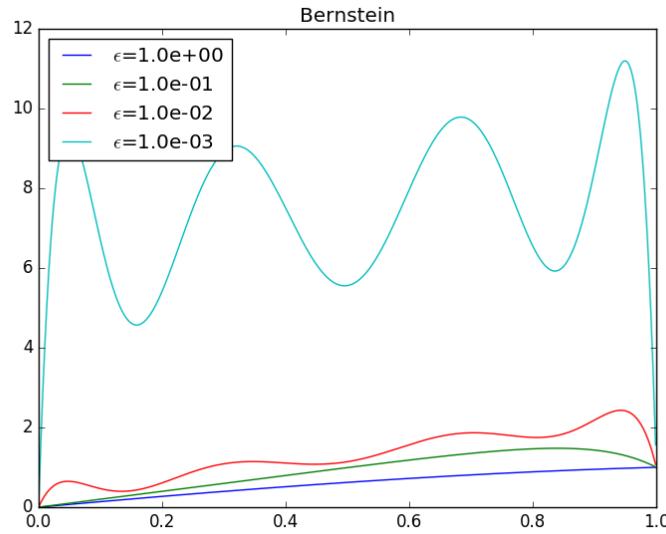


Fig. 5.3 Solution obtained with Galerkin approximation using Bernstein polynomials of order up to 8 for various ϵ .

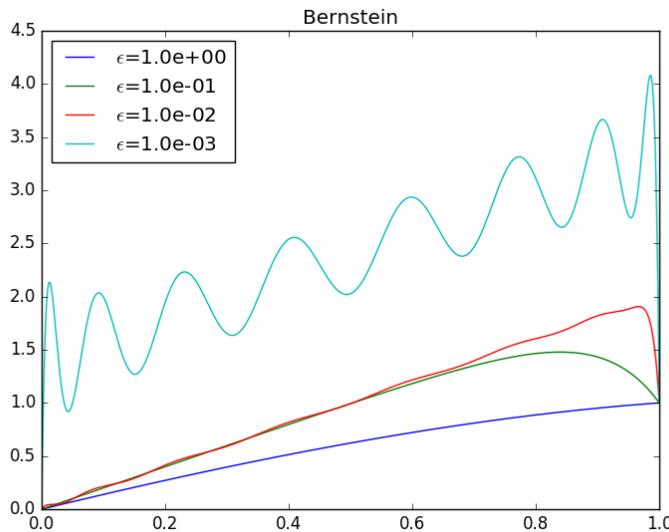


Fig. 5.4 Solution obtained with Galerkin approximation using Bernstein polynomials of order up to 16 for various ϵ .

We will return to this example later and show examples of techniques that can be used to improve the approximation. The complete source code can be found in `conv_diff.py`.

5.6 Exercises

Exercise 5.1: Refactor functions into a more general class

Section 5.1.2 lists three functions for computing the analytical solution of some simple model problems. There is quite some repetitive code, suggesting that the functions can benefit from being refactored into a class hierarchy, where the super class solves $-(a(x)u'(x))' = f(x)$ and where subclasses define the equations for the boundary conditions in a model. Make a method for returning the residual in the differential equation and the boundary conditions when the solution is inserted in these equations. Create a test function that verifies that all three residuals vanish for each of the model problems in Section 5.1.2. Also make a method that returns the solution either as `sympy` expression or as a string in L^AT_EX format. Add a fourth subclass for the problem $-(au')' = f$ with a Robin boundary condition:

$$u(0) = 0, \quad -u'(L) = C(u - D).$$

Demonstrate the use of this subclass for the case $f = 0$ and $a = \sqrt{1+x}$.

Solution. This is an exercise in software engineering. The model-specific information is related to the boundary conditions only. We can then let the super class take care of the differential equation and the solution process, while subclasses provide a method `get_bc` to return the symbolic expressions for the boundary equations.

The super class may be coded as shown below.

```
import sympy as sym
x, L, C, D, c_0, c_1, = sym.symbols('x L C D c_0 c_1')

class TwoPtBoundaryValueProblem(object):
    """
    Solve -(a*u)' = f(x) with boundary conditions
    specified in subclasses (method get_bc).
    a and f must be sympy expressions of x.
    """
    def __init__(self, f, a=1, L=L, C=C, D=D):
        """Default values for L, C, D are symbols."""
        self.f = f
        self.a = a
        self.L = L
        self.C = C
        self.D = D

        # Integrate twice
        u_x = - sym.integrate(f, (x, 0, x)) + c_0
```

```

u = sym.integrate(u_x/a, (x, 0, x)) + c_1
# Set up 2 equations from the 2 boundary conditions and solve
# with respect to the integration constants c_0, c_1
eq = self.get_bc(u)
eq = [sym.simplify(eq_) for eq_ in eq]
print('BC eq:', eq)
self.u = self.apply_bc(eq, u)

def apply_bc(self, eq, u):
    # Solve BC eqs respect to the integration constants
    r = sym.solve(eq, [c_0, c_1])
    # Substitute the integration constants in the solution
    u = u.subs(c_0, r[c_0]).subs(c_1, r[c_1])
    u = sym.simplify(sym.expand(u))
    return u

def get_solution(self, latex=False):
    return sym.latex(self.u, mode='plain') if latex else self.u

def get_residuals(self):
    """Return the residuals in the equation and BCs."""
    R_eq = sym.diff(sym.diff(self.u, x)*self.a, x) + self.f
    R_0, R_L = self.get_bc(self.u)
    residuals = [sym.simplify(R) for R in (R_eq, R_0, R_L)]
    return residuals

def get_bc(self, u):
    raise NotImplementedError(
        'class %s has not implemented get_bc' %
        self.__class__.__name__)

```

The various subclasses deal with the boundary conditions of the various model problems:

```

class Model1(TwoPtBoundaryValueProblem):
    """u(0)=0, u(L)=D."""
    def get_bc(self, u):
        return [u.subs(x, 0)-0,                      # x=0 condition
                u.subs(x, self.L) - self.D]          # x=L condition

class Model2(TwoPtBoundaryValueProblem):
    """u'(0)=C, u(L)=D."""
    def get_bc(self, u):
        return [sym.diff(u,x).subs(x, 0) - self.C, # x=0 cond.
                u.subs(x, self.L) - self.D]          # x=L cond.

class Model3(TwoPtBoundaryValueProblem):
    """u(0)=C, u(L)=D."""
    def get_bc(self, u):
        return [u.subs(x, 0) - self.C,
                u.subs(x, self.L) - self.D]

```

A suitable test function gets quite compact:

```
def test_TwoPtBoundaryValueProblem():
    f = 2
    model = Model1(f)
    print('Model 1, u:', model.get_solution())
    for R in model.get_residuals():
        assert R == 0

    f = x
    model = Model2(f)
    print('Model 2, u:', model.get_solution())
    for R in model.get_residuals():
        assert R == 0

    f = 0
    a = 1 + x**2
    model = Model3(f, a=a)
    print('Model 3, u:', model.get_solution())
    for R in model.get_residuals():
        assert R == 0
```

The fourth model is just about defining the boundary conditions as equations:

```
class Model4(TwoPtBoundaryValueProblem):
    """u(0)=0, -u'(L)=C*(u-D)."""
    def get_bc(self, u):
        return [u.subs(x, 0) - 0,
                -sym.diff(u, x).subs(x, self.L) -
                self.C*(u.subs(x, self.L) - self.D)]
```

A demo function goes like

```
def demo_Model4():
    f = 0
    model = Model4(f, a=sym.sqrt(1+x))
    print('Model 4, u:', model.get_solution())
```

The printout shows that the solution is

$$u(x) = \frac{2CD\sqrt{1+L}(\sqrt{1+x} - 1)}{2C\sqrt{1+L} + 2C + 1}.$$

Filename: `uxx_f_sympy_class.`

Exercise 5.2: Compute the deflection of a cable with sine functions

A hanging cable of length L with significant tension T has a deflection $w(x)$ governed by

$$Tw''(x) = \ell(x),$$

where $\ell(x)$ the vertical load per unit length. The cable is fixed at $x = 0$ and $x = L$ so the boundary conditions become $w(0) = w(L) = 0$. The deflection w is positive upwards, and ℓ is positive when it acts downwards.

If we assume a constant load $\ell(x) = \text{const}$, the solution is expected to be symmetric around $x = L/2$. For a function $w(x)$ that is symmetric around some point x_0 , it means that $w(x_0 - h) = w(x_0 + h)$, and then $w'(x_0) = \lim_{h \rightarrow 0} (w(x_0 + h) - w(x_0 - h))/(2h) = 0$. We can therefore utilize symmetry to halve the domain. We then seek $w(x)$ in $[0, L/2]$ with boundary conditions $w(0) = 0$ and $w'(L/2) = 0$.

The problem can be scaled by introducing dimensionless variables,

$$\bar{x} = \frac{x}{L/2}, \quad \bar{u} = \frac{w}{w_c},$$

where w_c is a characteristic size of w . Inserted in the problem for w ,

$$\frac{4Tw_c}{L^2} \frac{d^2\bar{u}}{d\bar{x}^2} = \ell \quad (= \text{const}).$$

A desire is to have u and its derivatives about unity, so choosing w_c such that $|d^2\bar{u}/d\bar{x}^2| = 1$ is an idea. Then $w_c = \frac{1}{4}\ell L^2/T$, and the problem for the scaled vertical deflection u becomes

$$u'' = 1, \quad x \in (0, 1), \quad u(0) = 0, \quad u'(1) = 0.$$

Observe that there are no physical parameters in this scaled problem. From now on we have for convenience renamed x to be the scaled quantity \bar{x} .

a) Find the exact solution for the deflection u .

Solution. Exercise 5.1 or Section 5.1.2 features tools for finding the analytical solution of this differential equation. The present model problem is close to model 2 in Section 5.1.2. We can modify the `model2` function:

```
def model():
    """Solve u'' = -1, u(0)=0, u'(1)=0."""
    import sympy as sym
    x, c_0, c_1, = sym.symbols('x c_0 c_1')
    u_x = sym.integrate(1, (x, 0, x)) + c_0
    u = sym.integrate(u_x, (x, 0, x)) + c_1
    r = sym.solve([u.subs(x,0) - 0,
                  sym.diff(u,x).subs(x, 1) - 0],
                  [c_0, c_1])
    u = u.subs(c_0, r[c_0]).subs(c_1, r[c_1])
    u = sym.simplify(sym.expand(u))
```

```
return u
```

The solution becomes

$$u(x) = \frac{1}{2}x(x - 2).$$

Plotting $u(x)$ shows that $|u| \in [0, \frac{1}{2}]$ which is compatible with the aim of the scaling, i.e., to have u of size *about* unity (at least not very small or very large).

b) A possible function space is spanned by $\psi_i = \sin((2i + 1)\pi x/2)$, $i = 0, \dots, N$. These functions fulfill the necessary condition $\psi_i(0) = 0$, but they also fulfill $\psi'_i(1) = 0$ such that both boundary conditions are fulfilled by the expansion $u = \sum_j c_j \varphi_j$.

Use a Galerkin and a least squares method to find the coefficients c_j in $u(x) = \sum_j c_j \psi_j$. Find how fast the coefficients decrease in magnitude by looking at c_j/c_{j-1} . Find the error in the maximum deflection at $x = 1$ when only one basis function is used ($N = 0$).

Hint. In this case, where the basis functions and their derivatives are orthogonal, it is easiest to set up the calculations by hand and use `sympy` to help out with the integrals.

Solution. With $u = \sum_{j=0}^N c_j \psi_j(x)$ the residual becomes

$$R = 1 - u'' = 1 + \sum_{j=0}^N c_j \psi_j''(x) = 1 + \sum_{j=0}^N c_j (2j+1)^2 \frac{\pi^2}{4} \sin((2j+1) \frac{\pi x}{2}).$$

Least squares method. The minimization of $\int_0^1 R^2 dx$ leads to the equations

$$(R, \frac{\partial R}{\partial c_i}) = 0, \quad i = 0, \dots, N.$$

We find that

$$\frac{\partial R}{\partial c_i} = (2i+1)^2 \frac{\pi^2}{4} \sin((2i+1) \frac{\pi x}{2}),$$

so the governing equations become

$$(1 + \sum_{j=0}^N c_j (2j+1)^2 \frac{\pi^2}{4} \sin((2j+1) \frac{\pi x}{2}), (2i+1)^2 \frac{\pi^2}{4} \sin((2i+1) \frac{\pi x}{2})) = 0.$$

By linearity of the inner product (or integral) this expression can be reordered to

$$\sum_{j=0}^N c_j \left((2j+1)^2 \frac{\pi^2}{4} \sin((2j+1)\frac{\pi x}{2}), (2i+1)^2 \frac{\pi^2}{4} \sin((2i+1)\frac{\pi x}{2}) \right) = \\ - \left(1, (2i+1)^2 \frac{\pi^2}{4} \sin((2i+1)\frac{\pi x}{2}) \right),$$

which is nothing but a linear system

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

with

$$A_{i,j} = (2j+1)^4 \frac{\pi^4}{16} \int_0^1 \sin((2j+1)\frac{\pi x}{2}) \sin((2i+1)\frac{\pi x}{2}) dx, \\ b_i = -(2i+1)^2 \frac{\pi^2}{4} \int_0^1 \sin((2i+1)\frac{\pi x}{2}) dx$$

Orthogonality of the sine functions $\sin(k\pi x/2)$ on $[0, 1]$ for integer k implies that $A_{i,j} = 0$ for $i \neq j$, and $A_{i,i}$ can be computed by `sympy`:

```
>>> from sympy import *
>>> i = symbols('i', integer=True)
>>> x = symbols('x', real=True)
>>> integrate(sin(i*pi*x/2)**2, (x, 0, 1))
1/2
```

Therefore,

$$A_{i,j} = \begin{cases} 0, & i \neq j \\ \frac{1}{2}(2i+1)^4 \frac{\pi^4}{16}, & i = j \end{cases}$$

The right-hand side can also be computed by `sympy`:

```
>>> integrate(sin((2*i+1)*pi*x/2), (x, 0, 1))
2/(pi*(2*i+1))
```

One should always be skeptical to symbolic software and integration of periodic functions like the sine and cosine since the answers can be too simplistic (see subexercise d!). A general test is to perform numerical integration with lots of sampling points to (partially) verify the symbolic formula. Here is an application of the midpoint rule:

```

def midpoint_rule(f, M=100000):
    """Integrate f(x) over [0,1] using M intervals."""
    from numpy import sum, linspace
    dx = 1.0/M                                # interval length
    x = linspace(dx/2, 1-dx/2, M)    # integration points
    return dx*sum(f(x))

def check_integral_b():
    from numpy import pi, sin
    for i in range(12):
        exact = 2/(pi*(2*i+1))
        numerical = midpoint_rule(
            f=lambda x: sin((2*i+1)*pi*x/2))
        print(i, abs(exact - numerical))

```

The output shows that the difference between numerical and exact integration is about 10^{-11} , which is “small” (and gets smaller by just increasing M). This result brings evidence that the `sympy` answer is correct. Alternatively, in this simple case, we can easily calculate the anti-derivative. It goes like

$$-\frac{2}{\pi(2i+1)} \cos((2k+1)\frac{\pi x}{2}),$$

and for $x = 1$ we get $\cos \frac{\pi}{2}$, $\cos 3\frac{\pi}{2}$, $\cos 5\frac{\pi}{2}$, and so on, which all evaluates to zero, and since the cosine is 1 for $x = 0$, the formula found by `sympy` is correct.

We then get

$$b_i = -(2i+1)^2 \frac{\pi^2}{4} \frac{2}{\pi(2i+1)} = -\frac{1}{2}(2i+1)\pi,$$

and consequently,

$$c_i = \frac{b_i}{A_{i,i}} = -\frac{\frac{1}{2}(2i+1)\pi}{\frac{1}{2}(2i+1)^4} \frac{\pi^4}{16} = -\frac{16}{\pi^3(2i+1)^3}.$$

Galerkin's method. The Galerkin method applied to this problem starts with

$$(u'', v) = (1, v) \quad \forall v \in V,$$

and the requirement that $v(0) = 0$ since $u(0) = 0$. Integration by parts and using $u'(1) = 0$ and $v(0) = 0$ makes the boundary term vanish, and the variational form becomes

$$(u', v') = -(1, v) \quad \forall v \in V.$$

Inserting $u = \sum_{j=0}^N c_j \psi_j(x)$ and $v = \psi_i$ leads to

$$\sum_{j=0}^N (\psi'_j, \psi'_i) c_j = (1, \psi_i), \quad i = 0, \dots, N.$$

With $\psi_i = \sin((2i+1)\frac{\pi x}{2})$ the matrix entries become

$$A_{i,j} = (2i+1)(2j+1) \frac{\pi^2}{4} \int_0^1 \cos((2i+1)\frac{\pi x}{2}) \cos((2j+1)\frac{\pi x}{2}) dx.$$

Orthogonality of the cosine functions implies $A_{i,j} = 0$ for $i \neq j$, and $A_{i,i}$ is computed by integrating the square of the cosine function,

```
>>> integrate(cos((k+1)*pi*x/2)**2, (x, 0, 1))
1/2
```

Now,

$$A_{i,i} = (2i+1)^2 \frac{\pi^2}{4} \frac{1}{2} = \frac{1}{8} (2i+1)^2 \pi^2.$$

The right-hand side has almost the same integral as in the least squares case,

$$b_i = - \int_0^1 \sin((2i+1)\frac{\pi x}{2}) dx = -\frac{2}{\pi(2i+1)}.$$

Consequently,

$$c_i = \frac{b_i}{A_{i,i}} = -\frac{16}{\pi^3 (2i+1)^3},$$

which is the same result as we obtained in the least squares method.

Decay of coefficients. The coefficients decay,

$$\frac{c_i}{c_{i+1}} = \left(\frac{2i+3}{2i+1}\right)^3 > 0.$$

The decay is most pronounced for the first terms:

```
>>> for i in range(10):
...     print(float(2*i+3)/(2*i+1))**3
...
27.0
4.62962962963
2.744
2.12536443149
1.82578875171
```

```
1.65063861758
1.53618570778
1.4557037037
1.39609200081
1.35019682169
```

Error in one-term solution. Keeping just one term ($N = 0$) means that

$$u(x) = -\frac{16}{\pi^3} \sin\left(\frac{\pi x}{2}\right).$$

The maximum deflection at $x = 1$ becomes $-16\pi^{-3} = -0.5160$, to be compared with the exact value $-\frac{1}{2}$. The error is 3.2 percent.

c) Visualize the solutions in b) for $N = 0, 1, 20$.

Solution. First we need a function to compute the approximate u in this case:

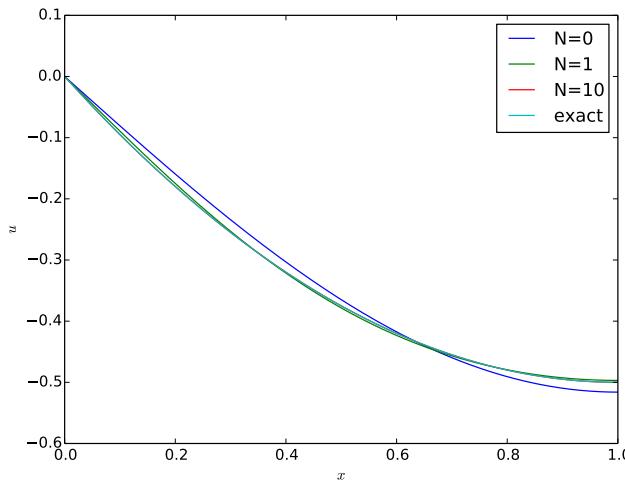
```
def sine_sum(x, N):
    s = 0
    from numpy import pi, sin, zeros
    u = [] # u[k] is the sum i=0,...,k
    k = 0
    for i in range(N+1):
        s += - 16.0/((2*i+1)**3*pi**3)*sin((2*i+1)*pi*x/2)
        u.append(s.copy()) # important with copy!
    return u
```

Note the need to append `s.copy()`: doing just `u.append(s)` will make, e.g., `u[0]` a reference to `s`, which at the end of the loop is an array corresponding to the maximum i value.

We also need a function that can create an appropriate plot:

```
def plot_sine_sum():
    from numpy import linspace
    x = linspace(0, 1, 501) # coordinates for plot
    u = sine_sum(x, N=10)
    u_e = 0.5*x*(x-2)
    N_values = 0, 1, 10
    for k in N_values:
        plt.plot(x, u[k])
    plt.plot(x, u_e)
    plt.legend(['N=%d' % k for k in N_values] + ['exact'],
              loc='upper right')
    plt.xlabel('$x$'); plt.ylabel('$u$')
    plt.savefig('tmpc.png'); plt.savefig('tmpc.pdf')
```

The plot shows that the solution for $N = 0$ has a slight deviation from the exact curve, but even $N = 1$ catches up visually with the exact solution (!).



d) The functions in b) were selected such that they fulfill the condition $\psi'(1) = 0$. However, in the Galerkin method, where we integrate by parts, the condition $u'(1) = 0$ is incorporated in the variational form. This leads to the idea of just choosing a simpler basis, namely “all” sine functions $\psi_i = \sin((i+1)\frac{\pi x}{2})$. Will the method adjust the coefficient such that the additional functions compared with those in b) get vanishing coefficients? Or will the additional basis functions improve the solution? Use Galerkin’s method.

Solution. According to the calculations in b), the Galerkin method, with $\psi_i = \sin((i+1)\frac{\pi x}{2})$, leads to the almost the same matrix entries on the diagonal:

$$\begin{aligned} A_{i,i} &= (i+1)(j+1) \frac{\pi^2}{4} \int_0^1 \cos((i+1)\frac{\pi x}{2}) \cos((j+1)\frac{\pi x}{2}) dx \\ &= (i+1)^2 \frac{\pi^2}{4} \frac{1}{2} = \frac{1}{8}(i+1)^2 \pi^2. \end{aligned}$$

The right-hand side becomes (as before)

$$b_i = - \int_0^1 \sin((i+1)\frac{\pi x}{2}) dx = -\frac{2}{\pi(i+1)}.$$

We may use `sympy` to integrate,

```
>>> integrate(sin((i+1)*pi*x/2), (x, 0, 1))
2/(pi*(i+1))
```

As noted in b), let us be a bit skeptical to this answer and check it. A quick check with numerical integration,

```
def check_integral_d_sympy_answer():
    from numpy import pi, sin
    for i in range(12):
        exact = 2/(pi*(i+1))
        numerical = midpoint_rule(
            f=lambda x: sin((i+1)*pi*x/2))
        print(i, abs(exact - numerical))
```

gives the output

```
0 6.54487575247e-12
1 0.31830988621
2 1.96350713466e-11
3 0.159154943092
4 3.27249061183e-11
5 0.106103295473
6 4.58150045679e-11
7 0.0795774715459
8 5.89047144395e-11
9 0.0636619773677
10 7.19949447281e-11
11 0.0530516476973
```

It is clear that for i odd, there are significant differences between the `sympy` answer and the midpoint rule with high resolution!

We therefore need to do hand calculations to investigate this problem further. The anti-derivative is very easy to realize in this case:

$$\begin{aligned} \int_0^1 \sin((i+1)\pi x/2) dx &= -\frac{2}{\pi(i+1)} \left(\cos((i+1)\frac{\pi}{2}) - \cos(0) \right) \\ &= \frac{2}{\pi(i+1)} \left(1 - \cos((i+1)\frac{\pi}{2}) \right). \end{aligned}$$

The value of the cosine expression depends on i , and the first values are

$i = 0$	$i = 1$	$i = 2$	$i = 3$
0	-1	0	1

This pattern repeats and is the same for four consecutive values of i . Hence, the integral is $2/(\pi(i+1))$ for even i ($i = 2k$ for integer k , or equivalently: when $i \bmod 2 = 0$). For $i = 4k + 1$, or equivalently: when $(i-1) \bmod 4 = 0$, the integral is $4/(\pi(4k+1))$, while for $i = 4k + 3$, the

integral vanishes. This is a more complicated answer than what `sympy` provides!

We can check our new answers against numerical integration:

```
def check_integral_d():
    from numpy import pi, sin
    for i in range(24):
        if i % 2 == 0:
            exact = 2/(pi*(i+1))
        elif (i-1) % 4 == 0:
            exact = 2*2/(pi*(i+1))
        else:
            exact = 0
        numerical = midpoint_rule(
            f=lambda x: sin((i+1)*pi*x/2))
        print(i, abs(exact - numerical))
```

The output now is around 10^{-10} and we take that as a sign that our exact results are reliable.

Carefully check symbolic computations!

The example above shows how `sympy` can fail. Wolfram Alpha does a better job: writing `integrate sin(k*x*pi/2) from 0 to 1` (use `k` instead of `i` since the latter is the imaginary unit) returns the result $4 \sin^2(\pi k/4)/(\pi k)$, which coincides with our result.

There are three general techniques to verify a symbolic computation:

- Use alternative software like Wolfram Alpha for comparison
- Check that the result satisfies the problem to be solved
- Make a high-resolution numerical approximation and compare

(The second technique is not so applicable here, since we work with a definite integral, but one could compute the indefinite integral instead, which is done correctly by `sympy`, and discuss values for $x = 1$.)

The final result for c_i is now

$$c_i = \frac{b_i}{A_{i,i}} = \begin{cases} -\frac{16}{\pi^3(i+1)^3}, & i \text{ even, or } i \bmod 2 = 0 \\ -\frac{32}{\pi^3(i+1)^3}, & (i-1) \bmod 4 = 0, \\ 0, & (i+1) \bmod 4 = 0 \end{cases}$$

We recognize that for i even, say $i = 2k$ for integer k , we have exactly the same result as in b):

$$-\sum_k \frac{16}{\pi^3(2k+1)^3} \sin((2k+1)x\frac{\pi x}{2}),$$

but we get an additional set of terms for $i = 4k + 1$,

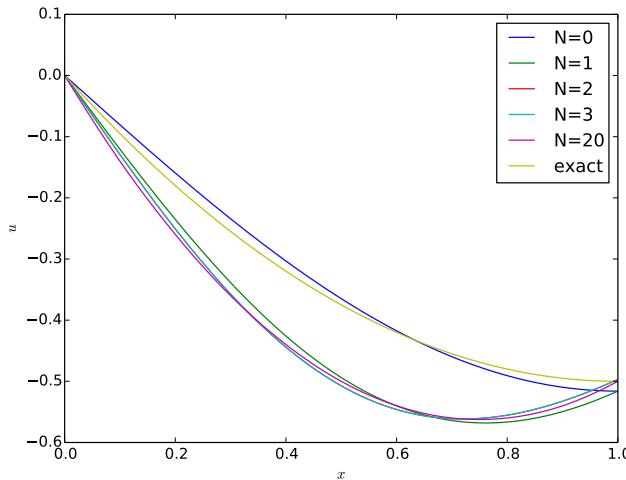
$$-\sum_k \frac{32}{\pi^3(i+1)^3} \sin((4k+1)x\frac{\pi x}{2}). \quad (5.63)$$

We can modify the software from c) to compute the approximate u with the present set of basis functions and coefficients:

```
def sine_sum_d(x, N):
    s = 0
    from numpy import pi, sin, zeros
    u = [] # u[k] is the sum i=0,...,k
    k = 0
    for i in range(N+1):
        if i % 2 == 0:      # even i
            s += - 16.0/((i+1)**3*pi**3)*sin((i+1)*pi*x/2)
        elif (i-1) % 4 == 0: # 1, 5, 9, 13, 17
            s += - 2*16.0/((i+1)**3*pi**3)*sin((i+1)*pi*x/2)
        else:
            s += 0
        u.append(s.copy())
    return u

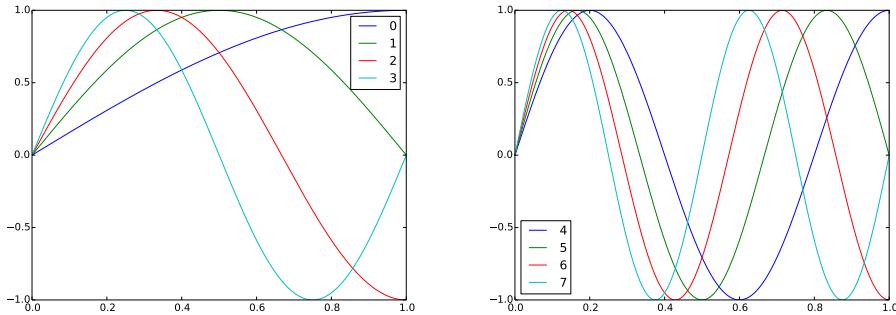
def plot_sine_sum_d():
    from numpy import linspace
    x = linspace(0, 1, 501) # coordinates for plot
    u = sine_sum_d(x, N=20)
    u_e = 0.5*x*(x-2)
    N_values = 0, 1, 2, 3, 20
    for k in N_values:
        plt.plot(x, u[k])
    plt.plot(x, u_e)
    plt.legend(['N=%d' % k for k in N_values] + ['exact'],
               loc='upper right')
    plt.xlabel('$x$'); plt.ylabel('$u$')
    #plt.axis([0.9, 1, -0.52, -0.49])
    plt.savefig('tmpd.png'); plt.savefig('tmpd.pdf')
```

The approximations for $N = 0, 1, 3, 20$ appear below.



While the approximation for $N = 0$ coincides with the one in b), we see that $N = 1$ and higher values of N lead to a clearly wrong curve. This strange feature has to be investigated!

Let us start by plotting the basis functions for $i = 0, 1, \dots, 7$:



We observe from the figure that all the basis functions corresponding to even i are symmetric around $x = 1$, which is an important property of the solution. The functions for odd i are anti-symmetric. However, for $i = 3, 7, 11, \dots$ the basis function has an integer number of periods on $[0, 1]$ so the integral becomes zero, $c_i = 0$, and consequently there is no effect from these functions. The functions corresponding to $i = 1, 5, 9, 13, \dots$ are anti-symmetric around $x = 1$ with nonzero coefficients. The derivative of an anti-symmetric function at the point of anti-symmetry is unity in

size. Since the derivatives of all the basis functions corresponding to even i vanish at $x = 1$, the extra terms ($i = 1, 5, 9, 13, \dots$) in (5.63) have a nonzero derivative, resulting in $u'(1) \neq 0$. That is, these terms destroy the solution!

But we computed c_i by a Galerkin method, which is equivalent to a least squares method, which gives us the “best” approximation possible? That is true, but it is the best approximation in the chosen space V . The problem is that we have populated (or rather polluted) the space V with some basis functions that have a wrong mathematical property: they are anti-symmetric around $x = 1$.

e) Now we drop the symmetry condition at $x = 1$ and extend the domain to $[0, 2]$ such that it covers the entire (scaled) physical cable. The problem now reads

$$u'' = 1, \quad x \in (0, 2), \quad u(0) = u(2) = 0.$$

This time we need basis functions that are zero at $x = 0$ and $x = 2$. The set $\sin((i+1)\frac{\pi x}{2})$ from d) is a candidate since they vanish at $x = 2$ for any i . Compute the approximation in this case. Why is this approximation without the problem that this set of basis functions introduced in d)?

Solution. The formulas are almost the same as in d), only the integration domain is different. Since the sine functions are orthogonal on $[0, 1]$, they are also orthogonal on $[0, 2]$. Because

```
>>> integrate(cos((i+1)*pi*x/2)**2, (x, 0, 2))
1
```

we get (in Galerkin’s method)

$$\begin{aligned} A_{i,i} &= (i+1)(j+1) \frac{\pi^2}{4} \int_0^2 \cos((i+1)\frac{\pi x}{2}) \cos((i+1)\frac{\pi x}{2}) dx \\ &= (i+1)^2 \frac{\pi^2}{4}. \end{aligned}$$

and

$$b_i = - \int_0^2 \sin((i+1)\frac{\pi x}{2}) dx = \frac{2}{\pi(i+1)} (\cos((i+1)\pi) - 1).$$

We have that $\cos((i+1)\pi) = -1$ for i even and $\cos((i+1)\pi) = 1$ for i odd. That is,

$$b_i = \begin{cases} -\frac{4}{\pi(i+1)}, & i \text{ even} \\ 0, & i \text{ odd} \end{cases}$$

The coefficients become

$$c_i = \frac{b_i}{A_{i,i}} = \begin{cases} -\frac{16}{\pi^3(2i+1)^3}, & i \text{ even} \\ 0, & i \text{ odd} \end{cases}$$

Introducing $i = 2k$ and then switching from k to i as summation index gives $c_i = -\frac{16}{\pi^3(2i+1)^3}$ and

$$u(x) = -\sum_{i=0}^N \frac{16}{\pi^3(2i+1)^3} \sin((i+1)\frac{\pi x}{2}),$$

which is the same expansion as in b).

The reason why the basis functions $\psi_i = \sin((i+1)\frac{\pi x}{2})$ work well in this case is that the problematic functions for $i = 1, 5, \dots$ in d) now live on $[0, 2]$ instead of $[0, 1]$. On $[0, 2]$ these functions have an integer number of periods such that the integral from 0 to 2 becomes zero. These basis functions are therefore excluded from the expansion since their coefficients vanish. The lesson learned is that two equivalent boundary value problems may make different demands to the basis functions.

Filename: `cable_sin`.

Exercise 5.3: Compute the deflection of a cable with power functions

a) Repeat Exercise 5.2 b), but work with the space

$$V = \text{span}\{x, x^2, x^3, x^4, \dots\}.$$

Choose the dimension of V to be 4 and observe that the exact solution is recovered by the Galerkin method.

Hint. Use the `solver` function from `varform1D.py`.

Solution. The Galerkin formulation of $u'' = 1$, $u(0) = 0$, $u'(1) = 0$, reads

$$(u', v') = -(1, v) \quad \forall v \in V,$$

and the linear system becomes

$$\sum_{j=1}^N (\psi'_i, \psi'_j) c_j = -(1, \psi_i), \quad i = 0, 1, \dots, N.$$

The `varform1D.solver` function needs a function specifying the integrands on the left- and right-hand sides of the variational formulation. Moreover, we must compute a dictionary of ψ_i and ψ'_i . The appropriate code becomes

```
from varform1D import solver
import sympy as sym
x, b = sym.symbols('x b')
f = 1

# Compute basis functions and their derivatives
N = 4
psi = {0: [x**i for i in range(N+1)]}
psi[1] = [sym.diff(psi[i], x) for psi_i in psi[0]]

# Galerkin

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

def integrand_rhs(psi, i):
    return -f*psi[0][i]

Omega = [0, 1]

u, c = solver(integrand_lhs, integrand_rhs, psi, Omega,
               verbose=True, symbolic=True)
print('Galerkin solution u:', sym.simplify(sym.expand(u)))
```

Running this code gives the output

```
solution u: x*(x - 2)/2
```

which coincides with the exact solution ($c_3 = c_4 = 0$).

b) What happens if we use a least squares method for this problem with the basis in a)?

Solution. The least squares formulation leads to

$$(R, \frac{\partial R}{\partial c_i}) = 0, \quad i = 0, \dots, N,$$

with

$$R = 1 - u'' = 1 - \sum_j c_j \psi''_j.$$

We have

$$\frac{\partial R}{\partial c_i} = \psi''_i,$$

leading to the equations

$$(1 + \sum_j c_j \psi''_j, \psi''_i), \quad i = 0, \dots, N,$$

which is a linear system

$$\sum_{j=0}^N (\psi''_j, \psi''_i) = (-1, \psi''_i), \quad i = 0, \dots, N.$$

The fundamental problem with the basis in a) is that $\psi''_0 = 0$, so if power functions of x are wanted, we need to work with the basis $V = \text{span}\{x^2, x^3, \dots\}$. If we do so, we can easily modify the code from a),

```
# Least squares
psi = [0: [x**(i+2) for i in range(N+1)]]
psi[1] = [sym.diff(psi_i, x) for psi_i in psi[0]]
psi[2] = [sym.diff(psi_i, x) for psi_i in psi[1]]

def integrand_lhs(psi, i, j):
    return psi[2][i]*psi[2][j]

def integrand_rhs(psi, i):
    return -f*psi[2][i]

Omega = [0, 1]

u, c = solver(integrand_lhs, integrand_rhs, psi, Omega,
               verbose=True, symbolic=True)
print('solution u:', sym.simplify(sym.expand(u)))
```

The result is $u = -\frac{1}{2}x^2$. This function does not obey $u'(1) = 0$ and is completely wrong. In this least squares method we cannot access the basis function x , which is needed in the exact solution, and we have no means to obtain $u'(1) = 0$.

Remark. There is a modification of the least squares method that can be applied here. The problem $u'' = 1$ must be rewritten as a system of two equations, $u'_1 = u_2$, $u'_2 = 1$. We expand $u_1 = \sum_{j=0}^N c_j \psi_j$ and $u_2 = \sum_{j=0}^N d_j \psi_j$. The residuals in both equations are added, squared, and differentiated with respect to c_i and d_i , $i = 0, \dots, N$. The result is a coupled equation system for the c_i and d_i coefficients.

Filename: `cable_xn`.

Exercise 5.4: Check integration by parts

Consider the Galerkin method for the problem involving u in Exercise 5.2. Show that the formulas for c_j are independent of whether we perform integration by parts or not.

Solution. The Galerkin method is

$$(u'', v) = (1, v) \quad \forall v \in V,$$

and with the choice of V we get

$$A_{i,j} = -(i+1)^2 \pi^2 \int_0^1 \sin^2((i+1)\frac{\pi x}{2}) dx,$$
$$b_i = \int_0^1 \sin((i+1)\frac{\pi x}{2}) dx$$

From Exercise 5.2 we realize that the integrals are the same as in the least squares method, and those results were identical to those of the Galerkin method with integration by parts.

Filename: `cable_integr_by_parts`.

We shall now take the ideas from previous chapters and put together such that we can solve PDEs using the flexible finite element basis functions. This is quite a machinery with many details, but the chapter is mostly an assembly of concepts and details we have already met.

6.1 Computing with finite elements

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

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

with variational formulation

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

Any $v \in V$ must obey $v(0) = v(L) = 0$ because of the Dirichlet conditions on u . The variational formulation is derived in Section 5.1.11.

6.1.1 Finite element mesh and basis functions

We introduce a finite element mesh with N_e cells, all with length h , and number the cells from left to right. Choosing P1 elements, there are two nodes per cell, and the coordinates of the nodes become

$$x_i = ih, \quad h = L/N_e, \quad i = 0, \dots, N_n - 1 = N_e.$$

Any node i is associated with a finite element basis function $\varphi_i(x)$. When approximating a given function f by a finite element function u , we expand u using finite element basis functions associated with *all* nodes in the mesh. The parameter N , which counts the unknowns from 0 to N , is then equal to $N_n - 1$ such that the total number of unknowns, $N + 1$, is the total number of nodes. However, when solving differential equations we will often have $N < N_n - 1$ because of Dirichlet boundary conditions. The reason is simple: we know what u are at some (here two) nodes, and the number of unknown parameters is naturally reduced.

In our case with homogeneous Dirichlet boundary conditions, we do not need any boundary function $B(x)$, so we can work with the expansion

$$u(x) = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x). \quad (6.1)$$

Because of the boundary conditions, we must demand $\psi_i(0) = \psi_i(L) = 0$, $i \in \mathcal{I}_s$. When ψ_i for all $i = 0, \dots, N$ is to be selected among the finite element basis functions φ_j , $j = 0, \dots, N_n - 1$, we have to avoid using φ_j functions that do not vanish at $x_0 = 0$ and $x_{N_n-1} = L$. However, all φ_j vanish at these two nodes for $j = 1, \dots, N_n - 2$. Only basis functions associated with the end nodes, φ_0 and φ_{N_n-1} , violate the boundary conditions of our differential equation. Therefore, we select the basis functions φ_i to be the set of finite element basis functions associated with all the interior nodes in the mesh:

$$\psi_i = \varphi_{i+1}, \quad i = 0, \dots, N.$$

The i index runs over all the unknowns c_i in the expansion for u , and in this case $N = N_n - 3$.

In the general case, and in particular on domains in higher dimensions, the nodes are not necessarily numbered from left to right, so we introduce a mapping from the node numbering, or more precisely the degree of freedom numbering, to the numbering of the unknowns in the final equation system. These unknowns take on the numbers $0, \dots, N$. Unknown number j in the linear system corresponds to degree of freedom number $\nu(j)$, $j \in \mathcal{I}_s$. We can then write

$$\psi_i = \varphi_{\nu(i)}, \quad i = 0, \dots, N.$$

With a regular numbering as in the present example, $\nu(j) = j + 1$, $j = 0, \dots, N = N_n - 3$.

6.1.2 Computation in the global physical domain

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

The entries in the coefficient matrix and right-hand side are

$$A_{i,j} = \int_0^L \psi'_i(x) \psi'_j(x) dx, \quad b_i = \int_0^L 2\psi_i(x) dx, \quad i, j \in \mathcal{I}_s.$$

Expressed in terms of finite element basis functions φ_i we get the alternative expressions

$$A_{i,j} = \int_0^L \varphi'_{i+1}(x) \varphi'_{j+1}(x) dx, \quad b_i = \int_0^L 2\varphi_{i+1}(x) dx, \quad i, j \in \mathcal{I}_s.$$

For the following calculations the subscripts on the finite element basis functions are more conveniently written as i and j instead of $i + 1$ and $j + 1$, so our notation becomes

$$A_{i-1,j-1} = \int_0^L \varphi'_i(x) \varphi'_j(x) dx, \quad b_{i-1} = \int_0^L 2\varphi_i(x) dx,$$

where the i and j indices run as $i, j = 1, \dots, N + 1 = N_n - 2$.

The $\varphi_i(x)$ function is a hat function with peak at $x = x_i$ and a linear variation in $[x_{i-1}, x_i]$ and $[x_i, x_{i+1}]$. The derivative is $1/h$ to the left of x_i and $-1/h$ to the right, or more formally,

$$\varphi'_i(x) = \begin{cases} 0, & x < x_{i-1}, \\ h^{-1}, & x_{i-1} \leq x < x_i, \\ -h^{-1}, & x_i \leq x < x_{i+1}, \\ 0, & x \geq x_{i+1} \end{cases} \quad (6.2)$$

Figure 6.1 shows $\varphi'_2(x)$ and $\varphi'_3(x)$.

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

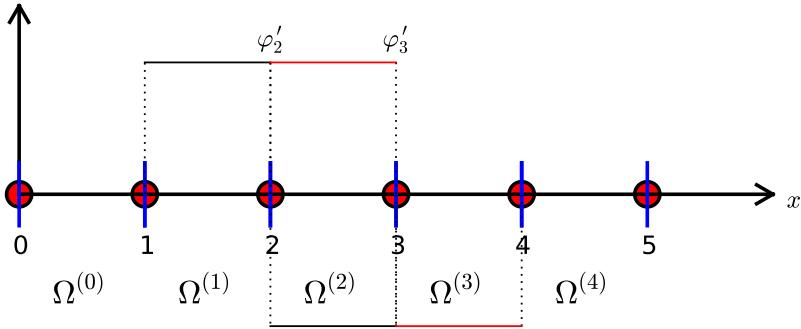


Fig. 6.1 Illustration of the derivative of piecewise linear basis functions associated with nodes in cell 2.

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

for $i = 1, \dots, N + 1$, but for $i = 1$, $A_{i-1,i-2}$ is not defined, and for $i = N + 1$, $A_{i-1,i}$ is not defined.

From Figure 6.1, we see that $\varphi_{i-1}'(x)$ and $\varphi_i'(x)$ have overlap of one cell $\Omega^{(i-1)} = [x_{i-1}, x_i]$ and that their product then is $-1/h^2$. The integrand is constant and therefore $A_{i-1,i-2} = -h^{-2}h = -h^{-1}$. A similar reasoning can be applied to $A_{i-1,i}$, which also becomes $-h^{-1}$. The integral of $\varphi_i'(x)^2$ gets contributions from two cells, $\Omega^{(i-1)} = [x_{i-1}, x_i]$ and $\Omega^{(i)} = [x_i, x_{i+1}]$, but $\varphi_i'(x)^2 = h^{-2}$ in both cells, and the length of the integration interval is $2h$ so we get $A_{i-1,i-1} = 2h^{-1}$.

The right-hand side involves an integral of $2\varphi_i(x)$, $i = 1, \dots, N_n - 2$, which is just the area under a hat function of height 1 and width $2h$, i.e., equal to h . Hence, $b_{i-1} = 2h$.

To summarize the linear system, we switch from i to $i + 1$ such that we can write

$$A_{i,i-1} = A_{i,i+1} = -h^{-1}, \quad A_{i,i} = 2h^{-1}, \quad b_i = 2h.$$

The equation system to be solved only involves the unknowns c_i for $i \in \mathcal{I}_s$. With our numbering of unknowns and nodes, we have that c_i

equals $u(x_{i+1})$. The complete matrix system then takes the following form:

$$\frac{1}{h} \begin{pmatrix} 1 & -1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & 0 & -1 & 1 & & \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \end{pmatrix} \quad (6.3)$$

6.1.3 Comparison with a finite difference discretization

A typical row in the matrix system (6.3) can be written as

$$-\frac{1}{h}c_{i-1} + \frac{2}{h}c_i - \frac{1}{h}c_{i+1} = 2h. \quad (6.4)$$

Let us introduce the notation u_j for the value of u at node j : $u_j = u(x_j)$, since we have the interpretation $u(x_j) = \sum_j c_j \varphi(x_j) = \sum_j c_j \delta_{ij} = c_j$. The unknowns c_0, \dots, c_N are u_1, \dots, u_{N_n-2} . Shifting i with $i+1$ in (6.4) and inserting $u_i = c_{i-1}$, we get

$$-\frac{1}{h}u_{i-1} + \frac{2}{h}u_i - \frac{1}{h}u_{i+1} = 2h, \quad (6.5)$$

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

$$-\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} = 2, \quad (6.6)$$

which is, in fact, equal to (6.5) if (6.5) is divided by h . Therefore, the finite difference and the finite element method are equivalent in this simple test problem.

Sometimes a finite element method generates the finite difference equations on a uniform mesh, and sometimes the finite element method

generates equations that are different. The differences are modest, but may influence the numerical quality of the solution significantly, especially in time-dependent problems. It depends on the problem at hand whether a finite element discretization is more or less accurate than a corresponding finite difference discretization.

6.1.4 Cellwise computations

Software for finite element computations normally employs the cell by cell computational procedure where an element matrix and vector are calculated for each cell and assembled in the global linear system. Let us go through the details of this type of algorithm.

All integrals are mapped to the local reference coordinate system $X \in [-1, 1]$. In the present case, the matrix entries contain derivatives with respect to x ,

$$A_{i-1,j-1}^{(e)} = \int_{\Omega^{(e)}} \varphi'_i(x) \varphi'_j(x) dx = \int_{-1}^1 \frac{d}{dx} \tilde{\varphi}_r(X) \frac{d}{dx} \tilde{\varphi}_s(X) \frac{h}{2} dX,$$

where the global degree of freedom i is related to the local degree of freedom r through $i = q(e, r)$. Similarly, $j = q(e, s)$. The local degrees of freedom run as $r, s = 0, 1$ for a P1 element.

The integral for the element matrix. There are simple formulas for the basis functions $\tilde{\varphi}_r(X)$ as functions of X . However, we now need to find the derivative of $\tilde{\varphi}_r(X)$ with respect to x . Given

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

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

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

From the chain rule,

$$\frac{d\tilde{\varphi}_r}{dx} = \frac{d\tilde{\varphi}_r}{dX} \frac{dX}{dx} = \frac{2}{h} \frac{d\tilde{\varphi}_r}{dX}. \quad (6.7)$$

The transformed integral is then

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

The integral for the element vector. The right-hand side is transformed according to

$$b_{i-1}^{(e)} = \int_{\Omega^{(e)}} 2\varphi_i(x) dx = \int_{-1}^1 2\tilde{\varphi}_r(X) \frac{h}{2} dX, \quad i = q(e, r), \quad r = 0, 1.$$

Detailed calculations of the element matrix and vector. Specifically for P1 elements we arrive at the following calculations for the element matrix entries:

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

The element vector entries become

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

Expressing these entries in matrix and vector notation, we have

$$\tilde{A}^{(e)} = \frac{1}{h} \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(e)} = h \begin{pmatrix} 1 \\ 1 \end{pmatrix}. \quad (6.8)$$

Contributions from the first and last cell. The first and last cell involve only one unknown and one basis function because of the Dirichlet boundary conditions at the first and last node. The element matrix therefore becomes a 1×1 matrix and there is only one entry in the element vector. On cell 0, only $\psi_0 = \varphi_1$ is involved, corresponding to integration with $\tilde{\varphi}_1$. On cell N_e , only $\psi_N = \varphi_{N_n-2}$ is involved, corresponding to integration with $\tilde{\varphi}_0$. We then get the special end-cell contributions

$$\tilde{A}^{(e)} = \frac{1}{h} \begin{pmatrix} 1 \end{pmatrix}, \quad \tilde{b}^{(e)} = h \begin{pmatrix} 1 \end{pmatrix}, \quad (6.9)$$

for $e = 0$ and $e = N_e$. In these cells, we have only one degree of freedom, not two as in the interior cells.

Assembly. The next step is to assemble the contributions from the various cells. The assembly of an element matrix and vector into the global matrix and right-hand side can be expressed as

$$A_{q(e,r),q(e,s)} = A_{q(e,r),q(e,s)} + \tilde{A}_{r,s}^{(e)}, \quad b_{q(e,r)} = b_{q(e,r)} + \tilde{b}_r^{(e)},$$

for r and s running over all local degrees of freedom in cell e .

To make the assembly algorithm more precise, it is convenient to set up Python data structures and a code snippet for carrying out all details of the algorithm. For a mesh of four equal-sized P1 elements and $L = 2$ we have

```
vertices = [0, 0.5, 1, 1.5, 2]
cells = [[0, 1], [1, 2], [2, 3], [3, 4]]
dof_map = [[0], [0, 1], [1, 2], [2]]
```

The total number of degrees of freedom is 3, being the function values at the internal 3 nodes where u is unknown. In cell 0 we have global degree of freedom 0, the next cell has u unknown at its two nodes, which become global degrees of freedom 0 and 1, and so forth according to the `dof_map` list. The mathematical $q(e,r)$ quantity is nothing but the `dof_map` list.

Assume all element matrices are stored in a list `Ae` such that $\text{Ae}[e][i,j]$ is $\tilde{A}_{i,j}^{(e)}$. A corresponding list for the element vectors is named `be`, where $\text{be}[e][r]$ is $\tilde{b}_r^{(e)}$. A Python code snippet illustrates all details of the assembly algorithm:

```
# A[i,j]: coefficient matrix, b[i]: right-hand side
for e in range(len(Ae)):
    for r in range(Ae[e].shape[0]):
        for s in range(Ae[e].shape[1]):
            A[dof_map[e,r],dof_map[e,s]] += Ae[e][i,j]
            b[dof_map[e,r]] += be[e][i,j]
```

The general case with N_e P1 elements of length h has

```
N_n = N_e + 1
vertices = [i*h for i in range(N_n)]
cells = [[e, e+1] for e in range(N_e)]
dof_map = [[0]] + [[e-1, e] for i in range(1, N_e)] + [[N_n-2]]
```

Carrying out the assembly results in a linear system that is identical to (6.3), which is not surprising, since the procedures is mathematically equivalent to the calculations in the physical domain.

So far, our technique for computing the matrix system have assumed that $u(0) = u(L) = 0$. The next section deals with the extension to nonzero Dirichlet conditions.

6.2 Boundary conditions: specified nonzero value

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

6.2.1 General construction of a boundary function

In Section 5.1.12 we introduced a boundary function $B(x)$ to deal with nonzero Dirichlet boundary conditions for u . The construction of such a function is not always trivial, especially not in multiple dimensions. However, a simple and general constructive idea exists when the basis functions have the property

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

where x_j is a boundary point. Examples on such functions are the Lagrange interpolating polynomials and finite element functions.

Suppose now that u has Dirichlet boundary conditions at nodes with numbers $i \in I_b$. For example, $I_b = \{0, N_n - 1\}$ in a 1D mesh with node numbering from left to right and Dirichlet conditions at the end nodes $i = 0$ and $i = N_n - 1$. Let U_i be the corresponding prescribed values of $u(x_i)$. We can then, in general, use

$$B(x) = \sum_{j \in I_b} U_j \varphi_j(x). \quad (6.10)$$

It is easy to verify that $B(x_i) = \sum_{j \in I_b} U_j \varphi_j(x_i) = U_i$.

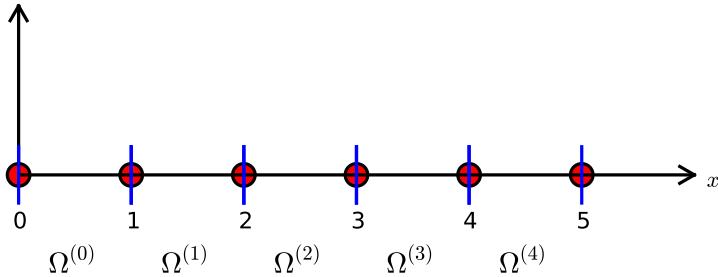
The unknown function can then be written as

$$u(x) = \sum_{j \in I_b} U_j \varphi_j(x) + \sum_{j \in \mathcal{I}_s} c_j \varphi_{\nu(j)}, \quad (6.11)$$

where $\nu(j)$ maps unknown number j in the equation system to node $\nu(j)$, I_b is the set of indices corresponding to basis functions associated with nodes where Dirichlet conditions apply, and \mathcal{I}_s is the set of indices used to number the unknowns from zero to N . We can easily show that with this u , a Dirichlet condition $u(x_k) = U_k$ is fulfilled:

$$u(x_k) = \sum_{j \in I_b} U_j \underbrace{\varphi_j(x_k)}_{\neq 0 \Leftrightarrow j=k} + \sum_{j \in \mathcal{I}_s} c_j \underbrace{\varphi_{\nu(j)}(x_k)}_{=0, k \notin \mathcal{I}_s} = U_k$$

Some examples will further clarify the notation. With a regular left-to-right numbering of nodes in a mesh with P1 elements, and Dirichlet conditions at $x = 0$, we use finite element basis functions associated with the nodes $1, 2, \dots, N_n - 1$, implying that $\nu(j) = j + 1$, $j = 0, \dots, N$, where $N = N_n - 2$. Consider a particular mesh:



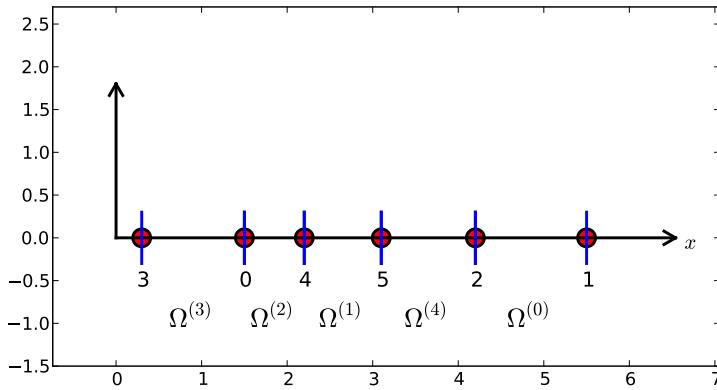
The expansion associated with this mesh becomes

$$u(x) = U_0 \varphi_0(x) + c_0 \varphi_1(x) + c_1 \varphi_2(x) + \dots + c_4 \varphi_5(x).$$

Switching to the more standard case of left-to-right numbering and boundary conditions $u(0) = C$, $u(L) = D$, we have $N = N_n - 3$ and

$$\begin{aligned} u(x) &= C \varphi_0 + D \varphi_{N_n-1} + \sum_{j \in \mathcal{I}_s} c_j \varphi_{j+1} \\ &= C \varphi_0 + D \varphi_{N_n} + c_0 \varphi_1 + c_1 \varphi_2 + \dots + c_{N_n-2} \varphi_{N_n-2}. \end{aligned}$$

Finite element meshes in non-trivial 2D and 3D geometries usually leads to an irregular cell and node numbering. Let us therefore take a look at an irregular numbering in 1D:



Say we in this mesh have Dirichlet conditions on the left-most and right-most node, with numbers 3 and 1, respectively. We can number the unknowns at the interior nodes as we want, e.g., from left to right, resulting in $\nu(0) = 0$, $\nu(1) = 4$, $\nu(2) = 5$, $\nu(3) = 2$. This gives

$$B(x) = U_3\varphi_3(x) + U_1\varphi_1(x),$$

and

$$u(x) = B(x) + \sum_{j=0}^3 c_j \varphi_{\nu(j)} = U_3\varphi_3 + U_1\varphi_1 + c_0\varphi_0 + c_1\varphi_4 + c_2\varphi_5 + c_3\varphi_2.$$

The idea of constructing B , described here, generalizes almost trivially to 2D and 3D problems: $B = \sum_{j \in I_b} U_j \varphi_j$, where I_b is the index set containing the numbers of all the nodes on the boundaries where Dirichlet values are prescribed.

6.2.2 Example on computing with a finite element-based boundary function

Let us see how the model problem $-u'' = 2$, $u(0) = C$, $u(L) = D$, is affected by a $B(x)$ to incorporate boundary values. Inserting the expression

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)$$

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

$$A_{i,j} = \int_0^L \psi'_i(x) \psi'_j(x) dx, \quad b_i = \int_0^L (f(x) \psi_i(x) - B'(x) \psi'_i(x)) dx.$$

We choose $\psi_i = \varphi_{i+1}$, $i = 0, \dots, N = N_n - 3$ if the node numbering is from left to right. (Later we also need the assumption that cells too are numbered from left to right.) The boundary function becomes

$$B(x) = C\varphi_0(x) + D\varphi_{N_n-1}(x).$$

The expansion for $u(x)$ is

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \varphi_{j+1}(x).$$

We can write the matrix and right-hand side entries as

$$\begin{aligned} A_{i-1,j-1} &= \int_0^L \varphi'_i(x) \varphi'_j(x) dx, \\ b_{i-1} &= \int_0^L (f(x) \varphi'_i(x) - (C\varphi'_0(x) + D\varphi'_{N_n-1}(x)) \varphi'_i(x)) dx, \end{aligned}$$

for $i, j = 1, \dots, N + 1 = N_n - 2$. Note that we have here used $B' = C\varphi'_0 + D\varphi'_{N_n-1}$.

Computations in physical coordinates. Most of the terms in the linear system have already been computed so we concentrate on the new contribution from the boundary function. The integral $C \int_0^L \varphi'_0(x) \varphi'_i(x) dx$, associated with the Dirichlet condition in $x = 0$, can only get a nonzero contribution from the first cell, $\Omega^{(0)} = [x_0, x_1]$ since $\varphi'_0(x) = 0$ on all other cells. Moreover, $\varphi'_0(x) \varphi'_i(x) dx \neq 0$ only for $i = 0$ and $i = 1$ (but node $i = 0$ is excluded from the formulation), since $\varphi_i = 0$ on the first cell if $i > 1$. With a similar reasoning we realize that $D \int_0^L \varphi'_{N_n-1}(x) \varphi'_i(x) dx$ can only get a nonzero contribution from the last cell. From the explanations of the calculations in Section 4.1.6 we then find that

$$\int_0^L \varphi'_0(x)\varphi'_1(x) dx = \left(-\frac{1}{h}\right) \cdot \frac{1}{h} \cdot h = -\frac{1}{h},$$

$$\int_0^L \varphi'_{N_n-1}(x)\varphi'_{N_n-2}(x) dx = \frac{1}{h} \cdot \left(-\frac{1}{h}\right) \cdot h = -\frac{1}{h}.$$

With these expressions we get

$$b_0 = \int_0^L f(x)\varphi_1 dx - C\left(-\frac{1}{h}\right), \quad b_N = \int_0^L f(x)\varphi_{N_n-2} dx - D\left(-\frac{1}{h}\right).$$

Cellwise computations on the reference element. As an equivalent alternative, we now turn to cellwise computations. The element matrices and vectors are calculated as in Section 6.1.4, so we concentrate on the impact of the new term involving $B(x)$. This new term, $B' = C\varphi'_0 + D\varphi'_{N_n-1}$, vanishes on all cells except for $e = 0$ and $e = N_e$. Over the first cell ($e = 0$) the $B'(x)$ function in local coordinates reads

$$\frac{dB}{dx} = C\frac{2}{h} \frac{d\tilde{\varphi}_0}{dX},$$

while over the last cell ($e = N_e$) it looks like

$$\frac{dB}{dx} = D\frac{2}{h} \frac{d\tilde{\varphi}_1}{dX}.$$

For an arbitrary interior cell, we have the formula

$$\tilde{b}_r^{(e)} = \int_{-1}^1 f(x(X))\tilde{\varphi}_r(X) \frac{h}{2} dX,$$

for an entry in the local element vector. In the first cell, the value at local node 0 is known so only the value at local node 1 is unknown. The associated element vector entry becomes

$$\begin{aligned} \tilde{b}_0^{(1)} &= \int_{-1}^1 \left(f\tilde{\varphi}_1 - C\frac{2}{h} \frac{d\tilde{\varphi}_0}{dX} \frac{2}{h} \frac{d\tilde{\varphi}_1}{dX} \right) \frac{h}{2} dX \\ &= \frac{h}{2} 2 \int_{-1}^1 \tilde{\varphi}_1 dX - C\frac{2}{h} \left(-\frac{1}{2}\right) \frac{2}{h} \frac{1}{2} \frac{h}{2} \cdot 2 = h + C\frac{1}{h}. \end{aligned}$$

The value at local node 1 in the last cell is known so the element vector here is

$$\begin{aligned}\tilde{b}_0^{N_e} &= \int_{-1}^1 \left(f\tilde{\varphi}_0 - D \frac{2}{h} \frac{d\tilde{\varphi}_1}{dX} \frac{2}{h} \frac{d\tilde{\varphi}_0}{dX} \right) \frac{h}{2} dX \\ &= \frac{h}{2} \cdot 2 \int_{-1}^1 \tilde{\varphi}_0 dX - D \frac{2}{h} \frac{1}{2} \frac{2}{h} \left(-\frac{1}{2}\right) \frac{h}{2} \cdot 2 = h + D \frac{1}{h}.\end{aligned}$$

The contributions from the $B(x)$ function to the global right-hand side vector becomes C/h for b_0 and D/h for b_N , exactly as we computed in the physical domain.

6.2.3 Modification of the linear system

From an implementational point of view, there is a convenient alternative to adding the $B(x)$ function and using only the basis functions associated with nodes where u is truly unknown. Instead of seeking

$$u(x) = \sum_{j \in I_b} U_j \varphi_j(x) + \sum_{j \in \mathcal{I}_s} c_j \varphi_{\nu(j)}(x), \quad (6.12)$$

we use the sum over all degrees of freedom, including the known boundary values:

$$u(x) = \sum_{j \in \mathcal{I}_s} c_j \varphi_j(x). \quad (6.13)$$

Note that the collections of unknowns $\{c_i\}_{i \in \mathcal{I}_s}$ in (6.12) and (6.13) are different. The index set $\mathcal{I}_s = \{0, \dots, N\}$ always goes to N , and the number of unknowns is $N + 1$, but in (6.12) the unknowns correspond to nodes where u is not known, while in (6.13) the unknowns cover u values at all the nodes. So, if the index set I_b contains N_b node numbers where u is prescribed, we have that $N = N_n - N_b$ in (6.12) and $N = N_n$ in (6.13).

The idea is to compute the entries in the linear system as if no Dirichlet values are prescribed. Afterwards, we modify the linear system to ensure that the known c_j values are incorporated.

A potential problem arises for the boundary term $[u'v]_0^L$ from the integration by parts: imagining no Dirichlet conditions means that we no longer require $v = 0$ at Dirichlet points, and the boundary term is then nonzero at these points. However, when we modify the linear system, we will erase whatever the contribution from $[u'v]_0^L$ should be at the Dirichlet points in the right-hand side of the linear system. We can therefore safely forget $[u'v]_0^L$ at any point where a Dirichlet condition applies.

Computations in the physical system. Let us redo the computations in the example in Section 6.2.1. We solve $-u'' = 2$ with $u(0) = 0$ and $u(L) = D$. The expressions for $A_{i,j}$ and b_i are the same, but the numbering is different as the numbering of unknowns and nodes now coincide:

$$A_{i,j} = \int_0^L \varphi'_i(x) \varphi'_j(x) dx, \quad b_i = \int_0^L f(x) \varphi_i(x) dx,$$

for $i, j = 0, \dots, N = N_n - 1$. The integrals involving basis functions corresponding to interior mesh nodes, $i, j = 1, \dots, N_n - 2$, are obviously the same as before. We concentrate on the contributions from φ_0 and φ_{N_n-1} :

$$\begin{aligned} A_{0,0} &= \int_0^L (\varphi'_0)^2 dx = \int_0^{x_1} (\varphi'_0)^2 dx \frac{1}{h}, \\ A_{0,1} &= \int_0^L \varphi'_0 \varphi'_1 dx = \int_0^{x_1} \varphi'_0 \varphi'_1 dx = -\frac{1}{h}, \\ A_{N,N} &= \int_0^L (\varphi'_N)^2 dx = \int_{x_{N_n-2}}^{x_{N_n-1}} (\varphi'_N)^2 dx = \frac{1}{h}, \\ A_{N,N-1} &= \int_0^L \varphi'_N \varphi'_{N-1} dx = \int_{x_{N_n-2}}^{x_{N_n-1}} \varphi'_N \varphi'_{N-1} dx = -\frac{1}{h}. \end{aligned}$$

The new terms on the right-hand side are also those involving φ_0 and φ_{N_n-1} :

$$\begin{aligned} b_0 &= \int_0^L 2\varphi_0(x) dx = \int_0^{x_1} 2\varphi_0(x) dx = h, \\ b_N &= \int_0^L 2\varphi_{N_n-1}(x) dx = \int_{x_{N_n-2}}^{x_{N_n-1}} 2\varphi_{N_n-1}(x) dx = h. \end{aligned}$$

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

$$\frac{1}{h} \begin{pmatrix} 1 & -1 & 0 & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} h \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \\ h \end{pmatrix} \quad (6.14)$$

Incorporation of Dirichlet values can now be done by replacing the first and last equation by the very simple equations $c_0 = 0$ and $c_N = D$, respectively. Note that the factor $1/h$ in front of the matrix then requires a factor h to be introduced appropriately on the diagonal in the first and last row of the matrix.

$$\frac{1}{h} \begin{pmatrix} h & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & 0 & 0 & h \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} 0 \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \\ D \end{pmatrix} \quad (6.15)$$

Note that because we do not require $\varphi_i(0) = 0$ and $\varphi_i(L) = 0$, $i \in \mathcal{I}_s$, the boundary term $[u'v]_0^L$, in principle, gives contributions $u'(0)\varphi_0(0)$ to b_0 and $u'(L)\varphi_N(L)$ to b_N ($u'\varphi_i$ vanishes for $x = 0$ or $x = L$ for $i = 1, \dots, N-1$). Nevertheless, we erase these contributions in b_0 and b_N and insert boundary values instead. This argument shows why we can drop computing $[u'v]_0^L$ at Dirichlet nodes when we implement the Dirichlet values by modifying the linear system.

6.2.4 Symmetric modification of the linear system

The original matrix system (6.3) is symmetric, but the modifications in (6.15) destroy this symmetry. Our described modification will in general destroy an initial symmetry in the matrix system. This is not a particular computational disadvantage for tridiagonal systems arising in 1D problems, but may be more serious in 2D and 3D problems when the systems are large and exploiting symmetry can be important for halving the storage demands and speeding up computations. Methods for solving symmetric matrix are also usually more stable and efficient than those for non-symmetric systems. Therefore, an alternative modification which preserves symmetry is attractive.

One can formulate a general algorithm for incorporating a Dirichlet condition in a symmetric way. Let c_k be a coefficient corresponding to a known value $u(x_k) = U_k$. We want to replace equation k in the system by $c_k = U_k$, i.e., insert zeroes in row number k in the coefficient matrix, set 1 on the diagonal, and replace b_k by U_k . A symmetry-preserving modification consists in first subtracting column number k in the coefficient matrix, i.e., $A_{i,k}$ for $i \in \mathcal{I}_s$, times the boundary value U_k , from the right-hand side: $b_i \leftarrow b_i - A_{i,k}U_k$, $i = 0, \dots, N$. Then we put zeroes in both row number k and column number k in the coefficient matrix, and finally set $b_k = U_k$. The steps in algorithmic form becomes

1. $b_i \leftarrow b_i - A_{i,k}U_k$ for $i \in \mathcal{I}_s$
2. $A_{i,k} = A_{k,i} = 0$ for $i \in \mathcal{I}_s$
3. $A_{k,k} = 1$
4. $b_k = U_k$

This modification goes as follows for the specific linear system written out in (6.14) in Section 6.2.3. First we subtract the first column in the coefficient matrix, times the boundary value, from the right-hand side. Because $c_0 = 0$, this subtraction has no effect. Then we subtract the last column, times the boundary value D , from the right-hand side. This action results in $b_{N-1} = 2h + D/h$ and $b_N = h - 2D/h$. Thereafter, we place zeros in the first and last row and column in the coefficient matrix and 1 on the two corresponding diagonal entries. Finally, we set $b_0 = 0$ and $b_N = D$. The result becomes

$$\frac{1}{h} \begin{pmatrix} h & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & 2 & -1 & \ddots & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & \cdots & 0 & 0 & h \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} 0 \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h + D/h \\ D \end{pmatrix} \quad (6.16)$$

6.2.5 Modification of the element matrix and vector

The modifications of the global linear system can alternatively be done for the element matrix and vector. Let us perform the associated calculations in the computational example where the element matrix and vector is given by (6.8). The modifications are needed in cells where one of the degrees of freedom is known. In the present example, this means the first and last cell. We compute the element matrix and vector as if there were no Dirichlet conditions. The boundary term $[u'v]_0^L$ is simply forgotten at nodes that have Dirichlet conditions because the modification of the element vector will anyway erase the contribution from the boundary term. In the first cell, local degree of freedom number 0 is known and the modification becomes

$$\tilde{A}^{(0)} = A = \frac{1}{h} \begin{pmatrix} h & 0 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(0)} = \begin{pmatrix} 0 \\ h \end{pmatrix}. \quad (6.17)$$

In the last cell we set

$$\tilde{A}^{(N_e)} = A = \frac{1}{h} \begin{pmatrix} 1 & -1 \\ 0 & h \end{pmatrix}, \quad \tilde{b}^{(N_e)} = \begin{pmatrix} h \\ D \end{pmatrix}. \quad (6.18)$$

We can also perform the symmetric modification. This operation affects only the last cell with a nonzero Dirichlet condition. The algorithm is the same as for the global linear system, resulting in

$$\tilde{A}^{(N_e)} = A = \frac{1}{h} \begin{pmatrix} 1 & 0 \\ 0 & h \end{pmatrix}, \quad \tilde{b}^{(N_e)} = \begin{pmatrix} h + D/h \\ D \end{pmatrix}. \quad (6.19)$$

The reader is encouraged to assemble the element matrices and vectors and check that the result coincides with the system (6.16).

6.3 Boundary conditions: specified derivative

Suppose our model problem $-u''(x) = f(x)$ features the boundary conditions $u'(0) = C$ and $u(L) = D$. As already indicated in Section 5.3, the former condition can be incorporated through the boundary term that arises from integration by parts. The details of this method will now be illustrated in the context of finite element basis functions.

6.3.1 The variational formulation

Starting with the Galerkin method,

$$\int_0^L (u''(x) + f(x))\psi_i(x) \, dx = 0, \quad i \in \mathcal{I}_s,$$

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

$$\int_0^L u'(x)' \psi'_i(x) \, dx - (u'(L)\psi_i(L) - u'(0)\psi_i(0)) = \int_0^L f(x)\psi_i(x) \, dx, \quad i \in \mathcal{I}_s.$$

The first boundary term, $u'(L)\psi_i(L)$, vanishes because $u(L) = D$. The second boundary term, $u'(0)\psi_i(0)$, can be used to implement the condition $u'(0) = C$, provided $\psi_i(0) \neq 0$ for some i (but with finite elements we fortunately have $\psi_0(0) = 1$). The variational form of the differential equation then becomes

$$\int_0^L u'(x)\varphi'_i(x) \, dx + C\varphi_i(0) = \int_0^L f(x)\varphi_i(x) \, dx, \quad i \in \mathcal{I}_s.$$

6.3.2 Boundary term vanishes because of the test functions

At points where u is known we may require ψ_i to vanish. Here, $u(L) = D$ and then $\psi_i(L) = 0$, $i \in \mathcal{I}_s$. Obviously, the boundary term $u'(L)\psi_i(L)$ then vanishes.

The set of basis functions $\{\psi_i\}_{i \in \mathcal{I}_s}$ contains, in this case, all the finite element basis functions on the mesh, except the one that is 1 at $x = L$.

The basis function that is left out is used in a boundary function $B(x)$ instead. With a left-to-right numbering, $\psi_i = \varphi_i$, $i = 0, \dots, N_n - 2$, and $B(x) = D\varphi_{N_n-1}$:

$$u(x) = D\varphi_{N_n-1}(x) + \sum_{j=0}^{N_n-2} c_j \varphi_j(x).$$

Inserting this expansion for u in the variational form (6.3.1) leads to the linear system

$$\sum_{j=0}^N \left(\int_0^L \varphi'_i(x) \varphi'_j(x) dx \right) c_j = \int_0^L (f(x) \varphi_i(x) - D\varphi'_{N_n-1}(x) \varphi'_i(x)) dx - C\varphi_i(0), \quad (6.20)$$

for $i = 0, \dots, N = N_n - 2$.

6.3.3 Boundary term vanishes because of linear system modifications

We may, as an alternative to the approach in the previous section, use a basis $\{\psi_i\}_{i \in \mathcal{I}_s}$ which contains all the finite element functions on the mesh: $\psi_i = \varphi_i$, $i = 0, \dots, N_n - 1 = N$. In this case, $u'(L)\psi_i(L) = u'(L)\varphi_i(L) \neq 0$ for the i corresponding to the boundary node at $x = L$ (where $\varphi_i = 1$). The number of this node is $i = N_n - 1 = N$ if a left-to-right numbering of nodes is utilized.

However, even though $u'(L)\varphi_{N_n-1}(L) \neq 0$, we do not need to compute this term. For $i < N_n - 1$ we realize that $\varphi_i(L) = 0$. The only nonzero contribution to the right-hand side comes from $i = N$ (b_N). Without a boundary function we must implement the condition $u(L) = D$ by the equivalent statement $c_N = D$ and modify the linear system accordingly. This modification will erase the last row and replace b_N by another value. Any attempt to compute the boundary term $u'(L)\varphi_{N_n-1}(L)$ and store it in b_N will be lost. Therefore, we can safely forget about boundary terms corresponding to Dirichlet boundary conditions also when we use the methods from Section 6.2.3 or Section 6.2.4.

The expansion for u reads

$$u(x) = \sum_{j \in \mathcal{I}_s} c_j \varphi_j(x).$$

Insertion in the variational form (6.3.1) leads to the linear system

$$\sum_{j \in \mathcal{I}_s} \left(\int_0^L \varphi'_i(x) \varphi'_j(x) dx \right) c_j = \int_0^L (f(x) \varphi_i(x)) dx - C \varphi_i(0), \quad i \in \mathcal{I}_s. \quad (6.21)$$

After having computed the system, we replace the last row by $c_N = D$, either straightforwardly as in Section 6.2.3 or in a symmetric fashion as in Section 6.2.4. These modifications can also be performed in the element matrix and vector for the right-most cell.

6.3.4 Direct computation of the global linear system

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

Consider first the approach where Dirichlet conditions are incorporated by a $B(x)$ function and the known degree of freedom C_{N_n-1} is left out of the linear system (see Section 6.3.2). The relevant formula for the linear system is given by (6.20). There are three differences compared to the extensively computed case where $u(0) = 0$ in Sections 6.1.2 and 6.1.4. First, because we do not have a Dirichlet condition at the left boundary, we need to extend the linear system (6.3) with an equation associated with the node $x_0 = 0$. According to Section 6.2.3, this extension consists of including $A_{0,0} = 1/h$, $A_{0,1} = -1/h$, and $b_0 = h$. For $i > 0$ we have $A_{i,i} = 2/h$, $A_{i-1,i} = A_{i,i+1} = -1/h$. Second, we need to include the extra term $-C\varphi_i(0)$ on the right-hand side. Since all $\varphi_i(0) = 0$ for $i = 1, \dots, N$, this term reduces to $-C\varphi_0(0) = -C$ and affects only the first equation ($i = 0$). We simply add $-C$ to b_0 such that $b_0 = h - C$. Third, the boundary term $-\int_0^L D\varphi_{N_n-1}(x) \varphi_i dx$ must be computed. Since $i = 0, \dots, N = N_n - 2$, this integral can only get a nonzero contribution with $i = N_n - 2$ over the last cell. The result becomes $-Dh/6$. The resulting linear system can be summarized in the form

$$\frac{1}{h} \begin{pmatrix} 1 & -1 & 0 & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & 0 & -1 & 2 & \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} h - C \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h - Dh/6 \end{pmatrix}. \quad (6.22)$$

Next we consider the technique where we modify the linear system to incorporate Dirichlet conditions (see Section 6.3.3). Now $N = N_n - 1$. The two differences from the case above is that the $-\int_0^L D\varphi_{N_n-1}\varphi_i \, dx$ term is left out of the right-hand side and an extra last row associated with the node $x_{N_n-1} = L$ where the Dirichlet condition applies is appended to the system. This last row is anyway replaced by the condition $c_N = D$ or this condition can be incorporated in a symmetric fashion. Using the simplest, former approach gives

$$\frac{1}{h} \begin{pmatrix} 1 & -1 & 0 & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & -1 & 2 & -1 \\ 0 & \cdots & \cdots & \cdots & 0 & 0 & h & \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} h - C \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \\ D \end{pmatrix}. \quad (6.23)$$

6.3.5 Cellwise computations

Now we compute with one element at a time, working in the reference coordinate system $X \in [-1, 1]$. We need to see how the $u'(0) = C$

condition affects the element matrix and vector. The extra term $-C\varphi_i(0)$ in the variational formulation only affects the element vector in the first cell. On the reference cell, $-C\varphi_i(0)$ is transformed to $-C\tilde{\varphi}_r(-1)$, where r counts local degrees of freedom. We have $\tilde{\varphi}_0(-1) = 1$ and $\tilde{\varphi}_1(-1) = 0$ so we are left with the contribution $-C\tilde{\varphi}_0(-1) = -C$ to $\tilde{b}_0^{(0)}$:

$$\tilde{A}^{(0)} = A = \frac{1}{h} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(0)} = \begin{pmatrix} h - C \\ h \end{pmatrix}. \quad (6.24)$$

No other element matrices or vectors are affected by the $-C\varphi_i(0)$ boundary term.

There are two alternative ways of incorporating the Dirichlet condition. Following Section 6.3.2, we get a 1×1 element matrix in the last cell and an element vector with an extra term containing D :

$$\tilde{A}^{(e)} = A = \frac{1}{h} \begin{pmatrix} 1 \end{pmatrix}, \quad \tilde{b}^{(e)} = h \begin{pmatrix} 1 - D/6 \end{pmatrix}, \quad (6.25)$$

Alternatively, we include the degree of freedom at the node with u specified. The element matrix and vector must then be modified to constrain the $\tilde{c}_1 = c_N$ value at local node $r = 1$:

$$\tilde{A}^{(N_e)} = A = \frac{1}{h} \begin{pmatrix} 1 & 1 \\ 0 & h \end{pmatrix}, \quad \tilde{b}^{(N_e)} = \begin{pmatrix} h \\ D \end{pmatrix}. \quad (6.26)$$

6.4 Implementation of finite element algorithms

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

6.4.1 Extensions of the code for approximation

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

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

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

- Integrand on the left-hand side: `ilhs(e, phi, r, s, X, x, h)`
- Integrand on the right-hand side: `irhs(e, phi, r, X, x, h)`
- Boundary term on the left-hand side: `blhs (e, phi, r, s, X, x, h)`
- Boundary term on the right-hand side: `brhs (e, phi, r, s, X, x, h)`

Here, `phi` is a dictionary where `phi[q]` holds a list of the derivatives of order q of the basis functions with respect to the physical coordinate x . The derivatives are available as Python functions of X . For example, `phi[0][r](X)` means $\tilde{\varphi}_r(X)$, and `phi[1][s](X, h)` means $d\tilde{\varphi}_s(X)/dx$ (we refer to the file `fe1D.py` for details regarding the function `basis` that computes the `phi` dictionary). The `r` and `s` arguments in the above functions correspond to the index in the integrand contribution from an integration point to $\tilde{A}_{r,s}^{(e)}$ and $\tilde{b}_r^{(e)}$. The variables `e` and `h` are the current element number and the length of the cell, respectively. Specific examples below will make it clear how to construct these Python functions.

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

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

for e in range(len(cells)):

    # Compute element matrix and vector
    n = len(dof_map[e]) # no of dofs in this element
    h = vertices[cells[e][1]] - vertices[cells[e][0]]
    <Initialize element matrix and vector: A_e, b_e>

    # Integrate over the reference cell
    points, weights = <numerical integration rule>
    for X, w in zip(points, weights):
        phi = <basis functions and derivatives at X>
```

```

detJ = h/2
dX = detJ*w

x = <affine mapping from X>
for r in range(n):
    for s in range(n):
        A_e[r,s] += ilhs(e, phi, r, s, X, x, h)*dX
        b_e[r] += irhs(e, phi, r, X, x, h)*dX

# Add boundary terms
for r in range(n):
    for s in range(n):
        A_e[r,s] += blhs(e, phi, r, s, X, x)*dX
        b_e[r] += brhs(e, phi, r, X, x, h)*dX

# Incorporate essential boundary conditions
for r in range(n):
    global_dof = dof_map[e][r]
    if global_dof in essbc:
        # local dof r is subject to an essential condition
        value = essbc[global_dof]
        # Symmetric modification
        b_e -= value*A_e[:,r]
        A_e[:,r] = 0
        A_e[:,r] = 0
        A_e[r,r] = 1
        b_e[r] = value

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

<solve linear system>

```

Having implemented this function, the user only has supply the `ilhs`, `irhs`, `blhs`, and `brhs` functions that specify the PDE and boundary conditions. The rest of the implementation forms a generic computational engine. The big finite element packages Diffpack and FEniCS are structured exactly the same way. A sample implementation of `ilhs` for the 1D Poisson problem is:

```

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

```

which returns $d\tilde{\varphi}_i(X)/dx d\tilde{\varphi}_j(X)/dx$. Reducing the amount of code the user has to supply and making the code as close as possible to the mathematical formulation makes the software user-friendly and easy to debug.

A complete function `finite_element1D_naive` for the 1D finite algorithm above, is found in the file `fe1D.py`. The term “naive” refers to a version of the algorithm where we use a standard dense square matrix as global matrix A . The implementation also has a verbose mode for printing out the element matrices and vectors as they are computed. Below is the complete function without the print statements. You should study in detail since it contains all the steps in the finite element algorithm.

```
def finite_element1D_naive(
    vertices, cells, dof_map,      # mesh
    essbc,                         # essbc[globdof]=value
    ilhs,                          # integrand left-hand side
    irhs,                          # integrand right-hand side
    blhs=lambda e, phi, r, s, X, x, h: 0,
    brhs=lambda e, phi, r, X, x, h: 0,
    intrule='GaussLegendre',       # integration rule class
    verbose=False,                  # print intermediate results?
):
    N_e = len(cells)
    N_n = np.array(dof_map).max() + 1

    A = np.zeros((N_n, N_n))
    b = np.zeros(N_n)

    for e in range(N_e):
        Omega_e = [vertices[cells[e][0]], vertices[cells[e][1]]]
        h = Omega_e[1] - Omega_e[0]

        d = len(dof_map[e]) - 1 # Polynomial degree
        # Compute all element basis functions and their derivatives
        phi = basis(d)

        # Element matrix and vector
        n = d+1 # No of dofs per element
        A_e = np.zeros((n, n))
        b_e = np.zeros(n)

        # Integrate over the reference cell
        if intrule == 'GaussLegendre':
            points, weights = GaussLegendre(d+1)
        elif intrule == 'NewtonCotes':
            points, weights = NewtonCotes(d+1)

        for X, w in zip(points, weights):
            detJ = h/2
            x = affine_mapping(X, Omega_e)
            dX = detJ*w

            # Compute contribution to element matrix and vector
            for r in range(n):
                for s in range(n):
                    A_e[r,s] += ilhs(phi, r, s, X, x, h)*dX
```

```

    b_e[r] += irhs(phi, r, X, x, h)*dX

    # Add boundary terms
    for r in range(n):
        for s in range(n):
            A_e[r,s] += blhs(phi, r, s, X, x, h)
        b_e[r] += brhs(phi, r, X, x, h)

    # Incorporate essential boundary conditions
    modified = False
    for r in range(n):
        global_dof = dof_map[e][r]
        if global_dof in essbc:
            # dof r is subject to an essential condition
            value = essbc[global_dof]
            # Symmetric modification
            b_e -= value*A_e[:,r]
            A_e[r,:] = 0
            A_e[:,r] = 0
            A_e[r,r] = 1
            b_e[r] = value
            modified = True

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

    c = np.linalg.solve(A, b)
    return c, A, b, timing

```

The `timing` object is a dictionary holding the CPU spent on computing `A` and the CPU time spent on solving the linear system. (We have left out the timing statements.)

6.4.2 Utilizing a sparse matrix

A potential efficiency problem with the `finite_element1D_naive` function is that it uses dense $(N + 1) \times (N + 1)$ matrices, while we know that only $2d + 1$ diagonals around the main diagonal are different from zero. Switching to a sparse matrix is very easy. Using the DOK (dictionary of keys) format, we declare `A` as

```

import scipy.sparse
A = scipy.sparse.dok_matrix((N_n, N_n))

```

Assignments or in-place arithmetics are done as for a dense matrix,

```
A[i,j] += term
```

```
A[i,j] = term
```

but only the index pairs (i, j) we have used in assignments or in-place arithmetics are actually stored. A tailored solution algorithm is needed. The most reliable is sparse Gaussian elimination. SciPy gives access to the **UMFPACK** algorithm for this purpose:

```
import scipy.sparse.linalg
c = scipy.sparse.linalg.spsolve(A.tocsr(), b, use_umfpack=True)
```

The declaration of A and the solve statement are the only changes needed in the `finite_element1D_naive` to utilize sparse matrices. The resulting modification is found in the function `finite_element1D`.

6.4.3 Application to our model problem

Let us demonstrate the finite element software on

$$-u''(x) = f(x), \quad x \in (0, L), \quad u'(0) = C, \quad u(L) = D.$$

This problem can be analytically solved by the `model2` function from Section 5.1.2. Let $f(x) = x^2$. Calling `model2(x**2, L, C, D)` gives

$$u(x) = D + C(x - L) + \frac{1}{12}(L^4 - x^4)$$

The variational formulation reads

$$(u', v) = (x^2, v) - Cv(0).$$

The entries in the element matrix and vector, which we need to set up the `ilhs`, `irhs`, `blhs`, and `brhs` functions, becomes

$$\begin{aligned} A_{r,s}^{(e)} &= \int_{-1}^1 \frac{d\tilde{\varphi}_r}{dx} \frac{\tilde{\varphi}_s}{dx} (\det J dX), \\ b^{(e)} &= \int_{-1}^1 x^2 \tilde{\varphi}_r \det J dX - C\tilde{\varphi}_r(-1)I(e, 0), \end{aligned}$$

where $I(e)$ is an indicator function: $I(e, q) = 1$ if $e = q$, otherwise $I(e) = 0$. We use this indicator function to formulate that the boundary term $Cv(0)$, which in the local element coordinate system becomes $C\tilde{\varphi}_r(-1)$, is only included for the element $e = 0$.

The functions for specifying the element matrix and vector entries must contain the integrand, but without the $\det J dX$ term as this term

is taken care of by the quadrature loop, and the derivatives $d\tilde{\varphi}_r(X)/dx$ with respect to the physical x coordinates are contained in `phi[1][r](X)`, computed by the function `basis`.

```
def ilhs(e, phi, r, s, X, x, h):
    return phi[1][r](X, h)*phi[1][s](X, h)

def irhs(e, phi, r, X, x, h):
    return x**2*phi[0][r](X)

def blhs(e, phi, r, s, X, x, h):
    return 0

def brhs(e, phi, r, X, x, h):
    return -C*phi[0][r](-1) if e == 0 else 0
```

We can then make the call to `finite_element1D_naive` or `finite_element1D` to solve the problem with two P1 elements:

```
from fe1D import finite_element1D_naive, mesh_uniform
C = 5; D = 2; L = 4
d = 1

vertices, cells, dof_map = mesh_uniform(
    N_e=2, d=d, Omega=[0,L], symbolic=False)
essbc = {}
essbc[dof_map[-1][-1]] = D

c, A, b, timing = finite_element1D(
    vertices, cells, dof_map, essbc,
    ilhs=ilhs, irhs=irhs, blhs=blhs, brhs=brhs,
    intrule='GaussLegendre')
```

It remains to plot the solution (with high resolution in each element). To this end, we use the `u_glob` function imported from `fe1D`, which imports it from `fe_approx1D_numint` (the `u_glob` function in `fe_approx1D.py` works with `elements` and `nodes`, while `u_glob` in `fe_approx1D_numint` works with `cells`, `vertices`, and `dof_map`):

```
u_exact = lambda x: D + C*(x-L) + (1./6)*(L**3 - x**3)
from fe1D import u_glob
x, u, nodes = u_glob(c, cells, vertices, dof_map)
u_e = u_exact(x, C, D, L)
print(u_exact(nodes, C, D, L) - c) # difference at the nodes

import matplotlib.pyplot as plt
plt.plot(x, u, 'b-', x, u_e, 'r--')
plt.legend(['finite elements, d=%d %d, exact', loc='upper left')
plt.show()
```

The result is shown in Figure 6.2. We see that the solution using P1 elements is exact at the nodes, but feature considerable discrepancy

between the nodes. Exercise 6.6 asks you to explore this problem further using other m and d values.

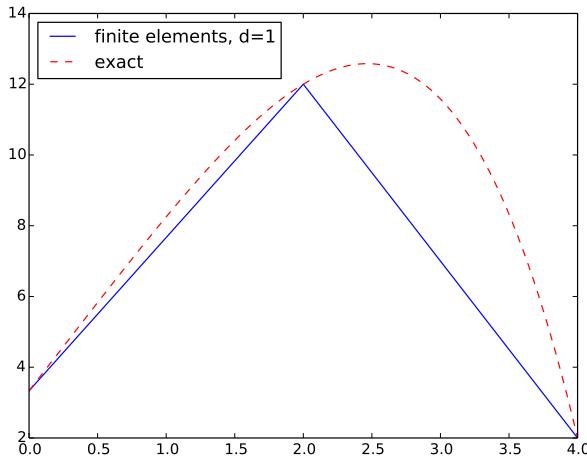


Fig. 6.2 Finite element and exact solution using two cells.

6.5 Variational formulations in 2D and 3D

The major difference between deriving variational formulations in 2D and 3D compared to 1D is the rule for integrating by parts. The cells have shapes different from an interval, so basis functions look a bit different, and there is a technical difference in actually calculating the integrals over cells. Otherwise, going to 2D and 3D is not a big step from 1D. All the fundamental ideas still apply.

6.5.1 Integration by parts

A typical second-order term in a PDE may be written in dimension-independent notation as

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

The explicit forms in a 2D problem become

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

and

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

We shall continue with the latter operator as the former arises from just setting $\alpha = 1$.

The integration by parts formula for $\int \nabla \cdot (\alpha \nabla)$

The general rule for integrating by parts is often referred to as Green's first identity:

$$-\int_{\Omega} \nabla \cdot (\alpha(\mathbf{x}) \nabla u) v \, dx = \int_{\Omega} \alpha(\mathbf{x}) \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} a \frac{\partial u}{\partial n} v \, ds, \quad (6.27)$$

where $\partial\Omega$ is the boundary of Ω and $\partial u / \partial n = \mathbf{n} \cdot \nabla u$ is the derivative of u in the outward normal direction, \mathbf{n} being an outward unit normal to $\partial\Omega$. The integrals $\int_{\Omega}()$ dx are area integrals in 2D and volume integrals in 3D, while $\int_{\partial\Omega}()$ ds is a line integral in 2D and a surface integral in 3D.

It will be convenient to divide the boundary into two parts:

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

The test functions v are (as usual) required to vanish on $\partial\Omega_D$.

6.5.2 Example on a multi-dimensional variational problem

Here is a quite general, stationary, linear PDE arising in many problems:

$$\mathbf{v} \cdot \nabla u + \beta u = \nabla \cdot (\alpha \nabla u) + f, \quad \mathbf{x} \in \Omega, \quad (6.28)$$

$$u = u_0, \quad \mathbf{x} \in \partial\Omega_D, \quad (6.29)$$

$$-\alpha \frac{\partial u}{\partial n} = g, \quad \mathbf{x} \in \partial\Omega_N. \quad (6.30)$$

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

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

Assume that the boundary function $u_0(\mathbf{x})$ is defined for all $\mathbf{x} \in \Omega$. The unknown function can then be expanded as

$$u = B + \sum_{j \in \mathcal{I}_s} c_j \psi_j, \quad B = u_0.$$

As long as any $\psi_j = 0$ on $\partial\Omega_D$, we realize that $u = u_0$ on $\partial\Omega_D$.

The variational formula is obtained from Galerkin's method, which technically means multiplying the PDE by a test function v and integrating over Ω :

$$\int_{\Omega} (\mathbf{v} \cdot \nabla u + \beta u) v \, dx = \int_{\Omega} \nabla \cdot (\alpha \nabla u) \, dx + \int_{\Omega} f v \, dx.$$

The second-order term is integrated by parts, according to the formula (6.27):

$$\int_{\Omega} \nabla \cdot (\alpha \nabla u) v \, dx = - \int_{\Omega} \alpha \nabla u \cdot \nabla v \, dx + \int_{\partial\Omega} \alpha \frac{\partial u}{\partial n} v \, ds.$$

Galerkin's method therefore leads to

$$\int_{\Omega} (\mathbf{v} \cdot \nabla u + \beta u) v \, dx = - \int_{\Omega} \alpha \nabla u \cdot \nabla v \, dx + \int_{\partial\Omega} \alpha \frac{\partial u}{\partial n} v \, ds + \int_{\Omega} f v \, dx.$$

The boundary term can be developed further by noticing that $v \neq 0$ only on $\partial\Omega_N$,

$$\int_{\partial\Omega} \alpha \frac{\partial u}{\partial n} v \, ds = \int_{\partial\Omega_N} \alpha \frac{\partial u}{\partial n} v \, ds,$$

and that on $\partial\Omega_N$, we have the condition $a \frac{\partial u}{\partial n} = -g$, so the term becomes

$$- \int_{\partial\Omega_N} g v \, ds.$$

The final variational form is then

$$\int_{\Omega} (\mathbf{v} \cdot \nabla u + \beta u) v \, dx = - \int_{\Omega} \alpha \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega_N} g v \, ds + \int_{\Omega} f v \, dx.$$

Instead of using the integral signs, we may use the inner product notation:

$$(\mathbf{v} \cdot \nabla u, v) + (\beta u, v) = -(\alpha \nabla u, \nabla v) - (g, v)_N + (f, v).$$

The subscript $_N$ in $(g, v)_N$ is a notation for a line or surface integral over $\partial\Omega_N$, while (\cdot, \cdot) is the area/volume integral over Ω .

We can derive explicit expressions for the linear system for $\{c_j\}_{j \in \mathcal{I}_s}$ that arises from the variational formulation. Inserting the u expansion results in

$$\begin{aligned} \sum_{j \in \mathcal{I}_s} ((\mathbf{v} \cdot \nabla \psi_j, \psi_i) + (\beta \psi_j, \psi_i) + (\alpha \nabla \psi_j, \nabla \psi_i)) c_j = \\ (g, \psi_i)_N + (f, \psi_i) - (\mathbf{v} \cdot \nabla u_0, \psi_i) + (\beta u_0, \psi_i) + (\alpha \nabla u_0, \nabla \psi_i). \end{aligned}$$

This is a linear system with matrix entries

$$A_{i,j} = (\mathbf{v} \cdot \nabla \psi_j, \psi_i) + (\beta \psi_j, \psi_i) + (\alpha \nabla \psi_j, \nabla \psi_i)$$

and right-hand side entries

$$b_i = (g, \psi_i)_N + (f, \psi_i) - (\mathbf{v} \cdot \nabla u_0, \psi_i) + (\beta u_0, \psi_i) + (\alpha \nabla u_0, \nabla \psi_i),$$

for $i, j \in \mathcal{I}_s$.

In the finite element method, we usually express u_0 in terms of basis functions and restrict i and j to run over the degrees of freedom that are not prescribed as Dirichlet conditions. However, we can also keep all the $\{c_j\}_{j \in \mathcal{I}_s}$ as unknowns, drop the u_0 in the expansion for u , and incorporate all the known c_j values in the linear system. This has been explained in detail in the 1D case, and the technique is the same for 2D and 3D problems.

6.5.3 Transformation to a reference cell in 2D and 3D

The real power of the finite element method first becomes evident when we want to solve partial differential equations posed on two- and three-dimensional domains of non-trivial geometric shape. As in 1D, the domain Ω is divided into N_e non-overlapping cells. The elements have simple shapes: triangles and quadrilaterals are popular in 2D, while tetrahe-

dra and box-shapes elements dominate in 3D. The finite element basis functions φ_i are, as in 1D, polynomials over each cell. The integrals in the variational formulation are, as in 1D, split into contributions from each cell, and these contributions are calculated by mapping a physical cell, expressed in physical coordinates \mathbf{x} , to a reference cell in a local coordinate system \mathbf{X} . This mapping will now be explained in detail.

We consider an integral of the type

$$\int_{\Omega^{(e)}} \alpha(\mathbf{x}) \nabla \varphi_i \cdot \nabla \varphi_j \, dx, \quad (6.31)$$

where the φ_i functions are finite element basis functions in 2D or 3D, defined in the physical domain. Suppose we want to calculate this integral over a reference cell, denoted by $\tilde{\Omega}^r$, in a coordinate system with coordinates $\mathbf{X} = (X_0, X_1)$ (2D) or $\mathbf{X} = (X_0, X_1, X_2)$ (3D). The mapping between a point \mathbf{X} in the reference coordinate system and the corresponding point \mathbf{x} in the physical coordinate system is given by a vector relation $\mathbf{x}(\mathbf{X})$. The corresponding Jacobian, J , of this mapping has entries

$$J_{i,j} = \frac{\partial x_j}{\partial X_i}.$$

The change of variables requires dx to be replaced by $\det J dX$. The derivatives in the ∇ operator in the variational form are with respect to \mathbf{x} , which we may denote by $\nabla_{\mathbf{x}}$. The $\varphi_i(\mathbf{x})$ functions in the integral are replaced by local basis functions $\tilde{\varphi}_r(\mathbf{X})$ so the integral features $\nabla_{\mathbf{x}} \tilde{\varphi}_r(\mathbf{X})$. We readily have $\nabla_{\mathbf{X}} \tilde{\varphi}_r(\mathbf{X})$ from formulas for the basis functions in the reference cell, but the desired quantity $\nabla_{\mathbf{x}} \tilde{\varphi}_r(\mathbf{X})$ requires some efforts to compute. All the details are provided below.

Let $i = q(e, r)$ and consider two space dimensions. By the chain rule,

$$\frac{\partial \tilde{\varphi}_r}{\partial X} = \frac{\partial \varphi_i}{\partial X} = \frac{\partial \varphi_i}{\partial x} \frac{\partial x}{\partial X} + \frac{\partial \varphi_i}{\partial y} \frac{\partial y}{\partial X},$$

and

$$\frac{\partial \tilde{\varphi}_r}{\partial Y} = \frac{\partial \varphi_i}{\partial Y} = \frac{\partial \varphi_i}{\partial x} \frac{\partial x}{\partial Y} + \frac{\partial \varphi_i}{\partial y} \frac{\partial y}{\partial Y}.$$

We can write these two equations as a vector equation

$$\begin{bmatrix} \frac{\partial \tilde{\varphi}_r}{\partial X} \\ \frac{\partial \tilde{\varphi}_r}{\partial Y} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial X} & \frac{\partial y}{\partial X} \\ \frac{\partial x}{\partial Y} & \frac{\partial y}{\partial Y} \end{bmatrix} \begin{bmatrix} \frac{\partial \varphi_i}{\partial x} \\ \frac{\partial \varphi_i}{\partial y} \end{bmatrix}$$

Identifying

$$\nabla_{\mathbf{X}} \tilde{\varphi}_r = \begin{bmatrix} \frac{\partial \tilde{\varphi}_r}{\partial X} \\ \frac{\partial \tilde{\varphi}_r}{\partial Y} \end{bmatrix}, \quad J = \begin{bmatrix} \frac{\partial x}{\partial X} & \frac{\partial y}{\partial X} \\ \frac{\partial x}{\partial Y} & \frac{\partial y}{\partial Y} \end{bmatrix}, \quad \nabla_{\mathbf{x}} \varphi_r = \begin{bmatrix} \frac{\partial \varphi_r}{\partial x} \\ \frac{\partial \varphi_r}{\partial y} \end{bmatrix},$$

we have the relation

$$\nabla_{\mathbf{X}} \tilde{\varphi}_r = J \cdot \nabla_{\mathbf{x}} \varphi_i,$$

which we can solve with respect to $\nabla_{\mathbf{x}} \varphi_i$:

$$\nabla_{\mathbf{x}} \varphi_i = J^{-1} \cdot \nabla_{\mathbf{X}} \tilde{\varphi}_r. \quad (6.32)$$

On the reference cell, $\varphi_i(\mathbf{x}) = \tilde{\varphi}_r(\mathbf{X})$, so

$$\nabla_{\mathbf{x}} \tilde{\varphi}_r(\mathbf{X}) = J^{-1}(\mathbf{X}) \cdot \nabla_{\mathbf{X}} \tilde{\varphi}_r(\mathbf{X}). \quad (6.33)$$

This means that we have the following transformation of the integral in the physical domain to its counterpart over the reference cell:

$$\int_{\Omega^{(e)}} \alpha(\mathbf{x}) \nabla_{\mathbf{x}} \varphi_i \cdot \nabla_{\mathbf{x}} \varphi_j \, d\mathbf{x} = \int_{\tilde{\Omega}^r} \alpha(\mathbf{X})(J^{-1} \cdot \nabla_{\mathbf{X}} \tilde{\varphi}_r) \cdot (J^{-1} \cdot \nabla_{\mathbf{X}} \tilde{\varphi}_s) \det J \, d\mathbf{X} \quad (6.34)$$

6.5.4 Numerical integration

Integrals are normally computed by numerical integration rules. For multi-dimensional cells, various families of rules exist. All of them are similar to what is shown in 1D: $\int f \, dx \approx \sum_j w_i f(\mathbf{x}_j)$, where w_j are weights and \mathbf{x}_j are corresponding points.

The file `numint.py` contains the functions `quadrature_for_triangles(n)` and `quadrature_for_tetrahedra(n)`, which returns lists of points and weights corresponding to integration rules with `n` points over the reference triangle with vertices $(0, 0)$, $(1, 0)$, $(0, 1)$, and the reference tetrahedron with vertices $(0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$, respectively. For example, the first two rules for integration over a triangle have 1 and 3 points:

```
>>> import numint
>>> x, w = numint.quadrature_for_triangles(num_points=1)
>>> x
[(0.3333333333333333, 0.3333333333333333)]
>>> w
[0.5]
>>> x, w = numint.quadrature_for_triangles(num_points=3)
```

```
>>> x
[(0.1666666666666666, 0.1666666666666666),
 (0.6666666666666666, 0.1666666666666666),
 (0.1666666666666666)]
>>> w
[0.1666666666666666, 0.1666666666666666, 0.1666666666666666]
```

Rules with 1, 3, 4, and 7 points over the triangle will exactly integrate polynomials of degree 1, 2, 3, and 4, respectively. In 3D, rules with 1, 4, 5, and 11 points over the tetrahedron will exactly integrate polynomials of degree 1, 2, 3, and 4, respectively.

6.5.5 Convenient formulas for P1 elements in 2D

We shall now provide some formulas for piecewise linear φ_I functions and their integrals *in the physical coordinate system*. These formulas make it convenient to compute with P1 elements without the need to work in the reference coordinate system and deal with mappings and Jacobians. A lot of computational and algorithmic details are hidden by this approach.

Let $\Omega^{(e)}$ be cell number e , and let the three vertices have global vertex numbers I , J , and K . The corresponding coordinates are (x_I, y_I) , (x_J, y_J) , and (x_K, y_K) . The basis function φ_I over $\Omega^{(e)}$ have the explicit formula

$$\varphi_I(x, y) = \frac{1}{2} \Delta (\alpha_I + \beta_I x + \gamma_I y), \quad (6.35)$$

where

$$\alpha_I = x_J y_K - x_K y_J, \quad (6.36)$$

$$\beta_I = y_J - y_K, \quad (6.37)$$

$$\gamma_I = x_K - x_J, \quad (6.38)$$

and

$$2\Delta = \det \begin{pmatrix} 1 & x_I & y_I \\ 1 & x_J & y_J \\ 1 & x_K & y_K \end{pmatrix}. \quad (6.39)$$

The quantity Δ is the area of the cell.

The following formula is often convenient when computing element matrices and vectors:

$$\int_{\Omega^{(e)}} \varphi_I^p \varphi_J^q \varphi_K^r dx dy = \frac{p! q! r!}{(p+q+r+2)!} 2\Delta. \quad (6.40)$$

(Note that the q in this formula is not to be mixed with the $q(e, r)$ mapping of degrees of freedom.)

As an example, the element matrix entry $\int_{\Omega^{(e)}} \varphi_I \varphi_J dx$ can be computed by setting $p = q = 1$ and $r = 0$, when $I \neq J$, yielding $\Delta/12$, and $p = 2$ and $q = r = 0$, when $I = J$, resulting in $\Delta/6$. We collect these numbers in a local element matrix:

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

The common element matrix entry $\int_{\Omega^{(e)}} \nabla \varphi_I \cdot \nabla \varphi_J dx$, arising from a Laplace term $\nabla^2 u$, can also easily be computed by the formulas above. We have

$$\nabla \varphi_I \cdot \nabla \varphi_J = \frac{\Delta^2}{4} (\beta_I \beta_J + \gamma_I \gamma_J) = \text{const},$$

so that the element matrix entry becomes $\frac{1}{4} \Delta^3 (\beta_I \beta_J + \gamma_I \gamma_J)$.

From an implementational point of view, one will work with local vertex numbers $r = 0, 1, 2$, parameterize the coefficients in the basis functions by r , and look up vertex coordinates through $q(e, r)$.

Similar formulas exist for integration of P1 elements in 3D.

6.5.6 A glimpse of the mathematical theory of the finite element method

Almost all books on the finite element method that introduces the abstract variational problem $a(u, v) = L(v)$ spend considerable pages on deriving error estimates and other properties of the approximate solution. The machinery with function spaces and bilinear and linear forms has the great advantage that a very large class of PDE problems can be analyzed in a unified way. This feature is often taken as an advantage of finite element methods over finite difference and volume methods. Since there are so many excellent textbooks on the mathematical properties of finite element methods [22, 5, 6, 13, 10, 26], this text will not repeat the theory, but give a glimpse of typical assumptions and general results for elliptic PDEs.

Remark. The mathematical theory of finite element methods is primarily developed for stationary PDE problems of elliptic nature whose solutions are smooth. However, such problems can be solved with the desired accuracy by most numerical methods and pose no difficulties. Time-dependent problems, on the other hand, easily lead to non-physical features in the numerical solutions and therefore require more care and knowledge by the user. Our focus on the accuracy of the finite element method will of this reason be centered around time-dependent problems, but then we need a different set of tools for the analysis. These tools are based on converting finite element equations to finite difference form and studying Fourier wave components.

Abstract variational forms. To list the main results from the mathematical theory of finite elements, we consider linear PDEs with an abstract variational form

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

This is the discretized problem (as usual in this book) where we seek $u \in V$. The weak formulation of the corresponding continuous problem, fulfilled by the exact solution $u_e \in V_e$ is here written as

$$a(u_e, v) = L(v) \quad \forall v \in V_e.$$

The space V is finite dimensional (with dimension $N + 1$), while V_e is infinite dimensional. Normally the hope is that $u \rightarrow u_e$ as $N \rightarrow \infty$ and $V \rightarrow V_e$.

Example on an abstract variational form and associated spaces. Consider the problem $-u''(x) = f(x)$ on $\Omega = [0, 1]$, with $u(0) = 0$ and $u'(1) = \beta$. The weak form is

$$a(u, v) = \int_0^1 u'v' dx, \quad L(v) = \int_0^1 fv dx + \beta v(1).$$

The space V for the approximate solution u can be chosen in many ways as previously described. The exact solution u_e fulfills $a(u, v) = L(v)$ for all v in V_e , and to specify what V_e is, we need to introduce *Hilbert spaces*. The Hilbert space $L^2(\Omega)$ consists of all functions that are square-integrable on Ω :

$$L^2(\Omega) = \left\{ \int_{\Omega} v^2 dx < \infty \right\}.$$

The space V_e is the space of all functions whose first-order derivative is also square-integrable:

$$V_e = H_0^1(\Omega) = \left\{ v \in L^2(\Omega) \mid \frac{dv}{dx} \in L^2(\Omega), \text{ and } v(0) = 0 \right\}.$$

The requirements of square-integrable zeroth- and first-order derivatives are motivated from the formula for $a(u, v)$ where products of the first-order derivatives are to be integrated on Ω . We remark that it is common that H_0^1 denote the space of H^1 functions that are zero everywhere on the boundary, but here we use it for functions that are zero only at $x = 0$.

The Sobolev space $H_0^1(\Omega)$ has an inner product

$$(u, v)_{H^1} = \int_{\Omega} (uv + \frac{du}{dx} \frac{dv}{dx}) dx,$$

and associated norm

$$\|v\|_{H^1} = \sqrt{(v, v)_{H^1}}.$$

Assumptions. A set of general results builds on the following assumptions. Let V_e be an infinite-dimensional inner-product space such that $u_e \in V_e$. The space has an associated norm $\|v\|$ (e.g., $\|v\|_{H^1}$ in the example above with $V_e = H_0^1(\Omega)$).

1. $L(v)$ is linear in its argument.
2. $a(u, v)$ is a bilinear in its arguments.
3. $L(v)$ is bounded (also called continuous) if there exists a positive constant c_0 such that $|L(v)| \leq c_0 \|v\| \forall v \in V_e$.
4. $a(u, v)$ is bounded (or continuous) if there exists a positive constant c_1 such that $|a(u, v)| \leq c_1 \|u\| \|v\| \forall u, v \in V_e$.
5. $a(u, v)$ is elliptic (or coercive) if there exists a positive constant c_2 such that $a(v, v) \geq c_2 \|v\|^2 \forall v \in V_e$.
6. $a(u, v)$ is symmetric: $a(u, v) = a(v, u)$.

Based on the above assumptions, which must be verified in each specific problem, one can derive some general results that are listed below.

Existence and uniqueness. There exists a unique solution of the problem: find $u_e \in V_e$ such that

$$a(u_e, v) = L(v) \quad \forall v \in V_e.$$

(This result is known as the Lax-Milgram Theorem. We remark that symmetry is not strictly needed for this theorem.)

Stability. The solution $u_e \in V_e$ obeys the stability estimate

$$\|u\| \leq \frac{c_0}{c_2}.$$

Equivalent minimization problem. The solution $u_e \in V_e$ also fulfills the minimization problem

$$\min_{v \in V_e} F(v), \quad F(v) = \frac{1}{2}a(v, v) - L(v).$$

Best approximation principle. The *energy norm* is defined as

$$\|v\|_a = \sqrt{a(v, v)}.$$

The discrete solution $u \in V$ is the best approximation in energy norm,

$$\|u_e - u\|_a \leq \|u_e - v\|_a \quad \forall v \in V.$$

This is quite remarkable: once we have V (i.e., a mesh and a finite element), the Galerkin method finds the best approximation in this space. In the example above, we have $\|v\|_a = \int_0^1 (v')^2 dx$, so the derivative u' is closer to u'_e than any other possible function in V :

$$\int_0^1 (u'_e - u')^2 dx \leq \int_0^1 (u' - v')^2 dx \quad \forall v \in V.$$

Best approximation property in the norm of the space. If $\|v\|$ is the norm associated with V_e , we have another best approximation property:

$$\|u_e - u\| \leq \left(\frac{c_1}{c_2}\right)^{\frac{1}{2}} \|u_e - v\| \quad \forall v \in V.$$

Symmetric, positive definite coefficient matrix. The discrete problem $a(u, v) = L(v) \quad \forall v \in V$ leads to a linear system $Ac = b$, where the coefficient matrix A is symmetric ($A^T = A$) and positive definite ($x^T Ax > 0$ for all vectors $x \neq 0$). One can then use solution methods that demand less storage and that are faster and more reliable than solvers for general linear systems. One is also guaranteed the existence and uniqueness of the discrete solution u .

Equivalent matrix minimization problem. The solution c of the linear system $Ac = b$ also solves the minimization problem $\min_w (\frac{1}{2}w^T Aw - b^T w)$ in the vector space \mathbb{R}^{N+1} .

A priori error estimate for the derivative. In our sample problem, $-u'' = f$ on $\Omega = [0, 1]$, $u(0) = 0$, $u'(1) = \beta$, one can derive the following error estimate for Lagrange finite element approximations of degree s :

$$\left(\int_0^1 (u'_e - u')^2 dx \right)^{\frac{1}{2}} \leq Ch^s \|u_e\|_{H^{s+1}},$$

where $\|u\|_{H^{s+1}}$ is a norm that integrates the sum of the square of all derivatives up to order $s + 1$, C is a constant, and h is the maximum cell length. The estimate shows that choosing elements with higher-degree polynomials (large s) requires more smoothness in u_e since higher-order derivatives need to be square-integrable.

A consequence of the error estimate is that $u' \rightarrow u'_e$ as $h \rightarrow 0$, i.e., the approximate solution converges to the exact one.

The constant C depends on the shape of triangles in 2D and tetrahedra in 3D: squeezed elements with a small angle lead to a large C , and such deformed elements are not favorable for the accuracy.

One can generalize the above estimate to the general problem class $a(u, v) = L(v)$: the error in the derivative is proportional to h^s . Note that the expression $\|u_e - u\|$ in the example is $\|u_e - u\|_{H^1}$ so it involves the sum of the zeroth and first derivative. The appearance of the derivative makes the error proportional to h^s - if we only look at the solution it converges as h^{s+1} (see below).

The above estimate is called an *a priori* estimate because the bound contains the exact solution, which is not computable. There are also *a posteriori* estimates where the bound involves the approximation u , which is available in computations.

A priori error estimate for the solution. The finite element solution of our sample problem fulfills

$$\|u_e - u\| \leq Ch^{s+1} \|u_e\|_{H^{s+1}},$$

This estimate shows that the error converges as h^2 for P1 elements. An equivalent finite difference method, see Section 6.1.3, is known to have an error proportional to h^2 , so the above estimate is expected. In general, the convergence is h^{s+1} for elements with polynomials of degree s . Note that the estimate for u' is proportional to h raised to one power less. We remark that the second estimate strictly speaking requires extra smoothness (regularity).

6.6 Implementation in 2D and 3D via FEniCS

From a principle of view, we have seen that variational forms of the type: find $a(u, v) = L \forall v \in V$ (and even general nonlinear problems $F(u; v) = 0$), can apply the computational machinery of introduced for the approximation problem $u = f$. We actually need two extensions only:

1. specify Dirichlet boundary conditions as part of V
2. incorporate Neumann flux boundary conditions in the variational form

The algorithms are all the same in any space dimension, we only need to choose the element type and associated integration rule. Once we know how to compute things in 1D, and made the computer code sufficiently flexible, the method and code should work for any variational form in any number of space dimensions! This fact is exactly the idea behind the FEniCS finite element software.

Therefore, if we know how to set up an approximation problem in any dimension in FEniCS, and know how to derive variational forms in higher dimensions, we are (in principle!) very close to solving a PDE problem in FEniCS. Building on the Section 4.7, we shall now solve a quite general 1D/2D/3D Poisson problem in FEniCS. There is much more to FEniCS than what is shown in this example, but it illustrates the fact that when we go beyond 1D, there exists software which leverage the full power of the finite element method as a method for solving any problem on any mesh in any number of space dimensions.

6.6.1 Mathematical problem

The following model describes the pressure u in the flow around a bore hole of radius a in a porous medium. If the hole is long in the vertical direction (z-direction) then it is natural to assume that the vertical changes are small and $u_z \approx \text{constant}$. Therefore, we can model it by a 2D domain in the cross section.

$$\nabla \cdot (\alpha \nabla u) = 0, \quad a < \|\boldsymbol{x}\| < b, \quad (6.41)$$

$$u(\boldsymbol{x}) = U_a, \quad \|\boldsymbol{x}\| = a, \quad (6.42)$$

$$u(\boldsymbol{x}) = U_b, \quad \|\boldsymbol{x}\| = b. \quad (6.43)$$

That is, we have a hollow circular 2D domain with inner radius a and outer radius b . The pressure is known on these two boundaries, so this is a pure Dirichlet problem.

Symmetry. The first thing we should observe is that the problem is radially symmetric, so we can change to polar coordinates and obtain a 1D problem in the radial direction:

$$(r\alpha u')' = 0, \quad u(a) = U_a, u(b) = U_b.$$

This is not very exciting beyond being able to find an analytical solution and compute the true error of a finite element approximation.

However, many software packages solve problems in Cartesian coordinates, and FEniCS basically do this, so we want to take advantage of symmetry in Cartesian coordinates and reformulate the problem in a smaller domain.

Looking at the domain as a cake with a hole, any piece of the cake will be a candidate for a reduced-size domain. The solution is symmetric about any line $\theta = \text{const}$ in polar coordinates, so at such lines we have the symmetry boundary condition $\partial u / \partial n = 0$, i.e., a homogeneous Neumann condition. In Figure 6.3 we have plotted a possible mesh of cells as triangles, here with dense refinement toward the bore hole, because we know the solution will decay most rapidly toward the origin. This mesh is a piece of the cake with four sides: Dirichlet conditions on the inner and outer boundary, named Γ_{D_a} and Γ_{D_b} , and $\partial u / \partial n = 0$ on the two other sides, named Γ_N . In this particular example, the arc of the piece of the cake is 45 degrees, but any value of the arc will work.

The boundary problem can then be expressed as

$$\nabla \cdot (\alpha \nabla u) = 0, \quad \mathbf{x} \in \Omega, \quad (6.44)$$

$$u(\mathbf{x}) = U_a, \quad \mathbf{x} \in \Gamma_{D_a}, \quad (6.45)$$

$$u(\mathbf{x}) = U_b, \quad \mathbf{x} \in \Gamma_{D_b}, \quad (6.46)$$

$$\frac{\partial u}{\partial n} = 0, \quad \mathbf{x} \in \Gamma_N. \quad (6.47)$$

6.6.2 Variational formulation

To obtain the variational formulation, we multiply the PDE by a test function v and integrate the second-order derivatives by part:

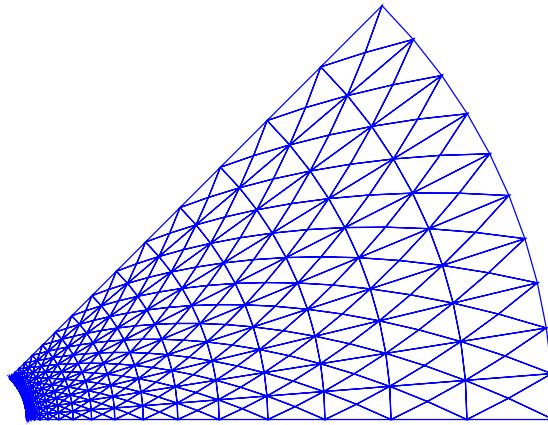


Fig. 6.3 Mesh of a hollow cylinder, with refinement and utilizing symmetry.

$$\begin{aligned} \int_{\Omega} \nabla \cdot (\alpha \nabla u) v \, dx &= - \int_{\Omega} \alpha \nabla u \cdot \nabla v \, dx + \int_{\Gamma_N} \alpha \frac{\partial u}{\partial n} v \, ds \\ &= - \int_{\Omega} \alpha \nabla u \cdot \nabla v \, dx, \quad \forall v \in V \end{aligned}$$

We are left with a problem of the form: find u such that $a(u, v) = L(v) \quad \forall v \in V$, with

$$a(u, v) = \int_{\Omega} \alpha \nabla u \cdot \nabla v \, dx, \tag{6.48}$$

$$L(v) = \int_{\Omega} 0v \, dx. \tag{6.49}$$

We write the integrand as $0v \, dx$ even though $L = 0$, because it is necessary in FEniCS to specify L as a linear form (i.e., a test function and some

form of integration need to be present) and not the number zero. The Dirichlet conditions make a nonzero solution.

6.6.3 The FEniCS solver

We suppose that we have a function `make_mesh` that can make the mesh for us. More details about this function will be provided later. A next step is then to define proper Dirichlet conditions. This might seem a bit complicated, but we introduce *markers* at the boundary for marking the Dirichlet boundaries. The inner boundary has marker 1 and the outer has marker 2. In this way, we can recognize the nodes that are on the boundary. It is usually a part of the mesh making process to compute both the mesh and its markers, so `make_mesh` returns a `Mesh` object as `mesh` and a `MeshFunction` object `markers`. Setting Dirichlet conditions in the solver is then a matter of introducing `DirichletBC` objects, one for each part of the boundary marked by `markers`, and then we collect all such Dirichlet conditions in a list that is used by the assembly process to incorporate the Dirichlet conditions in the linear system. The code goes like this:

```
V = FunctionSpace(mesh, 'P', degree)
bc_inner = DirichletBC(V, u_a, markers, 1)
bc_outer = DirichletBC(V, u_b, markers, 2)
bcs = [bc_inner, bc_outer]
```

Here, `u_a` and `u_b` are constants (floats) set by the user. In general, anything that can be evaluated pointwise can be used, such as `Expression`, `Function`, and `Constant`. The next step is to define the variational problem and solve it:

```
# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
a = alpha*dot(grad(u), grad(v))*dx
L = Constant(0)*v*dx # L = 0*v*dx = 0 does not work...

# Compute solution
u = Function(V)
solve(a == L, u, bcs)

f = File("mesh.xml")
f << mesh
```

In order to avoid `L=0` (`L` equal to the float zero), we have to tell FEniCS that is a linear form, so zero must be specified as `Constant(0)`.

Note that everything is the same as for the approximation problem in Section 4.7, except for the Dirichlet conditions and the formulas for a and L . FEniCS has, of course, access to very efficient solution methods, so we could add arguments to the `solve` call to apply state-of-the-art iterative methods and preconditioners for large-scale problems. However, for this little 2D case a standard sparse Gaussian elimination, as implied by `solve(a = L, u, bcs)` is a sufficient approach.

Finally, we can save the solution to file for using professional visualization software and, if desired, add a quick plotting using the built-in FEniCS tool `plot`:

```
# Save solution to file in VTK format
vtkfile = File(filename + '.pvd')
vtkfile << u

u.rename('u', 'u'); plot(u); plot(mesh)
import matplotlib.pyplot as plt
plt.show()
```

(The `u.rename` call is just for getting a more readable title in the plot.)

The above statements are collected in a function `solver` in the file `borehole_fenics.py`:

```
def solver(
    mesh,
    markers, # MeshFunctions for Dirichlet conditions
    alpha,   # Diffusion coefficient
    u_a,     # Inner pressure
    u_b,     # Outer pressure
    degree,  # Element polynomial degree
    filename, # Name of VTK file
):
    V = FunctionSpace(mesh, 'P', degree)
    bc_inner = DirichletBC(V, u_a, markers, 1)
    bc_outer = DirichletBC(V, u_b, markers, 2)
    bcs = [bc_inner, bc_outer]

    # Define variational problem
    u = TrialFunction(V)
    v = TestFunction(V)
    a = alpha*dot(grad(u), grad(v))*dx
    L = Constant(0)*v*dx # L = 0*v*dx = 0 does not work...

    # Compute solution
    u = Function(V)
    solve(a == L, u, bcs)

    f = File("mesh.xml")
    f << mesh
```

```

# Save solution to file in VTK format
vtkfile = File(filename + '.pvd')
vtkfile << u

u.rename('u', 'u'); plot(u); plot(mesh)
import matplotlib.pyplot as plt
plt.show()
return u

def problem():
    mesh, markers = make_mesh(Theta=25*pi/180, a=1, b=2,
                               nr=20, nt=20, s=1.9)
    beta = 5
    solver(mesh, markers, alpha=1, u_a=1, u_b=0, degree=1, filename='tmp')

if __name__ == '__main__':
    problem()

```

Be careful with name clashes!

It is easy when coding mathematics to use variable names that correspond to one-letter names in the mathematics. For example, in the mathematics of this problem there are two a variables: the radius of the inner boundary and the bilinear form in the variational formulation. Using `a` for the inner boundary in `solver` does not work: it is quickly overwritten by the bilinear form. We therefore have to introduce `x_a`. Long variable names are to be preferred for safe programming, though short names corresponding to the mathematics are often nicer.

6.6.4 Making the mesh

The hardest part of a finite element problem is very often to make the mesh. This is particularly the case in large industrial projects, but also often academic projects quickly lead to time-consuming work with constructing finite element meshes. In the present example we create the mesh for the symmetric problem by deforming an originally rectangular mesh. The rectangular mesh is made by the FEniCS object `RectangleMesh` on $[a, b] \times [0, 1]$. Therefore, we stretch the mesh towards the left before we bend the rectangle onto to “a piece of cake”. Figure 6.4 shows an example on the resulting mesh. The stretching gives us refine-

ment in the radial direction because we expect the variations to be quite large in this direction, but uniform in θ direction.

We first make the rectangle and set boundary markers here for the inner and outer boundaries (since these are particularly simple: $x = a$ and $x = b$). Here is how we make the rectangular mesh from lower left corner $(a, 0)$ to upper left corner $(b, 1)$ with `nr` quadrilateral cells in x direction (later to become the radial direction) and `nt` quadrilateral cells in the y direction:

```
mesh = RectangleMesh(Point(a, 0), Point(b, 1), nr, nt, 'crossed')
```

Each quadrilateral cell is divided into two triangles with right or left going diagonals, or four triangles using both diagonals. These choices of producing triangles from rectangles are named `right`, `left`, and `crossed`. Recall that FEniCS can only work with cells of triangular shape only and where the sides are straight. This means that we need a good resolution in θ direction to represent a circular boundary. With isoparametric elements, it is easier to get a higher-order polynomial approximation of curved boundaries.

We must then mark the boundaries for boundary conditions. Since we do not need to do anything with the homogeneous Neumann conditions, we can just mark the inner and outer boundary of the hole cylinder. This is very easy to do as long as the mesh is a rectangle since then the specifications of the boundaries are $x = a$ and $x = b$. The relevant FEniCS code requires the user to define a subclass of `SubDomain` and implement a function `inside` for indicating whether a given point `x` is on the desired boundary or not:

```
# x=a becomes the inner borehole boundary
class Inner(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and abs(x[0] - a) < tol

# x=b becomes the outer borehole boundary
class Outer(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and abs(x[0] - b) < tol

inner = Inner(); outer = Outer();
markers = MeshFunction('size_t', mesh, mesh.topology().dim() - 1)
markers.set_all(0)
inner.mark(markers, 1)
outer.mark(markers, 2)
```

With the instances `inner` and `outer` we fill a marker object, called `MeshFunction` in FEniCS. For this purpose we must introduce our own

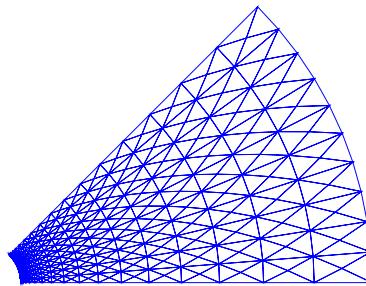


Fig. 6.4 Finite element mesh for a porous medium outside a bore hole (hollow cylinder).

convention of numbering boundaries: here we use 1 for all points on the inner boundary and 2 for the outer boundary, while all other points are marked by 0. The solver applies the `markers` object to set the right Dirichlet boundary conditions.

The next step is to deform the mesh. Given coordinates x , we can map these onto a stretched coordinate \bar{x} by

$$\bar{x} = a + (b - a) \left(\frac{x - a}{b - a} \right)^s, \quad (6.50)$$

where s is a parameter that controls the amount of stretching. The formula above gives a stretching towards $x = a$, while the next one stretches the coordinates towards $x = b$:

$$\bar{x} = a + (b - a) \left(\frac{x - a}{b - a} \right)^{1/s}. \quad (6.51)$$

The code shown later shows the details of mapping coordinates in a FEniCS mesh object.

The final step is to map the rectangle to a part of a hollow cylinder. Mathematically, a point (x, y) in the rectangle is mapped onto (\bar{x}, \bar{y}) in our final geometry:

$$\hat{x} = \bar{x} \cos(\Theta \bar{y}), \quad \hat{y} = \bar{x} \sin(\Theta \bar{y}).$$

The relevant FEniCS code becomes

```

# --- Deform mesh ---

# First make a denser mesh towards r=a
x = mesh.coordinates()[:,0]
y = mesh.coordinates()[:,1]

def denser(x, y):
    return [a + (b-a)*((x-a)/(b-a))**s, y]

x_bar, y_bar = denser(x, y)
xy_bar_coor = np.array([x_bar, y_bar]).transpose()
mesh.coordinates()[:] = xy_bar_coor

# Then map onto to a "piece of cake"

def cylinder(r, s):
    return [r*np.cos(Theta*s), r*np.sin(Theta*s)]

x_hat, y_hat = cylinder(x_bar, y_bar)
xy_hat_coor = np.array([x_hat, y_hat]).transpose()
mesh.coordinates()[:] = xy_hat_coor
return mesh, markers

```

Fortunately, the solver is independent of all the details of the mesh making. We could also have used the mesh tool `mshr` in FEniCS, but with our approach here we have full control of the refinement towards the hole.

6.6.5 Solving a problem

We assume that α is constant. Before solving such a specific problem, it can be wise to scale the problem since it often reduces the amount of input data in the model. Here, the variation in u is typically $|u_a - u_b|$ so we use that as characteristic pressure. The coordinates may be naturally scaled by the bore hole radius, so we have new, scaled variables

$$\bar{u} = \frac{u - u_a}{u_a - u_b}, \quad \bar{x} = \frac{x}{a}, \quad \bar{y} = \frac{y}{a}.$$

Now, we expect $\bar{u} \in [0, 1]$, which is a goal of scaling. Inserting this in the problem gives the PDE

$$\nabla^2 \bar{u} = 0$$

in a domain with inner radius 1 and $\bar{u} = 0$, and outer radius

$$\beta = \frac{a}{b},$$

with $\bar{u} = 1$. Our solver can solve this problem by setting `alpha=1`, `u_a=1`, and `u_b=0`. We see that the dimensionless parameter β goes to the mesh and not to the solver. Figure 6.5 shows a solution for $\beta = 2$ on a mesh with $4 \cdot 20 \cdot 20 = 1600$ triangles, 25 degree opening, and P1 elements. Switching to higher-order, say P3, is a matter of changing the `degree` parameter that goes to the function `V` in the solver:

```
mesh, markers = make_mesh(Theta=25*pi/180, a=1, b=2,
                         nr=20, nt=20, s=1.9)

beta = 2
solver(mesh, markers,
       alpha=1, u_a=1, u_b=0, degree=3, filename='borehole1')
```

The complete code is found in `borehole_fenics.py`. All fluids flow in the same way as long as the geometry is the same!

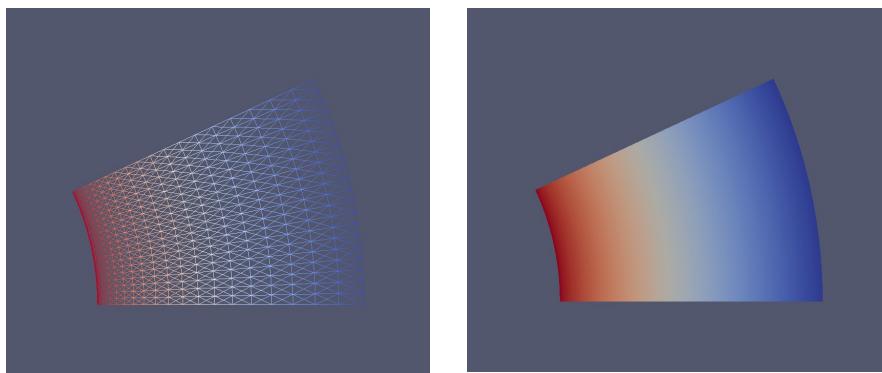


Fig. 6.5 Solution for (scaled) fluid pressure around a bore hole in a porous medium.

How can we solve a 3D version of this problem? Then we would make a long cylinder. The assumption is that nothing changes in the third direction, so $\partial/\partial z = 0$. This means that the cross sections at the end of the cylinder have homogeneous Neumann conditions $\partial u/\partial n = 0$. Therefore, nothing changes in the variational form. Actually, all we have to do is to generate a 3D box and use the same stretching and mapping to make the cylinder, and run the solver without changes!

6.7 Convection-diffusion and Petrov-Galerkin methods

Let us now return to the convection-diffusion problem introduced in Section 6.5.2. The problem in general reads,

$$\mathbf{v} \cdot \nabla u + \beta u = \nabla \cdot (\alpha \nabla u) + f, \quad \mathbf{x} \in \Omega, \quad (6.52)$$

$$u = u_0, \quad \mathbf{x} \in \partial\Omega_D, \quad (6.53)$$

$$-\alpha \frac{\partial u}{\partial n} = g, \quad \mathbf{x} \in \partial\Omega_N. \quad (6.54)$$

In Section 5.5 we investigated the simplified case in 1D

$$\begin{aligned} -u_x - \alpha u_{xx} &= 0, \\ u(0) = 0, u(1) &= 1. \end{aligned}$$

and the analytical solution was:

$$u_e(x) = \frac{e^{-x/\alpha} - 1}{e^{-1/\alpha} - 1}.$$

The approximation with global functions failed when α was significantly smaller than ν . The computed solution contained non-physical oscillations that were orders of magnitude larger than the true solution. The approximation did however improve as the number of degrees of freedom increased.

The variational problem is: Find $u \in V$ such that

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

where

$$\begin{aligned} a(u, v) &= \int_0^1 -u_x v + \mu u_x v_x \, dx, \\ L(v) &= \int_0^1 0v \, dx = 0. \end{aligned}$$

A Galerkin approximation with a finite element approximation is obtained by letting $u = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)$ and $v = \psi_i(x)$ which leads to a linear system of equations $\sum_j A_{i,j} c_j = b_i$ where

$$\begin{aligned} A_{i,j} &= \int_0^1 \mu \psi'_i \psi'_j + \psi'_i \psi_j \, dx, \\ b_i &= \int_0^1 0 \psi_i \, dx. \end{aligned}$$

Figure 6.6 shows the finite element solution on a coarse mesh of 10 elements for $\mu = 0.01$. Clearly, the finite element method has the same problem as was observed earlier in global functions in Section 6.5.2.

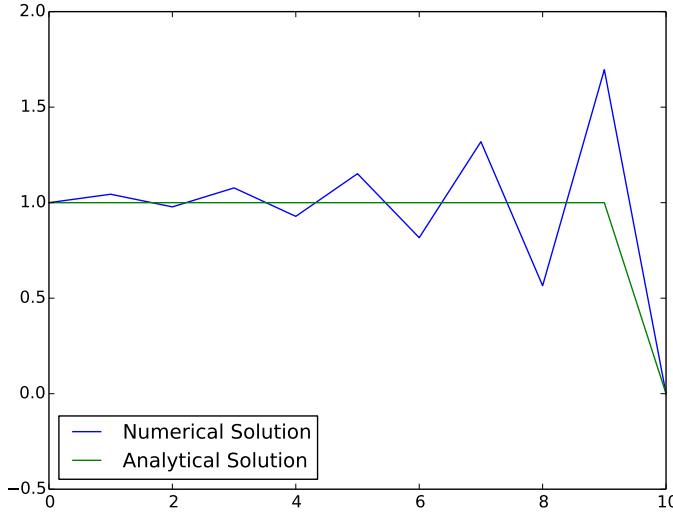


Fig. 6.6 Solution obtained with Galerkin approximation using linear elements $N_e = 10$ and $\mu = 0.01$.

For finite differences, the cure for these oscillations is *upwinding*. That is, instead of using a central difference scheme, we employ the following difference scheme:

$$\begin{aligned}\frac{du}{dx} \Big|_{x=x_i} &= \frac{1}{h}[u_{i+1} - u_i] && \text{if } v < 0, \\ \frac{du}{dx} \Big|_{x=x_i} &= \frac{1}{h}[u_i - u_{i-1}] && \text{if } v > 0.\end{aligned}$$

Using this scheme, oscillations will disappear as can be seen in Figure 6.7.

There is a relationship between upwinding and artificial diffusion. If we discretize u_x with a central difference and add diffusion as $\epsilon = h/2\Delta$ we get

$$\begin{aligned}
 & \frac{u_{i+1} - u_{i-1}}{2h} && \text{central scheme, first order derivative} \\
 & + \frac{h}{2} \frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} && \text{central scheme, second order derivative} \\
 & = \frac{u_i - u_{i-1}}{h} && \text{upwind scheme}
 \end{aligned}$$

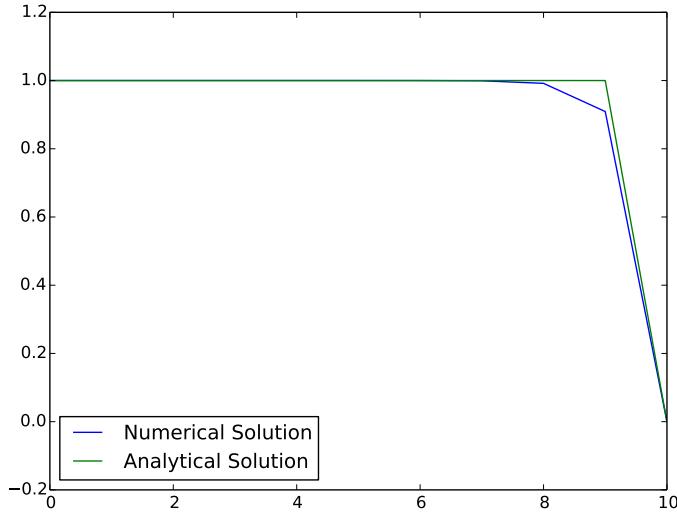


Fig. 6.7 Solution obtained upwinding $N_e = 10$ and $\mu = 0.01$.

Hence, upwinding is equivalent to adding artificial diffusion with $\epsilon = h/2$; that is, in both cases we actually solve the problem

$$-(\mu + \epsilon)u_{xx} + vu_x = f.$$

using a central difference scheme.

Finite difference upwinding is difficult to express using finite elements methods, but it is closely related to adding some kind of diffusion to the scheme. A clever method of adding diffusion is the so-called *Petrov-Galerkin* method. The Galerkin approximation consists of finding $u \in V$ such that for all $v \in V$ the variational formulation is satisfied.

$$\begin{aligned} a(u, v) &= \int_0^1 -u_x v + \mu u_x v_x \, dx, \\ L(v) &= \int_0^1 0v \, dx = 0. \end{aligned}$$

The Petrov-Galerkin method is a seemingly innocent and straightforward extension of Galerkin where we want to find $u \in V$ such that for all $w \in W$ the variational formulation is fulfilled.

$$\begin{aligned} a(u, w) &= \int_0^1 -u_x w + \mu u_x w_x \, dx, \\ L(w) &= \int_0^1 0w \, dx = 0. \end{aligned}$$

W can in principle be chosen freely, but in general we wish to obtain a quadratic matrix and hence the number of basis functions in V and W should be the same. Let $w = v + \beta h \mathbf{v} \cdot \nabla v$. In our simplified 1D case we had $\mathbf{v} = (1, 0)$, h is the element size and β is a tunable parameter.

$$\begin{aligned} A_{ij} &= a(\psi_i, \psi_j + \beta v \cdot \nabla \psi_j) \\ &= \underbrace{\int_{\Omega} \mu \nabla \psi_i \cdot \nabla (\psi_j + \beta \mathbf{v} \cdot \nabla \psi_j) \, dx}_{\text{standard Galerkin}} + \int_{\Omega} \mathbf{v} \nabla \psi_i \cdot (\psi_j + \beta v \cdot \nabla \psi_j) \, dx \\ &= \underbrace{\int_{\Omega} \mu \nabla \psi_i \cdot \nabla \psi_j \, dx}_{=0 \text{ for linear elements}} + \underbrace{\int_{\Omega} \mathbf{v} \cdot \nabla \psi_i \psi_j \, dx}_{\text{artificial diffusion in } \mathbf{v} \text{ direction}} \\ &\quad + \beta \underbrace{\int_{\Omega} \mu \nabla \psi_i \cdot \nabla (\mathbf{v} \cdot \nabla \psi_j) \, dx}_{\text{artificial diffusion in } \mathbf{v} \text{ direction}} + \beta \underbrace{\int_{\Omega} (v \cdot \nabla \psi_i)(\mathbf{v} \cdot \nabla \psi_j) \, dx}_{\text{artificial diffusion in } \mathbf{v} \text{ direction}} \end{aligned}$$

In general also the right hand side is altered, given a source term f then

$$b_i = L(\psi_i) = \int_{\Omega} f(\psi_i + \beta \mathbf{v} \cdot \nabla \psi_i) \, dx = 0.$$

The modification of the right-hand side is important as a strongly consistent scheme is obtained. In fact, since it is a Petrov-Galerkin projection, it can be shown that the truncation error (see 5.1.7) is zero regardless of the mesh size h .

The corresponding FEniCS code is

```
from fenics import *
```

```

import matplotlib.pyplot as plt

def boundary(x):
    return x[0] < 1E-15 or x[0] > 1.0 - 1E-15

u_analytical = \
    Expression("(exp(-x[0]/%e) - 1)/ (exp(-1/%e) - 1)" % \
    (alpha_value, alpha_value), degree=2)

mesh = UnitIntervalMesh(10)
V = FunctionSpace(mesh, "CG", 1)
u = TrialFunction(V)
v = TestFunction(V)

alpha_value = 1.0e-2
alpha = Constant(alpha_value)
beta_value = 0.5
beta = Constant(beta_value)

f = Constant(0)
h = mesh.hmin()
v = v - beta*h*v.dx(0) #Petrov-Galerkin

a = (-u.dx(0)*v + alpha*u.dx(0)*v.dx(0) + beta*h*u.dx(0)*v.dx(0))*dx
L = f*v*dx

bc = DirichletBC(V, u_analytical, boundary)
u = Function(V)
solve(a == L, u, bc)

```

Notice that the Petrov-Galerkin method is here implemented as $v = v - \beta * h * v.dx(0)$. Furthermore, we represent α and β as constants rather than floats in Python because we then avoid re-compilation every time we change these parameters.

The Petrov-Galerkin method is consistent (the weighted residual is zero) because it is a projection method. The projection is taken onto the space W rather than V and this may in particular on coarse meshes have a dramatic effect. On coarse meshes the numerical solutions obtained is usually a lot better than the Galerkin approach because the method makes the boundary layer smoother. However, on fine meshes that is able to resolve the boundary layer, the approximation is often better with a standard Galerkin approach. There are many extensions of the method, but it still remains a problem to find a proper and robust method for the convection-diffusion problem where unphysical oscillations do not show up and the boundary layer is not artificially smoothed.

6.8 Summary

- When approximating f by $u = \sum_j c_j \varphi_j$, the least squares method and the Galerkin/projection method give the same result. The interpolation/collocation method is simpler and yields different (mostly inferior) results.
- Fourier series expansion can be viewed as a least squares or Galerkin approximation procedure with sine and cosine functions.
- Basis functions should optimally be orthogonal or almost orthogonal, because this makes the coefficient matrix become diagonal or sparse and results in little round-off errors when solving the linear system. One way to obtain almost orthogonal basis functions is to localize the basis by making the basis functions that have local support in only a few cells of a mesh. This is utilized in the finite element method.
- Finite element basis functions are *piecewise* polynomials, normally with discontinuous derivatives at the cell boundaries. The basis functions overlap very little, leading to stable and efficient numerics involving sparse matrices.
- To use the finite element method for differential equations, we use the Galerkin method or the method of weighted residuals to arrive at a variational form. Technically, the differential equation is multiplied by a test function and integrated over the domain. Second-order derivatives are integrated by parts to allow for typical finite element basis functions that have discontinuous derivatives.
- The least squares method is not much used for finite element solution of differential equations of second order, because it then involves second-order derivatives which cause trouble for basis functions with discontinuous derivatives.
- We have worked with two common finite element terminologies and associated data structures (both are much used, especially the first one, while the other is more general):
 1. *elements, nodes, and mapping between local and global node numbers*
 2. an extended element concept consisting of *cell, vertices, degrees of freedom, local basis functions, geometry mapping, and mapping between local and global degrees of freedom*
- The meaning of the word "element" is multi-fold: the geometry of a finite element (also known as a cell), the geometry and its basis functions, or all information listed under point 2 above.

- One normally computes integrals in the finite element method element by element (cell by cell), either in a local reference coordinate system or directly in the physical domain.
- The advantage of working in the reference coordinate system is that the mathematical expressions for the basis functions depend on the element type only, not the geometry of that element in the physical domain. The disadvantage is that a mapping must be used, and derivatives must be transformed from reference to physical coordinates.
- Element contributions to the global linear system are collected in an element matrix and vector, which must be assembled into the global system using the degree of freedom mapping (`dof_map`) or the node numbering mapping (`elements`), depending on which terminology that is used.
- Dirichlet conditions, involving prescribed values of u at the boundary, are mathematically taken care of via a boundary function that takes on the right Dirichlet values, while the basis functions vanish at such boundaries. The finite element method features a general expression for the boundary function. In software implementations, it is easier to drop the boundary function and the requirement that the basis functions must vanish on Dirichlet boundaries and instead manipulate the global matrix system (or the element matrix and vector) such that the Dirichlet values are imposed on the unknown parameters.
- Neumann conditions, involving prescribed values of the derivative (or flux) of u , are incorporated in boundary terms arising from integrating terms with second-order derivatives by part. Forgetting to account for the boundary terms implies the condition $\partial u / \partial n = 0$ at parts of the boundary where no Dirichlet condition is set.

6.9 Exercises

Exercise 6.1: Compute the deflection of a cable with 2 P1 elements

Solve the problem for u in Exercise 5.2 using two P1 linear elements. Incorporate the condition $u(0) = 0$ by two methods: 1) excluding the unknown at $x = 0$, 2) keeping the unknown at $x = 0$, but modifying the linear system.

Solution. From Exercise 5.2, the Galerkin method, after integration by parts, reads

$$(u', v') = -(1, v) \quad \forall v \in V.$$

We have two elements, $\Omega^{(0)} = [0, \frac{1}{2}]$ and $\Omega^{(1)} = [\frac{1}{2}, 1]$.

Method 1: Excluding the unknown at $x = 0$. Since $u(0) = 0$, we exclude the value at $x = 0$ as degree of freedom in the linear system. (There is no need for any boundary function.) The expansion reads $u = c_0\varphi_1(x) + c_1\varphi_2(x)$. The element matrix has then only one entry in the first element,

$$\tilde{A}^{(0)} = \frac{1}{h}(1).$$

From element 1 we get the usual element matrix

$$\tilde{A}^{(1)} = \frac{1}{h} \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}.$$

The element vector in element 0 becomes

$$\tilde{b}^{(0)} = \frac{h}{2}(-1),$$

while the second element gives a contribution

$$\tilde{b}^{(1)} = \frac{h}{2} \begin{pmatrix} -1 \\ -1 \end{pmatrix}.$$

Assembling the contributions gives

$$\frac{1}{h} \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = -\frac{h}{2} \begin{pmatrix} 2 \\ 1 \end{pmatrix}.$$

Note that $h = \frac{1}{2}$. Solving this system yields

$$c_0 = -\frac{3}{8}, \quad c_1 = -\frac{1}{2} \quad \Rightarrow \quad u = -\frac{3}{8}\varphi_1(x) - \frac{1}{2}\varphi_2(x).$$

Evaluating the exact solution for $x = \frac{1}{2}$ and $x = 1$, we get $3/8$ and $1/2$, respectively, a result which shows that the numerical solution with P1 is exact at the three node points. The difference between the numerical and exact solution is that the numerical solution varies linearly over the two elements while the exact solution is quadratic.

Method 2: Modifying the linear system. Now we let c_i correspond to the value at node x_i , i.e., all known Dirichlet values become part of the

linear system. The expansion is now simply $u = \sum_{i=0}^2 c_i \varphi_i(x)$, with three unknowns c_0 , c_1 , and c_2 . Now the element matrix in the first and second element are equal. The same is true for the element vectors. Assembling yields

$$\frac{1}{h} \begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = -\frac{h}{2} \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}.$$

The next step is to modify the linear system to implement the Dirichlet condition $c_0 = 0$. We first multiply by $h = \frac{1}{2}$ and replace the first equation by $c_0 = 0$:

$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = -\begin{pmatrix} 0 \\ \frac{1}{4} \\ \frac{1}{8} \end{pmatrix}.$$

We see that the remaining 2×2 system is identical to the one previously solved, and the solution is the same.

$$u = 0\varphi_0(x) - \frac{3}{8}\varphi_1(x) - \frac{1}{2}\psi_2(x).$$

Filename: `cable_2P1.`

Exercise 6.2: Compute the deflection of a cable with 1 P2 element

Solve the problem for u in Exercise 5.2 using one P2 element with quadratic basis functions.

Solution. The P2 basis functions on a reference element $[-1, 1]$ are

$$\begin{aligned} \tilde{\varphi}_0(X) &= \frac{1}{2}(X - 1)X \\ \tilde{\varphi}_1(X) &= 1 - X^2 \\ \tilde{\varphi}_2(X) &= \frac{1}{2}(X + 1)X \end{aligned}$$

The element matrix and vector are easily calculated by some lines with `sympy`:

```
import sympy as sym
```

```

X, h = sym.symbols('X h')
half = sym.Rational(1, 2)
psi = [half*(X-1)*X, 1-X**2, half*(X+1)*X]
dpsi_dX = [sym.diff(psi[r], X) for r in range(len(psi))]

# Element matrix
# (2/h)*dpsi_dX[r]*(2/h)*dpsi_dX[s]*h/2
import numpy as np
d = 2
# Use a numpy matrix with general objects to hold A
A = np.empty((d+1, d+1), dtype=object)
for r in range(d+1):
    for s in range(d+1):
        integrand = dpsi_dX[r]*dpsi_dX[s]*2/h
        A[r,s] = sym.integrate(integrand, (X, -1, 1))
print(A)

# Element vector
# f*psi[r]*h/2, f=1
d = 2
b = np.empty(d+1, dtype=object)
for r in range(d+1):
    integrand = -psi[r]*h/2
    b[r] = sym.integrate(integrand, (X, -1, 1))
print(b)

```

The formatted element matrix and vector output becomes

```

[[7/(3*h) -8/(3*h) 1/(3*h)]
 [-8/(3*h) 16/(3*h) -8/(3*h)]
 [1/(3*h) -8/(3*h) 7/(3*h)]]
[-h/6 -2*h/3 -h/6]

```

or in mathematical notation:

$$\tilde{A}^{(e)} = \frac{1}{3h} \begin{pmatrix} 7 & -8 & 1 \\ -8 & 16 & -8 \\ 1 & -8 & 7 \end{pmatrix}, \quad \tilde{b}^{(e)} = -\frac{h}{6} \begin{pmatrix} 1 \\ 4 \\ 1 \end{pmatrix}.$$

Method 1: Excluding the unknown at $x = 0$. The expansion is $u = c_0\varphi_1(x) + c_1\varphi_2(x)$. The element matrix corresponding to the first element excludes contributions associated with the unknown at the left node, i.e., we exclude row and column 0. In the element vector, we exclude the first entry.

$$\tilde{A}^{(0)} = \frac{1}{3h} \begin{pmatrix} 16 & -8 \\ -8 & 7 \end{pmatrix}, \quad \tilde{b}^{(e)} = -\frac{h}{6} \begin{pmatrix} 4 \\ 1 \end{pmatrix}.$$

Now, $h = 1$. The solution of the linear system

$$\frac{1}{3h} \begin{pmatrix} 16 & -8 \\ -8 & 7 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = -\frac{h}{6} \begin{pmatrix} 4 \\ 1 \end{pmatrix}$$

is $c_1 = 3/8$ and $c_2 = 1/2$. As for P1 elements in Exercise 6.1, the values at the nodes are exact, but this time the variation between the nodes is quadratic, i.e., exact. One P2 element produces the complete, exact solution.

Method 2: Modifying the linear system. This time the expansion reads $u = \sum_{i=0}^2 c_i \varphi_i(x)$ with three unknowns c_0 , c_1 , and c_2 . The linear system consists of the complete 3×3 element matrix and the corresponding element vector:

$$\frac{1}{3h} \begin{pmatrix} 7 & -8 & 1 \\ -8 & 16 & -8 \\ 1 & -8 & 7 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = -\frac{h}{6} \begin{pmatrix} 1 \\ 4 \\ 1 \end{pmatrix}.$$

The boundary condition is incorporated by replacing the first equation by $c_0 = 0$, but prior to taking that action, we multiply by $3h$ and insert $h = 1$.

$$\begin{pmatrix} 1 & 0 & 0 \\ -8 & 16 & -8 \\ 1 & -8 & 7 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 0 \\ -2 \\ -\frac{1}{2} \end{pmatrix}.$$

Realizing that $c_0 = 0$, which means we can remove the first column of the system, shows that the equations are the same as above and hence that the solution is identical.

Filename: `cable_1P2`.

Exercise 6.3: Compute the deflection of a cable with a step load

We consider the deflection of a tension cable as described in Exercise 5.2: $w'' = \ell$, $w(0) = w(L) = 0$. Now the load is discontinuous:

$$\ell(x) = \begin{cases} \ell_1, & x < L/2, \\ \ell_2, & x \geq L/2 \end{cases} \quad x \in [0, L].$$

This load is not symmetric with respect to the midpoint $x = L/2$ so the solution loses its symmetry. Scaling the problem by introducing

$$\bar{x} = \frac{x}{L/2}, \quad u = \frac{w}{w_c}, \quad \bar{\ell} = \frac{\ell - \ell_1}{\ell_2 - \ell_1}.$$

This leads to a scaled problem on $[0, 2]$ (we rename \bar{x} as x for convenience):

$$u'' = \bar{\ell}(x) = \begin{cases} 1, & x < 1, \\ 0, & x \geq 1 \end{cases} \quad x \in (0, 1), \quad u(0) = 0, \quad u(2) = 0.$$

a) Find the analytical solution of the problem.

Hint. Integrate the equation separately for $x < 1$ and $x > 1$. Use the conditions that u and u' must be continuous at $x = 1$.

Solution. For $x < 1$ we get $u_1(x) = C_1x + C_2$, and the boundary condition $u_1(0) = 0$ implies $C_2 = 0$. For $x > 1$ we get $u_2(x) = \frac{1}{2}x^2 + C_3x + C_4$. Continuity of $u'(1)$ leads to

$$C_1 = 1 + C_3,$$

and continuity of $u(1)$ means

$$C_1 = \frac{1}{2} + C_3 + C_4,$$

while the condition $u_2(2) = 0$ gives the third equation we need:

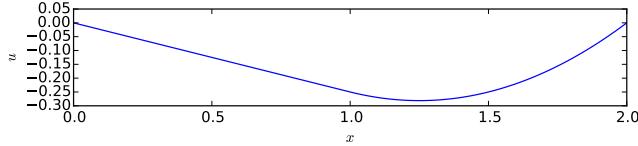
$$2 + 2C_3 + C_4 = 0.$$

We use `sympy` to solve them:

```
>>> from sympy import symbols, Rational, solve
>>> C1, C3, C4 = symbols('C1 C3 C4')
>>> solve([C1 - 1 - C3,
          C1 - Rational(1,2) - C3 - C4,
          2 + 2*C3 + C4], [C1,C3,C4])
{C1: -1/4, C4: 1/2, C3: -5/4}
```

Then

$$u(x) = \begin{cases} -\frac{1}{4}x, & x \leq 1, \\ \frac{1}{2}x^2 - \frac{5}{4}x + \frac{1}{2}, & x \geq 1 \end{cases}$$



b) Use $\psi_i = \sin((i+1)\frac{\pi x}{2})$, $i = 0, \dots, N$ and the Galerkin method to find an approximate solution $u = \sum_j c_j \psi_j$. Plot how fast the coefficients c_j tend to zero (on a log scale).

Solution. The Galerkin formulation of the problem becomes

$$(u', v') = -(\bar{\ell}, v) = \begin{cases} 0, & x \leq 1, \\ -(1, v), & x \geq 1 \end{cases} \quad \forall v \in V.$$

A requirement is that $v(0) = v(2) = 0$ because of the boundary conditions on u . The chosen basis functions fulfill this requirement for any integer i . Inserting $u = \sum_{j=0}^N c_j \psi_j$ and $v = \psi_i$, $i = 0, \dots, N$, gives as usual the linear system $\sum_j A_{i,j} c_j = b_i$, $i = 0, \dots, N$, where

$$A_{i,j} = (i+1)(j+1) \frac{\pi^2}{4} \int_0^2 \cos((i+1)\frac{\pi x}{2}) \cos((j+1)\frac{\pi x}{2}) dx.$$

The cosine functions are orthogonal on $[0, 2]$ so $A_{i,j} = 0$ for $i \neq j$, while $A_{i,i}$ is computed (e.g., by `sympy`) as in Exercise 5.2, part e. The result is

$$A_{i,i} = (i+1)^2 \frac{\pi^2}{4}.$$

The right-hand side is

$$b_i = - \int_1^2 \sin((i+1)\frac{\pi x}{2}) dx = \frac{2}{\pi(i+1)} (\cos((i+1)\pi) - \cos((i+1)\pi/2)).$$

(Trying to do the integral in `sympy` gives a complicated expression that needs discussion - it is easier to do all calculations by hand.) We have that $\cos((i+1)\pi) = -1$ for i even and $\cos((i+1)\pi) = 1$ for i odd, while $\cos((i+1)\pi/2)$ is discussed in Exercise 5.2, part d. The values of $\cos((i+1)\pi) - \cos((i+1)\pi/2)$ can be summarized in the following table:

$i \bmod 4 = 0$	$(i-1) \bmod 4 = 0$	$(i-2) \bmod 4 = 0$	$(i-3) \bmod 4 = 0$
$-1 - 0$	$1 - (-1)$	$-1 - 0$	$1 - 1$

The following function computes the approximate solution:

```
def sine_solution(x, N):
    from numpy import pi, sin
    s = 0
    u = [] # u[i] is the solution for N=i
    for i in range(N+1):
        if i % 4 == 0:
            cos_min_cos = -1
        elif (i-1) % 4 == 0:
            cos_min_cos = 2
        elif (i-2) % 4 == 0:
            cos_min_cos = -1
        elif (i-1) % 4 == 0:
            cos_min_cos = 0

        b_i = 2/(pi*(i+1))*cos_min_cos
        A_ii = (i+1)**2*pi**2/4
        c_i = b_i/A_ii
        s += c_i*sin((i+1)*x*pi/2)
        u.append(s.copy())
    return u
```

The exact solution is a function defined in a piecewise way. Below we make an implementation that works both for array and scalar arguments:

```
def exact_solution(x):
    if isinstance(x, np.ndarray):
        return np.where(x < 1, -1./4*x, 0.5*x**2 - 5./4*x + 0.5)
    else:
        return -1./4*x if x < 1 else 0.5*x**2 - 5./4*x + 0.5
```

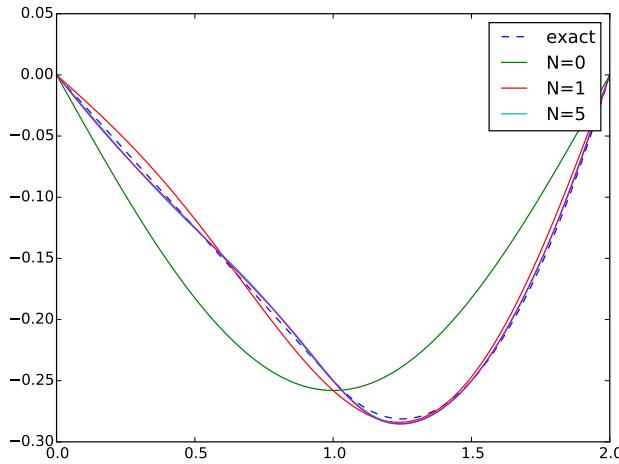
Now we can make a plot of the exact solution and approximate solutions for various N :

```
def plot_sine_solution():
    x = np.linspace(0, 2, 101)
    u = sine_solution(x, N=20)
    plt.figure()
```

```

x = np.linspace(0, 2, 101)
plt.plot(x, exact_solution(x), '---')
N_values = 0, 1, 5
for N in 0, 1, 5, 10:
    plt.plot(x, u[N])
plt.legend(['exact'] + [f'N={N}' % N for N in N_values])
plt.savefig('tmp2.png'); plt.savefig('tmp2.pdf')

```



c) Solve the problem with P1 finite elements. Plot the solution for $N_e = 2, 4, 8$ elements.

Solution. The element matrices and vectors are as for the well-known model problem $u'' = 1$, except that the element vectors vanish for all elements in $[0, 1]$. The following function defines a uniform mesh of P1 elements and runs a finite element algorithm where we use ready-made/known formulas for the element matrix and vector:

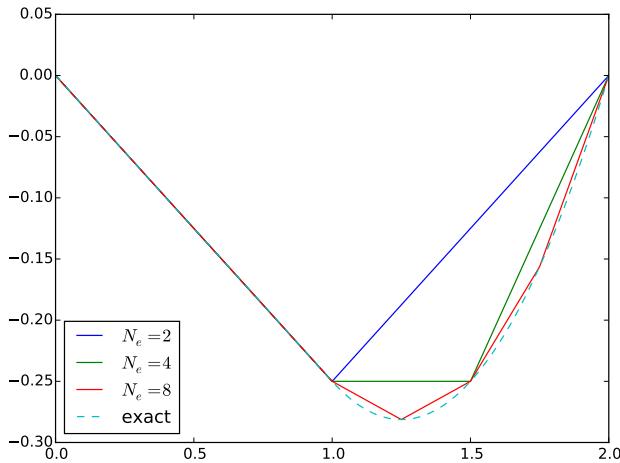
```

def P1_solution():
    plt.figure()
    from fe1D import mesh_uniform, u_glob
    N_e_values = [2, 4, 8]
    d = 1
    legends = []
    for N_e in N_e_values:
        vertices, cells, dof_map = mesh_uniform(
            N_e=N_e, d=d, Omega=[0,2], symbolic=False)
        h = vertices[1] - vertices[0]
        Ae = 1./h*np.array([

```

```
[[1, -1],
 [-1, 1]])
N = N_e + 1
A = np.zeros((N, N))
b = np.zeros(N)
for e in range(N_e):
    if vertices[e] >= 1:
        be = -h/2.*np.array(
            [1, 1])
    else:
        be = h/2.*np.array(
            [0, 0])
    for r in range(d+1):
        for s in range(d+1):
            A[dof_map[e][r], dof_map[e][s]] += Ae[r,s]
            b[dof_map[e][r]] += be[r]
# Enforce boundary conditions
A[0,:] = 0; A[0,0] = 1; b[0] = 0
A[-1,:] = 0; A[-1,-1] = 1; b[-1] = 0
c = np.linalg.solve(A, b)

# Plot solution
print('c:', c)
print('vertices:', vertices)
print('cells:', cells)
print('len(cells):', len(cells))
print('dof_map:', dof_map)
xc, u, nodes = u_glob(c, vertices, cells, dof_map)
plt.plot(xc, u)
legends.append('$N_e=%d$' % N_e)
plt.plot(xc, exact_solution(xc), '--')
legends.append('exact')
plt.legend(loc='lower left')
plt.savefig('tmp3.png'); plt.savefig('tmp3.pdf')
```



Filename: `cable_discont_load.`

Exercise 6.4: Compute with a non-uniform mesh

- a) Derive the linear system for the problem $-u'' = 2$ on $[0, 1]$, with $u(0) = 0$ and $u(1) = 1$, using P1 elements and a *non-uniform* mesh. The vertices have coordinates $x_0 = 0 < x_1 < \dots < x_{N_n-1} = 1$, and the length of cell number e is $h_e = x_{e+1} - x_e$.

Solution. The element matrix and vector for this problem is given by (6.8). The change in this exercise is that h is not a constant element length, but varying with the element number e . We therefore write

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

Assembling such element matrices yields

$$\begin{pmatrix} h_0^{-1} & -h_0^{-1} & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ -h_0^{-1} & h_0^{-1} + h_1^{-1} & -h_1^{-1} & \ddots & & & & & \vdots \\ 0 & -h_1^{-1} & h_1^{-1} + h_2^{-1} & -h_2^{-1} & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -h_{i-1}^{-1} & h_{i-1}^{-1} + h_i^{-1} & -h_i^{-1} & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -h_{N_e}^{-1} \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & -h_{N_e}^{-1} & h_{N_e}^{-1} \end{pmatrix}$$

The element vectors assemble to

$$\begin{pmatrix} h_0 \\ h_0 + h_1 \\ \vdots \\ \vdots \\ h_{i-1} + h_i \\ \vdots \\ \vdots \\ h_{N_e} \end{pmatrix}$$

b) It is of interest to compare the discrete equations for the finite element method in a non-uniform mesh with the corresponding discrete equations arising from a finite difference method. Go through the derivation of the finite difference formula $u''(x_i) \approx [D_x D_x u]_i$ and modify it to find a natural discretization of $u''(x_i)$ on a non-uniform mesh. Compare the finite element and difference discretizations

Solution. Using the definition of the centered, 2nd-order finite difference approximation to u'' we can set up

$$[D_x D_x u]_i = [D_x(D_x u)]_i = \frac{\frac{u_{i+1}-u_i}{x_{i+1}-x_i} - \frac{u_i-u_{i-1}}{x_i-x_{i-1}}}{x_{i+1/2} - x_{i-1/2}}.$$

Now,

$$x_{i+1/2} - x_{i-1/2} = \frac{1}{2}(x_i - x_{i-1}) + \frac{1}{2}(x_{i+1} - x_i) = \frac{1}{2}(x_{i+1} - x_{i-1}).$$

We then get the difference equation

$$u''(x_i) \approx \frac{2}{h_i + h_{i-1}} \left(\frac{u_{i+1} - u_i}{h_i} - \frac{u_i - u_{i-1}}{h_{i-1}} \right) = 2.$$

The factor 2 on either side cancels.

Looking at the finite element equations in a), the equation for a general row i reads

$$\frac{1}{h_{i-1}} c_{i-1} - \left(\frac{1}{h_{i-1}} + \frac{1}{h_i} \right) c_i + \frac{1}{h_i} c_{i+1} = h_{i-1} + h_i.$$

Replacing c_i by u_i (assuming we keep unknowns at all nodes) and rearranging gives

$$-\frac{1}{h_{i-1}}(u_i - u_{i-1}) + \frac{1}{h_i}(u_{i+1} - u_i) = h_{i-1} + h_i.$$

Dividing by the right-hand side gives

$$-\frac{1}{h_{i-1} + h_i} \left(\frac{1}{h_{i-1}}(u_i - u_{i-1}) - \frac{1}{h_i}(u_{i+1} - u_i) \right) = 1.$$

This is the same difference equation as we have in the finite difference method.

Filename: `nonuniform_P1.`

Problem 6.5: Solve a 1D finite element problem by hand

The following scaled 1D problem is a very simple, yet relevant, model for convective transport in fluids:

$$u' = \epsilon u'', \quad u(0) = 0, \quad u(1) = 1, \quad x \in [0, 1]. \quad (6.55)$$

- a)** Find the analytical solution to this problem. (Introduce $w = u'$, solve the first-order differential equation for $w(x)$, and integrate once more.)
- b)** Derive the variational form of this problem.
- c)** Introduce a finite element mesh with uniform partitioning. Use P1 elements and compute the element matrix and vector for a general element.

- d)** Incorporate the boundary conditions and assemble the element contributions.
- e)** Identify the resulting linear system as a finite difference discretization of the differential equation using

$$[D_{2x}u = \epsilon D_x D_x u]_i .$$

- f)** Compute the numerical solution and plot it together with the exact solution for a mesh with 20 elements and $\epsilon = 10, 1, 0.1, 0.01$.
Filename: `convdiff1D_P1`.

Exercise 6.6: Investigate exact finite element solutions

Consider

$$-u''(x) = x^m, \quad x \in (0, L), \quad u'(0) = C, \quad u(L) = D,$$

where $m \geq 0$ is an integer, and L , C , and D are given numbers. Utilize a mesh with two (non-uniform) elements: $\Omega^{(0)} = [0, 3]$ and $\Omega^{(1)} = [3, 4]$. Plot the exact solution and the finite element solution for polynomial degree $d = 1, 2, 3, 4$ and $m = 0, 1, 2, 3, 4$. Find values of d and m that make the finite element solution exact at the nodes in the mesh.

Hint. Use the `mesh_uniform`, `finite_element1D`, and `u_glob2` functions from the `fe1D.py` module.

Solution. The `model2` function from Section 5.1.2 can find the exact solution by `model2(x**m, L, C, D)`. We fix, for simplicity, the values of L , C , and D as $L = 4$, $C = 5$, and $D = 2$. After calculating a symbolic solution, we can convert the expression to a Python function with `sympy.lambdify`. For each d value we then create a uniform mesh and displace the vertex with number 1 to the value 3. The various functions for specifying the element matrix and vector entries are as given in Section 6.4.2, since the model problem is the same. Our code then becomes

```
from u_xx_f_sympy import model2, x
import sympy as sym
import numpy as np
from fe1D import finite_element1D, mesh_uniform, u_glob
import matplotlib.pyplot as plt

C = 5
```

```

D = 2
L = 4

m_values = [0, 1, 2, 3, 4]
d_values = [1, 2, 3, 4]
for m in m_values:
    u = model2(x**m, L, C, D)
    print('\nm=%d, u: %s' % (m, u))
    u_exact = sym.lambdify([x], u)

    for d in d_values:
        vertices, cells, dof_map = mesh_uniform(
            N_e=2, d=d, Omega=[0,L], symbolic=False)
        vertices[1] = 3 # displace vertex
        essbc = {}
        essbc[dof_map[-1][-1]] = D

        c, A, b, timing = finite_element1D(
            vertices, cells, dof_map,
            essbc,
            ilhs=lambda e, phi, r, s, X, x, h:
                phi[1][r](X, h)*phi[1][s](X, h),
            irhs=lambda e, phi, r, X, x, h:
                x**m*phi[0][r](X),
            blhs=lambda e, phi, r, s, X, x, h: 0,
            brhs=lambda e, phi, r, X, x, h:
                -C*phi[0][r](-1) if e == 0 else 0,
            intrule='GaussLegendre')

        # Visualize
        # (Recall that x is a symbol, use xc for coordinates)
        xc, u, nodes = u_glob(c, vertices, cells, dof_map)
        u_e = u_exact(xc)
        print('Max diff at nodes, d=%d: %d, \\
              np.abs(u_exact(nodes) - c).max())
        plt.figure()
        plt.plot(xc, u, 'b-', xc, u_e, 'r--')
        plt.legend(['finite elements, d=%d' % d, 'exact'],
                   loc='lower left')
        figname = 'tmp_%d_%d' % (m, d)
        plt.savefig(figname + '.png'); plt.savefig(figname + '.pdf')

```

First we look at the numerical solution at the nodes:

```

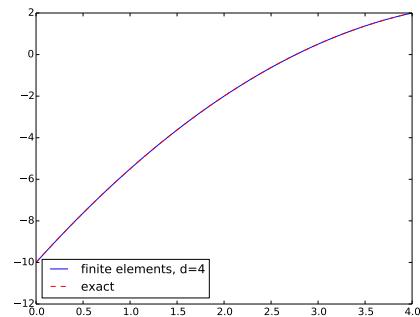
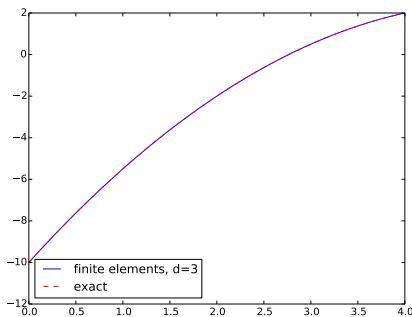
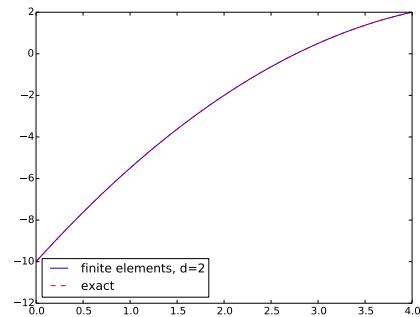
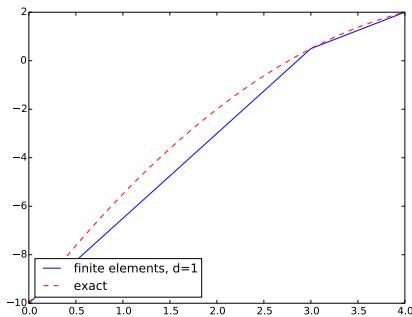
m=0, u: -x**2/2 + 5*x - 10
Max diff at nodes, d=1: 2.22044604925e-16
Max diff at nodes, d=2: 3.5527136788e-15
Max diff at nodes, d=3: 1.7763568394e-15
Max diff at nodes, d=4: 2.46913600677e-13

m=1, u: -x**3/6 + 5*x - 22/3
Max diff at nodes, d=1: 8.881784197e-16
Max diff at nodes, d=2: 1.7763568394e-15
Max diff at nodes, d=3: 7.9936057773e-15

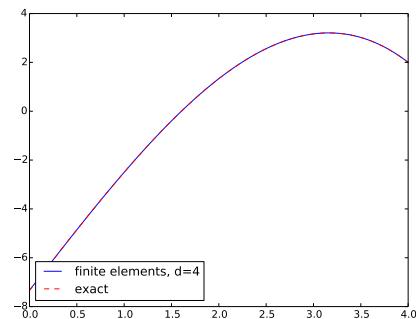
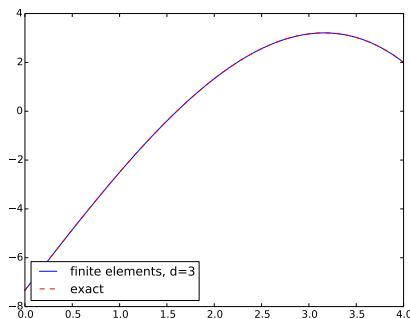
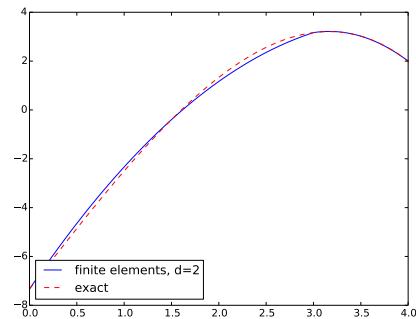
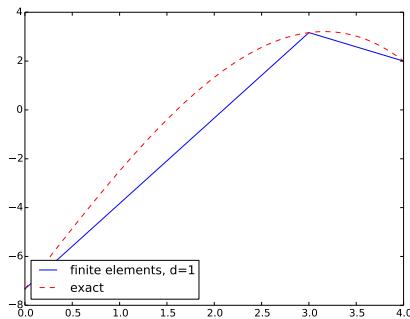
```

```
Max diff at nodes, d=4: 3.01092484278e-13  
  
m=2, u: -x**4/12 + 5*x + 10/3  
Max diff at nodes, d=1: 3.10862446895e-15  
Max diff at nodes, d=2: 0.084375  
Max diff at nodes, d=3: 0.0333333333333  
Max diff at nodes, d=4: 5.20472553944e-13  
  
m=3, u: -x**5/20 + 5*x + 166/5  
Max diff at nodes, d=1: 1.355555555556  
Max diff at nodes, d=2: 0.3796875  
Max diff at nodes, d=3: 0.185714285714  
Max diff at nodes, d=4: 0.0254255022334  
  
m=4, u: -x**6/30 + 5*x + 1778/15  
Max diff at nodes, d=1: 4.8  
Max diff at nodes, d=2: 1.4428125  
Max diff at nodes, d=3: 0.719047619047  
Max diff at nodes, d=4: 0.16865583147
```

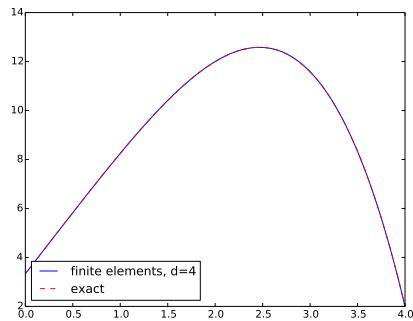
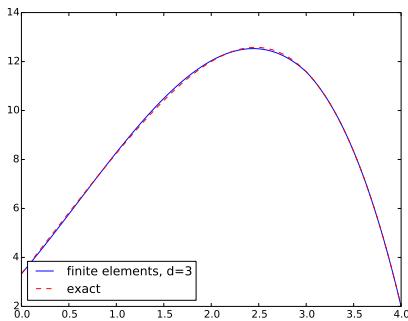
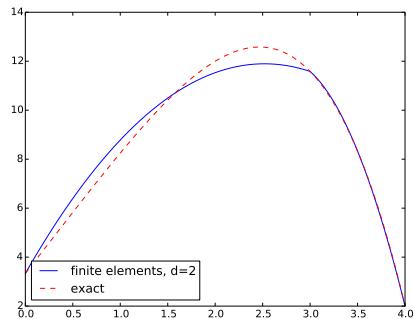
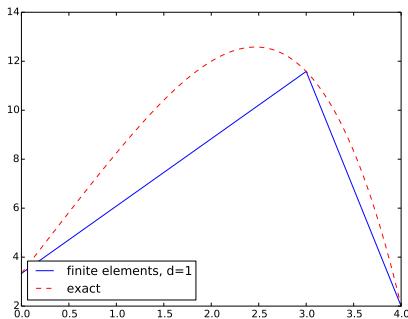
We observe that all elements are capable of computing the exact values at the nodes for $m = 0$ and $m = 1$. With $m = 0$, the solution is quadratic in x , and P2, P3, and P4 will be exact. It is more of a surprise that also the P1 elements are exact in this case. A peculiar feature is that P1 elements are also exact at the nodes $m = 2$, but not P2 and P3 elements (the solution goes like x^4 so it is not surprising that P2 and P3 elements give a numerical error also at the nodes). Clearly, P4 elements produce the exact solution for $m = 4$ since u is a polynomial of degree 4. For larger m values we have discrepancy between the numerical and exact values at the nodes.



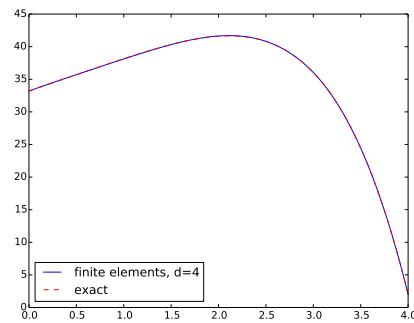
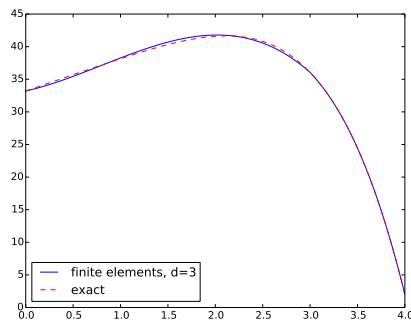
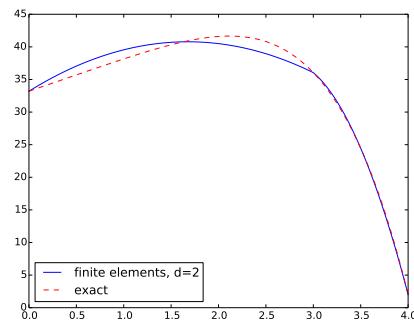
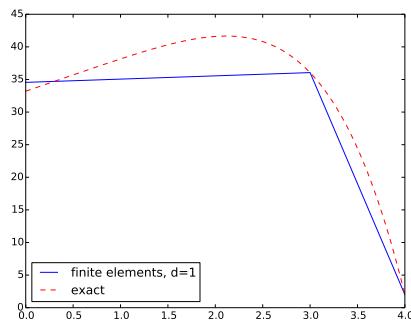
Plots for $m=0$.



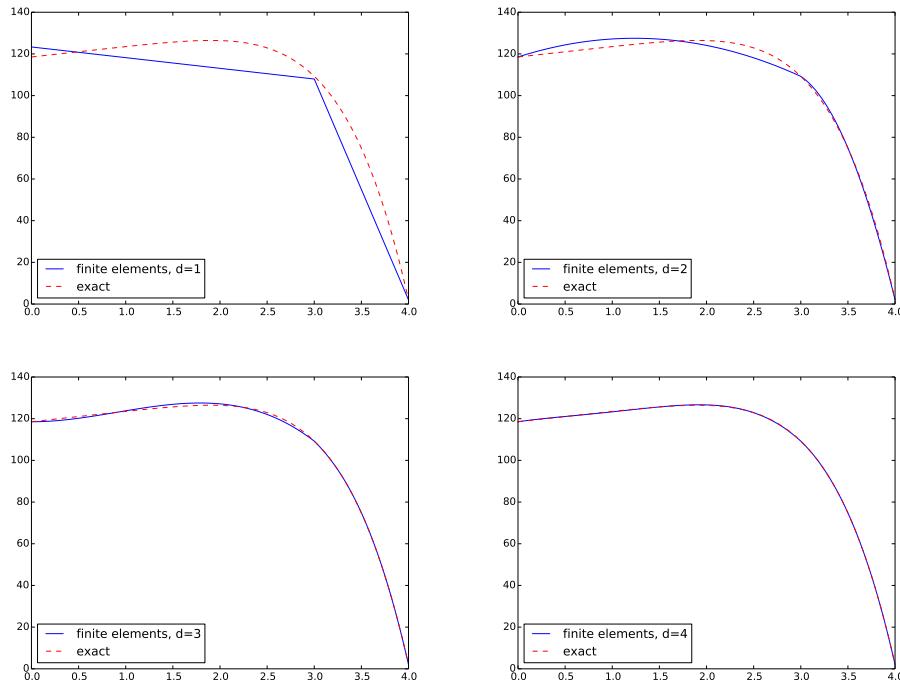
Plots for $m=1$.



Plots for $m=2$.



Plots for $m=3$.



Plots for $m=4$. Filename: u_xx_xm_P1to4.

Exercise 6.7: Compare finite elements and differences for a radially symmetric Poisson equation

We consider the Poisson problem in a disk with radius R with Dirichlet conditions at the boundary. Given that the solution is radially symmetric and hence dependent only on the radial coordinate ($r = \sqrt{x^2 + y^2}$), we can reduce the problem to a 1D Poisson equation

$$-\frac{1}{r} \frac{d}{dr} \left(r \frac{du}{dr} \right) = f(r), \quad r \in (0, R), \quad u'(0) = 0, \quad u(R) = U_R. \quad (6.56)$$

- a) Derive a variational form of (6.56) by integrating over the whole disk, or posed equivalently: use a weighting function $2\pi r v(r)$ and integrate r from 0 to R .
- b) Use a uniform mesh partition with P1 elements and show what the resulting set of equations becomes. Integrate the matrix entries exact by hand, but use a Trapezoidal rule to integrate the f term.

- c) Explain that an intuitive finite difference method applied to (6.56) gives

$$\frac{1}{r_i} \frac{1}{h^2} \left(r_{i+\frac{1}{2}}(u_{i+1} - u_i) - r_{i-\frac{1}{2}}(u_i - u_{i-1}) \right) = f_i, \quad i = rh.$$

For $i = 0$ the factor $1/r_i$ seemingly becomes problematic. One must always have $u'(0) = 0$, because of the radial symmetry, which implies $u_{-1} = u_1$, if we allow introduction of a fictitious value u_{-1} . Using this u_{-1} in the difference equation for $i = 0$ gives

$$\begin{aligned} \frac{1}{r_0} \frac{1}{h^2} \left(r_{\frac{1}{2}}(u_1 - u_0) - r_{-\frac{1}{2}}(u_0 - u_1) \right) &= \\ \frac{1}{r_0} \frac{1}{2h^2} ((r_0 + r_1)(u_1 - u_0) - (r_{-1} + r_0)(u_0 - u_1)) &\approx 2(u_1 - u_0), \end{aligned}$$

if we use $r_{-1} + r_1 \approx 2r_0$.

Set up the complete set of equations for the finite difference method and compare to the finite element method in case a Trapezoidal rule is used to integrate the f term in the latter method.

Filename: `radial_Poisson1D_P1`.

Exercise 6.8: Compute with variable coefficients and P1 elements by hand

Consider the problem

$$-\frac{d}{dx} \left(\alpha(x) \frac{du}{dx} \right) + \gamma u = f(x), \quad x \in \Omega = [0, L], \quad u(0) = \alpha, \quad u'(L) = \beta. \quad (6.57)$$

We choose $\alpha(x) = 1 + x^2$. Then

$$u(x) = \alpha + \beta(1 + L^2) \tan^{-1}(x), \quad (6.58)$$

is an exact solution if $f(x) = \gamma u$.

Derive a variational formulation and compute general expressions for the element matrix and vector in an arbitrary element, using P1 elements and a uniform partitioning of $[0, L]$. The right-hand side integral is challenging and can be computed by a numerical integration rule. The Trapezoidal rule (4.50) gives particularly simple expressions. Filename: `atan1D_P1`.

Exercise 6.9: Solve a 2D Poisson equation using polynomials and sines

The classical problem of applying a torque to the ends of a rod can be modeled by a Poisson equation defined in the cross section Ω :

$$-\nabla^2 u = 2, \quad (x, y) \in \Omega,$$

with $u = 0$ on $\partial\Omega$. Exactly the same problem arises for the deflection of a membrane with shape Ω under a constant load.

For a circular cross section one can readily find an analytical solution. For a rectangular cross section the analytical approach ends up with a sine series. The idea in this exercise is to use a single basis function to obtain an approximate answer.

We assume for simplicity that the cross section is the unit square: $\Omega = [0, 1] \times [0, 1]$.

- a)** We consider the basis $\psi_{p,q}(x, y) = \sin((p+1)\pi x) \sin(q\pi y)$, $p, q = 0, \dots, n$. These basis functions fulfill the Dirichlet condition. Use a Galerkin method and $n = 0$.
- b)** The basis function involving sine functions are orthogonal. Use this property in the Galerkin method to derive the coefficients $c_{p,q}$ in a formula $u = \sum_p \sum_q c_{p,q} \psi_{p,q}(x, y)$.
- c)** Another possible basis is $\psi_i(x, y) = (x(1-x)y(1-y))^{i+1}$, $i = 0, \dots, N$. Use the Galerkin method to compute the solution for $N = 0$. Which choice of a single basis function is best, $u \sim x(1-x)y(1-y)$ or $u \sim \sin(\pi x) \sin(\pi y)$? In order to answer the question, it is necessary to search the web or the literature for an accurate estimate of the maximum u value at $x = y = 1/2$.

Filename: `torsion_sin_xy`.

Exercise 6.10: Solve a 3D Laplace problem with FEniCS

Solve the problem in Section 6.6 in 3D.

Hint. Use BoxMesh as starting point. Consult the FEniCS tutorial [20] if necessary.

Filename: `borehole_fenics3D`.

Exercise 6.11: Solve a 1D Laplace problem with FEniCS

Solve the problem in Section 6.6 in 1D, using the radial formulation $(ru')' = 0$.

Hint. Use `IntervalMesh` for generating the mesh and introduce a stretching if desired. Consult the FEniCS tutorial [20] so you can extract the solution in an array and make your own curve plot of it.

This problem can be solved without markers, see the section on multiple Dirichlet conditions in the tutorial [20] (it is even easier to solve by saying that the solution at the boundary obeys a linear function from u_a to u_b and use this as the only Dirichlet condition).

Filename: `borehole_fenics1D`.

There are at least three different strategies for performing a discretization in time:

1. Use *finite differences* for time derivatives to arrive at a recursive set of spatial problems that can be discretized by the finite element method.
2. Discretize in space by finite elements first, and then solve the resulting system of ordinary differential equations (ODEs) by some *standard library* for ODEs.
3. Discretize in space and time simultaneously by space-time finite elements.

With the first strategy, we discretize in time prior to the space discretization, while the second strategy consists of doing exactly the opposite. It should come as no surprise that in many situations these two strategies end up in exactly the same systems to be solved, but this is not always the case. Also the third approach often reproduces standard finite difference schemes such as the Backward Euler and the Crank-Nicolson schemes for lower-order elements, but offers an interesting framework for deriving higher-order methods. In this chapter we shall be concerned with the first strategy, which is the most common strategy as it turns the time-dependent PDE problem to a sequence of stationary problems for which efficient finite element solution strategies often are available. The second strategy would naturally employ well-known ODE software, which are available as user-friendly routines in Python. However, these routines are presently not efficient enough for PDE problems in 2D and 3D. The

first strategy gives complete hands-on control of the implementation and the computational efficiency in time and space.

We shall use a simple diffusion problem to illustrate the basic principles of how a time-dependent PDE is solved by finite differences in time and finite elements in space. Of course, instead of finite elements, we may employ other types of basis functions, such as global polynomials. Our model problem reads

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u + f(\mathbf{x}, t), \quad \mathbf{x} \in \Omega, \quad t \in (0, T], \quad (7.1)$$

$$u(\mathbf{x}, 0) = I(\mathbf{x}), \quad \mathbf{x} \in \Omega, \quad (7.2)$$

$$\frac{\partial u}{\partial n} = 0, \quad \mathbf{x} \in \partial\Omega, \quad t \in (0, T]. \quad (7.3)$$

Here, $u(\mathbf{x}, t)$ is the unknown function, α is a constant, and $f(\mathbf{x}, t)$ and $I(\mathbf{x})$ are given functions. We have assigned the particular boundary condition (7.3) to minimize the details on handling boundary conditions in the finite element method.

Remark. For systems of PDEs the strategy for discretization in time may have great impact on overall efficiency and accuracy. The Navier-Stokes equations for an incompressible Newtonian fluid is a prime example where many methods have been proposed and where there are notable differences between the different methods. Furthermore, the differences often depend significantly on the application. Discretization in time *before* discretization in space allows for manipulations of the equations and schemes that are very efficient compared to schemes based on discretizing in space first. The schemes are so-called operator-splitting schemes or projection based schemes. These schemes do, however, suffer from loss of accuracy particularly in terms of errors associated with the boundaries. The numerical error is caused by the splitting of the equations which leads to non-trivial splitting of the boundary conditions. It is beyond the scope to discuss these schemes and their differences in this text, but we mentioned that this is the topic of the review article [21] and the more comprehensive books [11, 31].

7.1 Discretization in time by a Forward Euler scheme

The discretization strategy is to first apply a simple finite difference scheme in time and derive a recursive set of spatially continuous PDE

problems, one at each time level. For each spatial PDE problem we can set up a variational formulation and employ the finite element method for solution.

7.1.1 Time discretization

We can apply a finite difference method in time to (7.1). First we need 'a mesh' in time, here taken as uniform with mesh points $t_n = n\Delta t$, $n = 0, 1, \dots, N_t$. A Forward Euler scheme consists of sampling (7.1) at t_n and approximating the time derivative by a forward difference $[D_t^+ u]^n \approx (u^{n+1} - u^n)/\Delta t$. A list of finite difference formulas can be found in A.1. This approximation turns (7.1) into a differential equation that is discrete in time, but still continuous in space. With a finite difference operator notation we can write the time-discrete problem as

$$[D_t^+ u = \alpha \nabla^2 u + f]^n, \quad (7.4)$$

for $n = 1, 2, \dots, N_t - 1$. Writing this equation out in detail and isolating the unknown u^{n+1} on the left-hand side, demonstrates that the time-discrete problem is a recursive set of problems that are continuous in space:

$$u^{n+1} = u^n + \Delta t \left(\alpha \nabla^2 u^n + f(\mathbf{x}, t_n) \right). \quad (7.5)$$

Given $u^0 = I$, we can use (7.5) to compute u^1, u^2, \dots, u^{N_t} .

More precise notation

For absolute clarity in the various stages of the discretizations, we introduce $u_e(\mathbf{x}, t)$ as the exact solution of the space-and time-continuous partial differential equation (7.1) and $u_e^n(\mathbf{x})$ as the time-discrete approximation, arising from the finite difference method in time (7.4). More precisely, u_e fulfills

$$\frac{\partial u_e}{\partial t} = \alpha \nabla^2 u_e + f(\mathbf{x}, t), \quad (7.6)$$

while u_e^{n+1} , with a superscript, is the solution of the time-discrete equations

$$u_e^{n+1} = u_e^n + \Delta t \left(\alpha \nabla^2 u_e^n + f(\mathbf{x}, t_n) \right). \quad (7.7)$$

The u_e^{n+1} quantity is then discretized in space and approximated by u^{n+1} .

7.1.2 Space discretization

We now introduce a finite element approximation to u_e^n and u_e^{n+1} in (7.7), where the coefficients depend on the time level:

$$u_e^n \approx u^n = \sum_{j=0}^N c_j^n \psi_j(\mathbf{x}), \quad (7.8)$$

$$u_e^{n+1} \approx u^{n+1} = \sum_{j=0}^N c_j^{n+1} \psi_j(\mathbf{x}). \quad (7.9)$$

Note that, as before, N denotes the number of degrees of freedom in the spatial domain. The number of time points is denoted by N_t . We define a space V spanned by the basis functions $\{\psi_i\}_{i \in \mathcal{I}_s}$.

7.1.3 Variational forms

A Galerkin method or a weighted residual method with weighting functions w_i can now be formulated. We insert (7.8) and (7.9) in (7.7) to obtain the residual

$$R = u^{n+1} - u^n - \Delta t \left(\alpha \nabla^2 u^n + f(\mathbf{x}, t_n) \right).$$

The weighted residual principle,

$$\int_{\Omega} R w \, dx = 0, \quad \forall w \in W,$$

results in

$$\int_{\Omega} \left[u^{n+1} - u^n - \Delta t \left(\alpha \nabla^2 u^n + f(\mathbf{x}, t_n) \right) \right] w \, dx = 0, \quad \forall w \in W.$$

From now on we use the Galerkin method so $W = V$. Isolating the unknown u^{n+1} on the left-hand side gives

$$\int_{\Omega} u^{n+1} v \, dx = \int_{\Omega} \left[u^n + \Delta t \left(\alpha \nabla^2 u^n + f(\boldsymbol{x}, t_n) \right) \right] v \, dx, \quad \forall v \in V.$$

As usual in spatial finite element problems involving second-order derivatives, we apply integration by parts on the term $\int (\nabla^2 u^n) v \, dx$:

$$\int_{\Omega} \alpha (\nabla^2 u^n) v \, dx = - \int_{\Omega} \alpha \nabla u^n \cdot \nabla v \, dx + \int_{\partial\Omega} \alpha \frac{\partial u^n}{\partial n} v \, dx.$$

The last term vanishes because we have the Neumann condition $\partial u^n / \partial n = 0$ for all n . Our discrete problem in space and time then reads

$$\int_{\Omega} u^{n+1} v \, dx = \int_{\Omega} u^n v \, dx - \Delta t \int_{\Omega} \alpha \nabla u^n \cdot \nabla v \, dx + \Delta t \int_{\Omega} f^n v \, dx, \quad \forall v \in V. \quad (7.10)$$

This is the variational formulation of our recursive set of spatial problems.

Nonzero Dirichlet boundary conditions

As in stationary problems, we can introduce a boundary function $B(\boldsymbol{x}, t)$ to take care of nonzero Dirichlet conditions:

$$u_e^n \approx u^n = B(\boldsymbol{x}, t_n) + \sum_{j=0}^N c_j^n \psi_j(\boldsymbol{x}), \quad (7.11)$$

$$u_e^{n+1} \approx u^{n+1} = B(\boldsymbol{x}, t_{n+1}) + \sum_{j=0}^N c_j^{n+1} \psi_j(\boldsymbol{x}). \quad (7.12)$$

7.1.4 Notation for the solution at recent time levels

In a program it is only necessary to have the two variables u^{n+1} and u^n at the same time at a given time step. It is therefore unnatural to use the index n in computer code. Instead a natural variable naming is `u` for u^{n+1} , the new unknown, and `u_n` for u^n , the solution at the previous time level. When we have several preceding (already computed) time levels,

it is natural to number them like `u_nm1`, `u_nm2`, `u_nm3`, etc., backwards in time, corresponding to u^{n-1} , u^{n-2} , and u^{n-3} . Essentially, this means a one-to-one mapping of notation in mathematics and software, except for u^{n+1} . We shall therefore, to make the distance between mathematics and code as small as possible, often introduce just u for u^{n+1} in the mathematical notation. Equation (7.10) with this new naming convention is consequently expressed as

$$\int_{\Omega} uv \, dx = \int_{\Omega} u^n v \, dx - \Delta t \int_{\Omega} \alpha \nabla u^n \cdot \nabla v \, dx + \Delta t \int_{\Omega} f^n v \, dx. \quad (7.13)$$

This variational form can alternatively be expressed by the inner product notation:

$$(u, v) = (u^n, v) - \Delta t(\alpha \nabla u^n, \nabla v) + \Delta t(f^n, v). \quad (7.14)$$

To simplify the notation for the solution at recent previous time steps and avoid notation like `u_nm1`, `u_nm2`, `u_nm3`, etc., we will let u_1 denote the solution at previous time step, u_2 is the solution two time steps ago, etc.

7.1.5 Deriving the linear systems

In the following, we adopt the previously introduced convention that the unknowns c_j^{n+1} are written as c_j , while the known c_j^n from the previous time level is simply written as c_j^n . To derive the equations for the new unknown coefficients c_j , we insert

$$u = \sum_{j=0}^N c_j \psi_j(\mathbf{x}), \quad u^n = \sum_{j=0}^N c_j^n \psi_j(\mathbf{x})$$

in (7.13) or (7.14), let the equation hold for all $v = \psi_i$, $i = 0, \dots, N$, and order the terms as matrix-vector products:

$$\sum_{j=0}^N (\psi_i, \psi_j) c_j = \sum_{j=0}^N (\psi_i, \psi_j) c_j^n - \Delta t \sum_{j=0}^N (\nabla \psi_i, \alpha \nabla \psi_j) c_j^n + \Delta t (f^n, \psi_i), \quad i = 0, \dots, N. \quad (7.15)$$

This is a linear system $\sum_j A_{i,j} c_j = b_i$ with

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

and

$$b_i = \sum_{j=0}^N (\psi_i, \psi_j) c_j^n - \Delta t \sum_{j=0}^N (\nabla \psi_i, \alpha \nabla \psi_j) c_j^n + \Delta t (f^n, \psi_i).$$

It is instructive and convenient for implementations to write the linear system on the form

$$Mc = Mc_1 - \Delta t K c_1 + \Delta t f, \quad (7.16)$$

where

$$\begin{aligned} M &= \{M_{i,j}\}, \quad M_{i,j} = (\psi_i, \psi_j), \quad i, j \in \mathcal{I}_s, \\ K &= \{K_{i,j}\}, \quad K_{i,j} = (\nabla \psi_i, \alpha \nabla \psi_j), \quad i, j \in \mathcal{I}_s, \\ f &= \{f_i\}, \quad f_i = (f(\mathbf{x}, t_n), \psi_i), \quad i \in \mathcal{I}_s, \\ c &= \{c_i\}, \quad i \in \mathcal{I}_s, \\ c_1 &= \{c_i^n\}, \quad i \in \mathcal{I}_s. \end{aligned}$$

We realize that M is the matrix arising from a term with the zero-th derivative of u , and called the mass matrix, while K is the matrix arising from a Laplace term $\nabla^2 u$. The K matrix is often known as the *stiffness matrix*. (The terms mass and stiffness stem from the early days of finite elements when applications to vibrating structures dominated. The mass matrix arises from the mass times acceleration term in Newton's second law, while the stiffness matrix arises from the elastic forces (the "stiffness") in that law. The mass and stiffness matrix appearing in a diffusion have slightly different mathematical formulas compared to the classic structure problem.)

Remark. The mathematical symbol f has two meanings, either the function $f(\mathbf{x}, t)$ in the PDE or the f vector in the linear system to be solved at each time level.

7.1.6 Computational algorithm

We observe that M and K can be precomputed so that we can avoid computing the matrix entries at every time level. Instead, some matrix-

vector multiplications will produce the linear system to be solved. The computational algorithm has the following steps:

1. Compute M and K .
2. Initialize u^0 by interpolation or projection
3. For $n = 1, 2, \dots, N_t$:
 - a. compute $b = Mc_1 - \Delta t K c_1 + \Delta t f$
 - b. solve $Mc = b$
 - c. set $c_1 = c$

In case of finite element basis functions, interpolation of the initial condition at the nodes means $c_j^n = I(\mathbf{x}_j)$. Otherwise one has to solve the linear system

$$\sum_j \psi_j(\mathbf{x}_i) c_j^n = I(\mathbf{x}_i),$$

where \mathbf{x}_i denotes an interpolation point. Projection (or Galerkin's method) implies solving a linear system with M as coefficient matrix:

$$\sum_j M_{i,j} c_j^n = (I, \psi_i), \quad i \in \mathcal{I}_s.$$

7.1.7 Example using cosinusoidal basis functions

Let us go through a computational example and demonstrate the algorithm from the previous section. We consider a 1D problem

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \quad x \in (0, L), \quad t \in (0, T], \quad (7.17)$$

$$u(x, 0) = A \cos(\pi x/L) + B \cos(10\pi x/L), \quad x \in [0, L], \quad (7.18)$$

$$\frac{\partial u}{\partial x} = 0, \quad x = 0, L, \quad t \in (0, T]. \quad (7.19)$$

We use a Galerkin method with basis functions

$$\psi_i = \cos(i\pi x/L).$$

These basis functions fulfill (7.19), which is not a requirement (there are no Dirichlet conditions in this problem), but helps to make the approximation good.

Since the initial condition (7.18) lies in the space V where we seek the approximation, we know that a Galerkin or least squares approximation of the initial condition becomes exact. Therefore, the initial condition can be expressed as

$$c_1^n = A, \quad c_{10}^n = B,$$

while $c_i^n = 0$ for $i \neq 1, 10$.

The M and K matrices are easy to compute since the basis functions are orthogonal on $[0, L]$. Hence, we only need to compute the diagonal entries. We get

$$M_{i,i} = \int_0^L \cos^2(ix\pi/L) dx,$$

which is computed as

```
>>> import sympy as sym
>>> x, L = sym.symbols('x L')
>>> i = sym.symbols('i', integer=True)
>>> sym.integrate(sym.cos(i*x*sym.pi/L)**2, (x,0,L))
Piecewise((L, Eq(pi*i/L, 0)), (L/2, True))
```

which means L if $i = 0$ and $L/2$ otherwise. Similarly, the diagonal entries of the K matrix are computed as

```
>>> sym.integrate(sym.diff(cos(i*x*sym.pi/L), x)**2, (x,0,L))
pi**2*i**2*Piecewise((0, Eq(pi*i/L, 0)), (L/2, True))/L**2
```

so

$$M_{0,0} = L, \quad M_{i,i} = L/2, \quad i > 0, \quad K_{0,0} = 0, \quad K_{i,i} = \frac{\pi^2 i^2}{2L}, \quad i > 0.$$

The equation system becomes

$$\begin{aligned} Lc_0 &= Lc_0^0 - \Delta t \cdot 0 \cdot c_0^0, \\ \frac{L}{2}c_i &= \frac{L}{2}c_i^n - \Delta t \frac{\pi^2 i^2}{2L} c_i^n, \quad i > 0. \end{aligned}$$

The first equation leads to $c_0 = 0$ for any n since we start with $c_0^0 = 0$ and $K_{0,0} = 0$. The others imply

$$c_i = (1 - \Delta t (\frac{\pi i}{L})^2)^n c_i^n.$$

With the notation c_i^n for c_i at the n -th time level, we can apply the relation above recursively and get

$$c_i^n = (1 - \Delta t (\frac{\pi i}{L})^2)^n c_i^0.$$

Since only two of the coefficients are nonzero at time $t = 0$, we have the closed-form discrete solution

$$u_i^n = A(1 - \Delta t (\frac{\pi}{L})^2)^n \cos(\pi x/L) + B(1 - \Delta t (\frac{10\pi}{L})^2)^n \cos(10\pi x/L).$$

7.1.8 Comparing P1 elements with the finite difference method

We can compute the M and K matrices using P1 elements in 1D. A uniform mesh on $[0, L]$ is introduced for this purpose. Since the boundary conditions are solely of Neumann type in this sample problem, we have no restrictions on the basis functions ψ_i and can simply choose $\psi_i = \varphi_i$, $i = 0, \dots, N = N_n - 1$.

From Section 6.1.2 or 6.1.4 we have that the K matrix is the same as we get from the finite difference method: $h[D_x D_x u]_i^n$, while from Section 4.3.2 we know that M can be interpreted as the finite difference approximation $h[u + \frac{1}{6}h^2 D_x D_x u]_i^n$. The equation system $Mc = b$ in the algorithm is therefore equivalent to the finite difference scheme

$$[D_t^+ (u + \frac{1}{6}h^2 D_x D_x u) = \alpha D_x D_x u + f]_i^n. \quad (7.20)$$

(More precisely, $Mc = b$ divided by h gives the equation above.)

Lumping the mass matrix. As explained in Section 4.3.3, one can turn the M matrix into a diagonal matrix $\text{diag}(h/2, h, \dots, h, h/2)$ by applying the Trapezoidal rule for integration. Then there is no need to solve a linear system at each time level, and the finite element scheme becomes identical to a standard finite difference method

$$[D_t^+ u = \alpha D_x D_x u + f]_i^n. \quad (7.21)$$

The Trapezoidal integration is not as accurate as exact integration and introduces an error. Normally, one thinks of any error as an overall decrease of the accuracy. Nevertheless, errors may cancel each other, and the error introduced by numerical integration may in certain problems lead to improved overall accuracy in the finite element method. The interplay of the errors in the current problem is analyzed in detail in Section 7.4. The effect of the error is at least not more severe than what is produced by the finite difference method and both are of the same order ($\mathcal{O}(h^2)$).

Making M diagonal is usually referred to as *lumping the mass matrix*. There is an alternative method to using an integration rule based on the node points: one can sum the entries in each row, place the sum on the diagonal, and set all other entries in the row equal to zero. For P1 elements both methods of lumping the mass matrix give the same result, but this is in general not true for higher order elements.

7.2 Discretization in time by a Backward Euler scheme

7.2.1 Time discretization

The Backward Euler scheme in time applied to our diffusion problem can be expressed as follows using the finite difference operator notation:

$$[D_t^- u = \alpha \nabla^2 u + f(\mathbf{x}, t)]^n.$$

Here $[D_t^- u]^n \approx (u^n - u^{n-1})/\Delta t$. Written out, and collecting the unknown u^n on the left-hand side and all the known terms on the right-hand side, the time-discrete differential equation becomes

$$u^n - \Delta t \alpha \nabla^2 u^n = u^{n-1} + \Delta t f(\mathbf{x}, t_n). \quad (7.22)$$

From equation (7.22) we can compute u^1, u^2, \dots, u^{N_t} , if we have a start $u^0 = I$ from the initial condition. However, (7.22) is a partial differential equation in space and needs a solution method based on discretization in space. For this purpose we use an expansion as in (7.8)-(7.9).

7.2.2 Variational forms

Inserting (7.8)-(7.9) in (7.22), multiplying by any $v \in V$ (or $\psi_i \in V$), and integrating by parts, as we did in the Forward Euler case, results in the variational form

$$\int_{\Omega} (u^n v + \Delta t \alpha \nabla u^n \cdot \nabla v) \, dx = \int_{\Omega} u^{n-1} v \, dx + \Delta t \int_{\Omega} f^n v \, dx, \quad \forall v \in V. \quad (7.23)$$

Expressed with u for the unknown u^n and u^n for the previous time level, as we have done before, the variational form becomes

$$\int_{\Omega} (uv + \Delta t \alpha \nabla u \cdot \nabla v) \, dx = \int_{\Omega} u^n v \, dx + \Delta t \int_{\Omega} f^n v \, dx, \quad (7.24)$$

or with the more compact inner product notation,

$$(u, v) + \Delta t(\alpha \nabla u, \nabla v) = (u^n, v) + \Delta t(f^n, v). \quad (7.25)$$

7.2.3 Linear systems

Inserting $u = \sum_j c_j \psi_i$ and $u^n = \sum_j c_j^n \psi_i$, and choosing v to be the basis functions $\psi_i \in V$, $i = 0, \dots, N$, together with doing some algebra, lead to the following linear system to be solved at each time level:

$$(M + \Delta t K)c = Mc_1 + \Delta t f, \quad (7.26)$$

where M , K , and f are as in the Forward Euler case and we use the previously introduced notation $c = \{c_i\}$ and $c_1 = \{c_i^n\}$.

This time we really have to solve a linear system at each time level. The computational algorithm goes as follows.

1. Compute M , K , and $A = M + \Delta t K$
2. Initialize u^0 by interpolation or projection
3. For $n = 1, 2, \dots, N_t$:
 - a. compute $b = Mc_1 + \Delta t f$
 - b. solve $Ac = b$
 - c. set $c_1 = c$

In case of finite element basis functions, interpolation of the initial condition at the nodes means $c_j^n = I(\mathbf{x}_j)$. Otherwise one has to solve the

linear system $\sum_j \psi_j(\mathbf{x}_i) c_j = I(\mathbf{x}_i)$, where \mathbf{x}_i denotes an interpolation point. Projection (or Galerkin's method) implies solving a linear system with M as coefficient matrix: $\sum_j M_{i,j} c_j^n = (I, \psi_i)$, $i \in \mathcal{I}_s$.

Finite difference operators corresponding to P1 elements. We know what kind of finite difference operators the M and K matrices correspond to (after dividing by h), so (7.26) can be interpreted as the following finite difference method:

$$[D_t^-(u + \frac{1}{6}h^2 D_x D_x u) = \alpha D_x D_x u + f]_i^n. \quad (7.27)$$

The mass matrix M can be lumped, as explained in Section 7.1.8, and then the linear system arising from the finite element method with P1 elements corresponds to a plain Backward Euler finite difference method for the diffusion equation:

$$[D_t^- u = \alpha D_x D_x u + f]_i^n. \quad (7.28)$$

7.3 Dirichlet boundary conditions

Suppose now that the boundary condition (7.3) is replaced by a mixed Neumann and Dirichlet condition,

$$u(\mathbf{x}, t) = u_0(\mathbf{x}, t), \quad \mathbf{x} \in \partial\Omega_D, \quad (7.29)$$

$$-\alpha \frac{\partial}{\partial n} u(\mathbf{x}, t) = g(\mathbf{x}, t), \quad \mathbf{x} \in \partial\Omega_N. \quad (7.30)$$

Using a Forward Euler discretization in time, the variational form at a time level becomes

$$\int_{\Omega} u^{n+1} v \, dx = \int_{\Omega} (u^n - \Delta t \alpha \nabla u^n \cdot \nabla v) \, dx + \Delta t \int_{\Omega} f v \, dx - \Delta t \int_{\partial\Omega_N} g v \, ds, \quad \forall v \in V. \quad (7.31)$$

7.3.1 Boundary function

The Dirichlet condition $u = u_0$ at $\partial\Omega_D$ can be incorporated through a boundary function $B(\mathbf{x}) = u_0(\mathbf{x})$ and demanding that the basis functions $\psi_j = 0$ at $\partial\Omega_D$. The expansion for u^n is written as

$$u^n(\mathbf{x}) = u_0(\mathbf{x}, t_n) + \sum_{j \in \mathcal{I}_s} c_j^n \psi_j(\mathbf{x}).$$

Inserting this expansion in the variational formulation and letting it hold for all test functions $v \in V$, i.e., all basis functions ψ_i leads to the linear system

$$\begin{aligned} \sum_{j \in \mathcal{I}_s} \left(\int_{\Omega} \psi_i \psi_j \, dx \right) c_j^{n+1} &= \sum_{j \in \mathcal{I}_s} \left(\int_{\Omega} (\psi_i \psi_j - \Delta t \alpha \nabla \psi_i \cdot \nabla \psi_j) \, dx \right) c_j^n - \\ &\quad \int_{\Omega} (u_0(\mathbf{x}, t_{n+1}) - u_0(\mathbf{x}, t_n) + \Delta t \alpha \nabla u_0(\mathbf{x}, t_n) \cdot \nabla \psi_i) \, dx \\ &\quad + \Delta t \int_{\Omega} f \psi_i \, dx - \Delta t \int_{\partial\Omega_N} g \psi_i \, ds, \quad i \in \mathcal{I}_s. \end{aligned}$$

7.3.2 Finite element basis functions

When using finite elements, each basis function φ_i is associated with a node \mathbf{x}_i . We have a collection of nodes $\{\mathbf{x}_i\}_{i \in I_b}$ on the boundary $\partial\Omega_D$. Suppose U_k^n is the known Dirichlet value at \mathbf{x}_k at time t_n ($U_k^n = u_0(\mathbf{x}_k, t_n)$). The appropriate boundary function is then

$$B(\mathbf{x}, t_n) = \sum_{j \in I_b} U_j^n \varphi_j.$$

The unknown coefficients c_j are associated with the rest of the nodes, which have numbers $\nu(i)$, $i \in \mathcal{I}_s = \{0, \dots, N\}$. The basis functions of V are chosen as $\psi_i = \varphi_{\nu(i)}$, $i \in \mathcal{I}_s$, and all of these vanish at the boundary nodes as they should. The expansion for u^{n+1} and u^n become

$$\begin{aligned} u^n &= \sum_{j \in I_b} U_j^n \varphi_j + \sum_{j \in \mathcal{I}_s} c_j^n \varphi_{\nu(j)}, \\ u^{n+1} &= \sum_{j \in I_b} U_j^{n+1} \varphi_j + \sum_{j \in \mathcal{I}_s} c_j \varphi_{\nu(j)}. \end{aligned}$$

The equations for the unknown coefficients $\{c_j\}_{j \in \mathcal{I}_s}$ become

$$\begin{aligned} \sum_{j \in \mathcal{I}_s} \left(\int_{\Omega} \varphi_i \varphi_j \, dx \right) c_j &= \sum_{j \in \mathcal{I}_s} \left(\int_{\Omega} (\varphi_i \varphi_j - \Delta t \alpha \nabla \varphi_i \cdot \nabla \varphi_j) \, dx \right) c_j^n - \\ &\quad \sum_{j \in I_b} \int_{\Omega} \left(\varphi_i \varphi_j (U_j^{n+1} - U_j^n) + \Delta t \alpha \nabla \varphi_i \cdot \nabla \varphi_j U_j^n \right) \, dx \\ &\quad + \Delta t \int_{\Omega} f \varphi_i \, dx - \Delta t \int_{\partial \Omega_N} g \varphi_i \, ds, \quad i \in \mathcal{I}_s. \end{aligned}$$

7.3.3 Modification of the linear system

Instead of introducing a boundary function B we can work with basis functions associated with all the nodes and incorporate the Dirichlet conditions by modifying the linear system. Let \mathcal{I}_s be the index set that counts all the nodes: $\{0, 1, \dots, N = N_n - 1\}$. The expansion for u^n is then $\sum_{j \in \mathcal{I}_s} c_j^n \varphi_j$ and the variational form becomes

$$\begin{aligned} \sum_{j \in \mathcal{I}_s} \left(\int_{\Omega} \varphi_i \varphi_j \, dx \right) c_j &= \sum_{j \in \mathcal{I}_s} \left(\int_{\Omega} (\varphi_i \varphi_j - \Delta t \alpha \nabla \varphi_i \cdot \nabla \varphi_j) \, dx \right) c_{1,j} \\ &\quad + \Delta t \int_{\Omega} f \varphi_i \, dx - \Delta t \int_{\partial \Omega_N} g \varphi_i \, ds. \end{aligned}$$

We introduce the matrices M and K with entries $M_{i,j} = \int_{\Omega} \varphi_i \varphi_j \, dx$ and $K_{i,j} = \int_{\Omega} \alpha \nabla \varphi_i \cdot \nabla \varphi_j \, dx$, respectively. In addition, we define the vectors c , c_1 , and f with entries c_i , $c_{1,i}$, and $\int_{\Omega} f \varphi_i \, dx - \int_{\partial \Omega_N} g \varphi_i \, ds$, respectively.

The equation system can then be written as

$$Mc = Mc_1 - \Delta t K c_1 + \Delta t f. \quad (7.32)$$

When M , K , and f are assembled without paying attention to Dirichlet boundary conditions, we need to replace equation k by $c_k = U_k$ for k corresponding to all boundary nodes ($k \in I_b$). The modification of M consists in setting $M_{k,j} = 0$, $j \in \mathcal{I}_s$, and the $M_{k,k} = 1$. Alternatively, a modification that preserves the symmetry of M can be applied. At each time level one forms $b = Mc_1 - \Delta t K c_1 + \Delta t f$ and sets $b_k = U_k^{n+1}$, $k \in I_b$, and solves the system $Mc = b$.

In case of a Backward Euler method, the system becomes (7.26). We can write the system as $Ac = b$, with $A = M + \Delta t K$ and $b = Mc_1 + f$. Both M and K needs to be modified because of Dirichlet boundary conditions, but the diagonal entries in K should be set to zero and those in M to unity. In this way, for $k \in I_b$ we have $A_{k,k} = 1$. The right-hand side must read $b_k = U_k^n$ for $k \in I_b$ (assuming the unknown is sought at time level t_n).

7.3.4 Example: Oscillating Dirichlet boundary condition

We shall address the one-dimensional initial-boundary value problem

$$u_t = (\alpha u_x)_x + f, \quad x \in \Omega = [0, L], \quad t \in (0, T], \quad (7.33)$$

$$u(x, 0) = 0, \quad x \in \Omega, \quad (7.34)$$

$$u(0, t) = a \sin \omega t, \quad t \in (0, T], \quad (7.35)$$

$$u_x(L, t) = 0, \quad t \in (0, T]. \quad (7.36)$$

A physical interpretation may be that u is the temperature deviation from a constant mean temperature in a body Ω that is subject to an oscillating temperature (e.g., day and night, or seasonal, variations) at $x = 0$.

We use a Backward Euler scheme in time and P1 elements of constant length h in space. Incorporation of the Dirichlet condition at $x = 0$ through modifying the linear system at each time level means that we carry out the computations as explained in Section 7.2 and get a system (7.26). The M and K matrices computed without paying attention to Dirichlet boundary conditions become

$$M = \frac{h}{6} \begin{pmatrix} 2 & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 1 & 4 & 1 & \ddots & & & \vdots \\ 0 & 1 & 4 & 1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & 0 & 1 & 4 & 1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & 1 & 4 & 1 \\ 0 & \cdots & \cdots & \cdots & 0 & 1 & 2 \end{pmatrix} \quad (7.37)$$

and

$$K = \frac{\alpha}{h} \begin{pmatrix} 1 & -1 & 0 & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & -1 & 2 & -1 \\ 0 & \cdots & \cdots & \cdots & 0 & -1 & 1 \end{pmatrix} \quad (7.38)$$

The right-hand side of the variational form contains no source term (f) and no boundary term from the integration by parts ($u_x = 0$ at $x = L$ and we compute as if $u_x = 0$ at $x = 0$ too) and we are therefore left with $M c_1$. However, we must incorporate the Dirichlet boundary condition $c_0 = a \sin \omega t_n$. Let us assume that our numbering of nodes is such that $\mathcal{I}_s = \{0, 1, \dots, N = N_n - 1\}$. The Dirichlet condition can then be incorporated by ensuring that this is the first equation in the linear system. To this end, the first row in K and M is set to zero, but the diagonal entry $M_{0,0}$ is set to 1. The right-hand side is $b = M c_1$, and we set $b_0 = a \sin \omega t_n$. We can write the complete linear system as

$$c_0 = a \sin \omega t_n, \quad (7.39)$$

$$\frac{h}{6}(c_{i-1} + 4c_i + c_{i+1}) + \Delta t \frac{\alpha}{h}(-c_{i-1} + 2c_i - c_{i+1}) = \frac{h}{6}(c_{1,i-1} + 4c_{1,i} + c_{1,i+1}), \quad (7.40)$$

$$i = 1, \dots, N_n - 2,$$

$$\frac{h}{6}(c_{i-1} + 2c_i) + \Delta t \frac{\alpha}{h}(-c_{i-1} + c_i) = \frac{h}{6}(c_{1,i-1} + 2c_{1,i}), \quad (7.41)$$

$$i = N_n - 1.$$

The Dirichlet boundary condition can alternatively be implemented through a boundary function $B(x, t) = a \sin \omega t \varphi_0(x)$:

$$u^n(x) = a \sin \omega t_n \varphi_0(x) + \sum_{j \in \mathcal{I}_s} c_j \varphi_{\nu(j)}(x), \quad \nu(j) = j + 1.$$

Now, $N = N_n - 2$ and the c vector contains values of u at nodes $1, 2, \dots, N_n - 1$. The right-hand side gets a contribution

$$\int_0^L (a(\sin \omega t_n - \sin \omega t_{n-1}) \varphi_0 \varphi_i - \Delta t \alpha a \sin \omega t_n \nabla \varphi_0 \cdot \nabla \varphi_i) \, dx. \quad (7.42)$$

7.4 Accuracy of the finite element solution

7.4.1 Methods of analysis

There are three major tools for analyzing the accuracy of time-dependent finite element problems:

- Truncation error
- Finite element error analysis framework
- Amplification factor analysis

The truncation error is the dominating tool used to analyze finite difference schemes. As we saw in Section 5.1.7 the truncation error analysis is closely related to the method of weighted residuals. A mathematical analysis in terms of the finite difference methods and truncation errors can be found in [32], while the finite elements for parabolic problems using Galerkin is analyzed in [29].

To explain the numerical artifacts from the previous section and highlight the difference between the finite difference and the finite element methods, we turn to the method based on analyzing amplification factors. For wave equations, the counterpart is often referred to as analysis of dispersion relations.

The idea of the method of analyzing amplification factors is to see how sinusoidal waves are amplified in time. For example, if high frequency components are damped much less than the analytical damping, we may see this as ripples or noise in the solution.

Let us address the diffusion equation in 1D, $u_t = \alpha u_{xx}$ for $x \in \Omega = (0, \pi)$ and $t \in (0, T]$. For the case where we have homogeneous Dirichlet conditions and the initial condition is $u(x, 0) = u_0(x)$, the solution to the problem can be expressed as

$$u(x, t) = \sum_{k=1}^{\infty} B_k e^{-\alpha k^2 t} \sin(kx),$$

where $B_k = \int_{\Omega} u_0 \sin(kx)$. This is the well-known Fourier decomposition of a signal in sine waves (one can also use cosine functions or a combination of sines and cosines). For a given wave $\sin(kx)$ with wave length $\lambda = 2\pi/k$, this part of the signal will in time develop as $e^{-\alpha k^2 t}$. Smooth signals will need only a few long waves (B_k decays rapidly with k), while discontinuous or noisy signals will have an amount of short waves with significant amplitude (B_k decays slowly with k).

The amplification factor is defined as $A_e = e^{-\alpha k^2 \Delta t}$ and expresses how much a wave with frequency k is damped over a time step. The corresponding numerical amplification factor will vary with the discretization method and also discretization parameters in space.

From the analytical expression for the amplification factor, we see that $e^{-\alpha k^2}$ is always less than 1. Further, we notice that the amplification factor has a strong dependency on the frequency of the Fourier component. For low frequency components (when k is small), the amplification factor is relatively large although always less than 1. For high frequency components, when k approaches ∞ , the amplification factor goes to 0. Hence, high frequency components (rapid changes such as discontinuities or noise) present in the initial condition will be quickly damped, while low frequency components stay for a longer time interval.

The purpose of this section is to discuss the amplification factor of numerical schemes and compare the amplification factor of the scheme with the known analytical amplification factor.

7.4.2 Fourier components and dispersion relations

Let us again consider the diffusion equation in 1D, $u_t = \alpha u_{xx}$. To allow for general boundary conditions, we include both the $\sin(kx)$ and $\cos(kx)$, or for convenience we expand the Fourier series in terms of $\{e^{ikx}\}_{k=-\infty}^{\infty}$. Hence, we perform a separation in terms of the (Fourier) wave component

$$u = e^{\beta t + ikx}$$

where $\beta = -\alpha k^2$ and $i = \sqrt{-1}$ is the imaginary unit.

Discretizing in time such that $t = n\Delta t$, this exact wave component can alternatively be written as

$$u = A_e^n e^{ikx}, \quad A_e = e^{-\alpha k^2 \Delta t}. \quad (7.43)$$

We remark that A_e is a function of the parameter k , but to avoid to clutter the notation here we write A_e instead of $A_{e,k}$. This convention will be used also for the discrete case.

As we will show, many numerical schemes for the diffusion equation also have a similar wave component as solution:

$$u^n = A^n e^{ikx}, \quad (7.44)$$

where A is an amplification factor to be calculated by inserting (7.44) in the discrete equations. Normally $A \neq A_e$, and the difference in the amplification factor is what introduces (visible) numerical errors. To compute A , we need explicit expressions for the discrete equations for $\{c_j\}_{j \in \mathcal{I}_s}$ in the finite element method. That is, we need to assemble the linear system and look at a general row in the system. This row can be written as a finite difference scheme, and the analysis of the finite element solution is therefore performed in the same way as for finite difference methods. Expressing the discrete finite element equations as finite difference operators turns out to be very convenient for the calculations.

We introduce $x_q = qh$, or $x_q = q\Delta x$, for the node coordinates, to align the notation with that frequently used in finite difference methods. A convenient start of the calculations is to establish some results for various finite difference operators acting on the wave component

$$u_q^n = A^n e^{ikq\Delta x}. \quad (7.45)$$

The forward Euler scheme (see A.1) is

$$\begin{aligned}
u'_q(t_n) &\approx [D_t^+ u_q]^n = \frac{u_q^{n+1} - u_q^n}{\Delta t} \\
&= \frac{A^{n+1} - A^n}{\Delta t} e^{ikq\Delta x} \\
&= A^n e^{ikq\Delta x} \frac{A - 1}{\Delta t}.
\end{aligned}$$

Similarly, the actions of the most common operators of relevance for the model problem at hand are listed below.

$$[D_t^+ A^n e^{ikq\Delta x}]^n = A^n e^{ikq\Delta x} \frac{A - 1}{\Delta t}, \quad (7.46)$$

$$[D_t^- A^n e^{ikq\Delta x}]^n = A^n e^{ikq\Delta x} \frac{1 - A^{-1}}{\Delta t}, \quad (7.47)$$

$$[D_t A^n e^{ikq\Delta x}]^{n+\frac{1}{2}} = A^{n+\frac{1}{2}} e^{ikq\Delta x} \frac{A^{\frac{1}{2}} - A^{-\frac{1}{2}}}{\Delta t} = A^n e^{ikq\Delta x} \frac{A - 1}{\Delta t}, \quad (7.48)$$

$$[D_x D_x A^n e^{ikq\Delta x}]_q = -A^n \frac{4}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right). \quad (7.49)$$

7.4.3 Forward Euler discretization

We insert (7.45) in the Forward Euler scheme with P1 elements in space and $f = 0$ (note that this type of analysis can only be carried out if $f = 0$)

$$[D_t^+ (u + \frac{1}{6} h^2 D_x D_x u)]_q^n = \alpha [D_x D_x u]_q^n. \quad (7.50)$$

We have (using (7.46) and (7.49)):

$$[D_t^+ D_x D_x A e^{ikx}]_q^n = -A^n e^{ikp\Delta x} \frac{A - 1}{\Delta t} \frac{4}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right).$$

The term then reduces to

$$\frac{A - 1}{\Delta t} - \frac{1}{6} \Delta x^2 \frac{A - 1}{\Delta t} \frac{4}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right),$$

or

$$\frac{A - 1}{\Delta t} \left(1 - \frac{2}{3} \sin^2(k\Delta x/2)\right).$$

Introducing $p = k\Delta x/2$ and $F = \alpha\Delta t/\Delta x^2$, the complete scheme becomes

$$(A - 1) \left(1 - \frac{2}{3} \sin^2 p \right) = -4F \sin^2 p,$$

from which we find A to be

$$A = 1 - 4F \frac{\sin^2 p}{1 - \frac{2}{3} \sin^2 p}. \quad (7.51)$$

How does this A change the stability criterion compared to the Forward Euler finite difference scheme and centered differences in space? The stability criterion is $|A| \leq 1$, which here implies $A \leq 1$ and $A \geq -1$. The former is always fulfilled, while the latter leads to

$$4F \frac{\sin^2 p}{1 - \frac{2}{3} \sin^2 p} \leq 2.$$

The factor $\sin^2 p / (1 - \frac{2}{3} \sin^2 p)$ can be plotted for $p \in [0, \pi/2]$, and the maximum value goes to 3 as $p \rightarrow \pi/2$. The worst case for stability therefore occurs for the shortest possible wave, $p = \pi/2$, and the stability criterion becomes

$$F \leq \frac{1}{6} \Rightarrow \Delta t \leq \frac{\Delta x^2}{6\alpha}, \quad (7.52)$$

which is a factor 1/3 worse than for the standard Forward Euler finite difference method for the diffusion equation, which demands $F \leq 1/2$. Lumping the mass matrix will, however, recover the finite difference method and therefore imply $F \leq 1/2$ for stability. In other words, introducing an error in the integration (while keeping the order of accuracy) improves the stability by a factor of 3.

7.4.4 Backward Euler discretization

We can use the same approach of analysis and insert (7.45) in the Backward Euler scheme with P1 elements in space and $f = 0$:

$$[D_t^- (u + \frac{1}{6} h^2 D_x D_x u) - \alpha D_x D_x u]_i^n. \quad (7.53)$$

Similar calculations as in the Forward Euler case lead to

$$(1 - A^{-1}) \left(1 - \frac{2}{3} \sin^2 p \right) = -4F \sin^2 p,$$

and hence

$$A = \left(1 + 4F \frac{\sin^2 p}{1 - \frac{2}{3} \sin^2 p} \right)^{-1}.$$

The quantity in the parentheses is always greater than unity, so $|A| \leq 1$ regardless of the size of F and p . As expected, the Backward Euler scheme is unconditionally stable.

7.4.5 Comparing amplification factors

It is of interest to compare A and A_e as functions of p for some F values. Figure 7.2 displays the amplification factors for the Backward Euler scheme corresponding to a coarse mesh with $F = 2$ and a mesh at the stability limit of the Forward Euler scheme in the finite difference method, $F = 1/2$. Figures 7.1 and 7.2 shows how the accuracy increases with lower F values for both the Forward and Backward Euler schemes, respectively. Figure 7.1 clearly shows that $p = \pi/2$ is the worst case. Accuracy increases with smaller Δt for $F = \frac{1}{6}$ to $F = \frac{1}{12}$ as the distance between the exact and appropriate amplitudes decreases for all p . Backward Euler is stable for all Δt which means that we can employ larger F than the forward Euler scheme. Figure 7.2 shows the amplification factors for $F = 1/2$ and $F = 2$ for a coarse discretization, while Figure 7.3 shows the improvements on a finer mesh. Corresponding figures for the second order Crank-Nicolson method are 7.4 and 7.5. The striking fact, however, is that the accuracy of the finite element method is significantly less than the finite difference method for the same value of F . Lumping the mass matrix to recover the numerical amplification factor A of the finite difference method is therefore a good idea in this problem.

The difference between the exact and the numerical amplification factors gives insight into the order of the approximation. Considering for example the forward Euler scheme, the difference $A_e - A$, where A_e and A are given in (7.43) and (7.51) is a complicated expression. However, performing a Taylor expansion in terms of Δt using `sympy` is straightforward:

```
>>> import sympy as sym
>>> k, dt, dx, alpha = sym.symbols("k dt dx alpha")
>>> p = k*dx/2
>>> F = alpha*dt/(dx*dx)
>>> Ae = sym.exp(-alpha*k**2*dt) # exact
>>> Af = 1 - 4*F*sym.sin(p)**2/(1 - 2.0/3.0*sym.sin(p)**2) # FE
>>> (Ae - Af).series(dt, n=2)
```

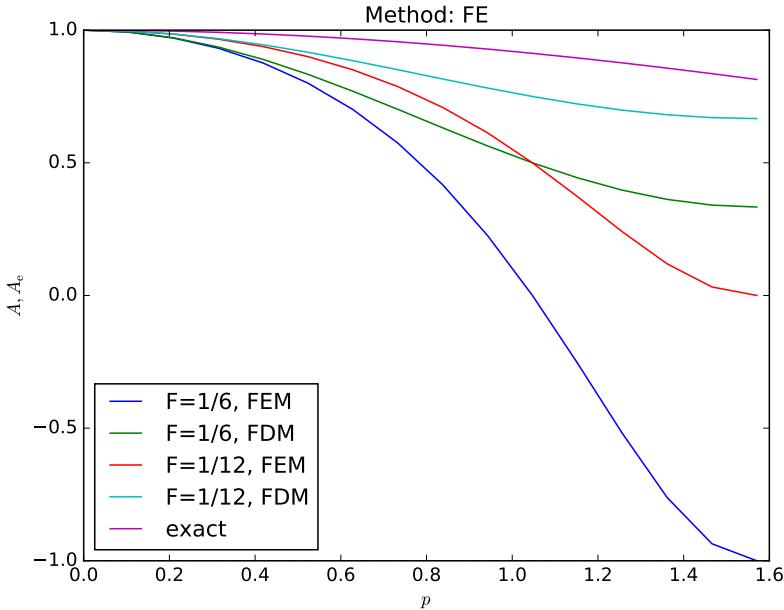


Fig. 7.1 Comparison of fine-mesh amplification factors for Forward Euler discretization of a 1D diffusion equation.

```
dt*(-alpha*k**2 + 4*alpha*sin(dx*k/2)**2/
(-0.6666666666666667*dx**2*sin(dx*k/2)**2 + dx**2)) + 0(dt**2)
```

Hence, the differences between the numerical and exact amplification factor is first order in time, as expected.

The L_2 error of the numerical solution at time step n is

$$\|u - u_e\|_{L_2} = \sqrt{\int_0^1 (u - u_e)^2 dx} = \sqrt{\int_0^1 ((A_e^n - A^n) e^{ikx})^2 dx}$$

Again this yields a complicated expression for hand-calculations, but the following `sympy` commands provides the estimate:

```
>>> n, i, x = sym.symbols("n i x")
>>> e = (Ae**n - Af**n)*sym.exp(i*k*x)
>>> L2_error_squared = sym.integrate(e**2, (x, 0, 1))
>>> sym.sqrt(L2_error_squared.series(dt, n=2))
0(dt)
```

We remark here that it is an advantage to take the square-root after the deriving the Taylor-series.

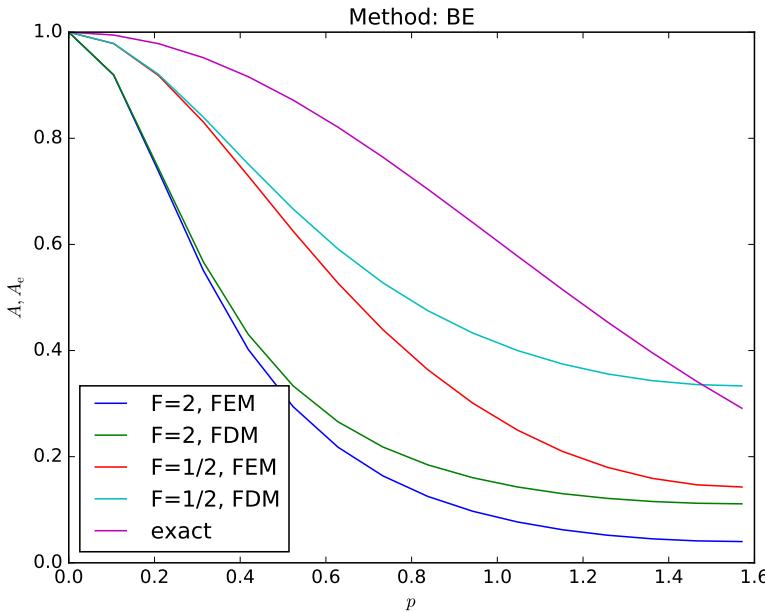


Fig. 7.2 Comparison of coarse-mesh amplification factors for Backward Euler discretization of a 1D diffusion equation.

We may also compute the expansion in terms of k and Δx for both the amplification factor or the $L2$ error of the error in the amplification factor and we find that both are first order in Δt , fourth order in k , and zeroth order in Δx .

```
>>> (Ae-Af).series(k, n=4)
0(k**4)
>>> (Ae-Af).series(dt, n=1)
0(dt)
>>> (Ae-Af).series(dx, n=1)
exp(-alpha*dt*k**2) - 1 + alpha*dt*k**2 + 0(dx)
```

Hence, if the error obtained by our numerical scheme is dominated by the error in the amplification factor, we may expect it to die out quite quickly in terms of k . To improve the error, we must decrease Δt as decreasing the Δx will not improve the error in the amplification factor. In general, for a time-dependent problem with an appropriate scheme we expect an error estimate in the asymptotic regime of the form

$$\|u_e - u\| \leq C(\Delta t)^\alpha + Dh^\beta \quad (7.54)$$

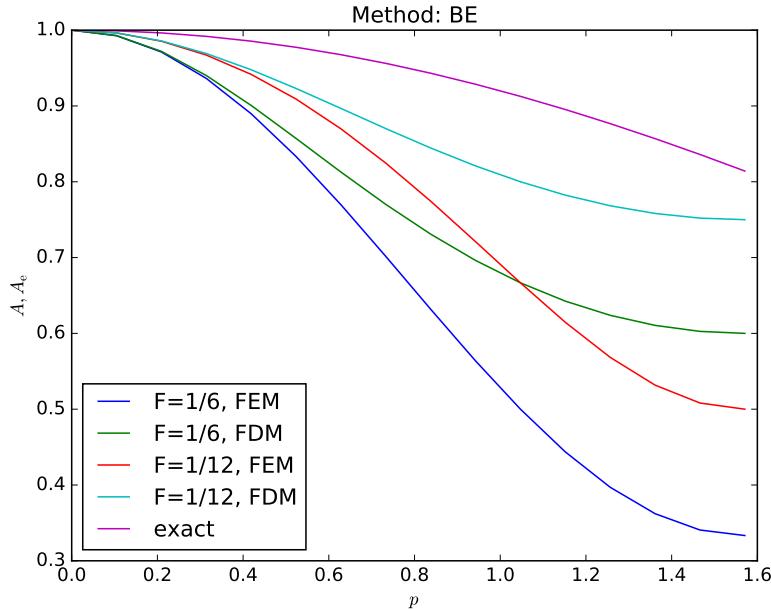


Fig. 7.3 Comparison of fine-mesh amplification factors for Backward Euler discretization of a 1D diffusion equation.

where C, D, α, β depend on the discretization scheme. However, this estimate only holds in the *asymptotic regime*. As the amplification factor analysis shows, we cannot remove the error in the amplification factor by decreasing h . Similarly, we cannot remove spatial error by decreasing Δt . As such, the common way of determining the order of convergence is to first choose a very small h such that the Dh^β term is negligible compared to $C(\Delta t)^\alpha$ and then determine C, α . When C, α are determined then D, β is found in the same way by choosing small Δt .

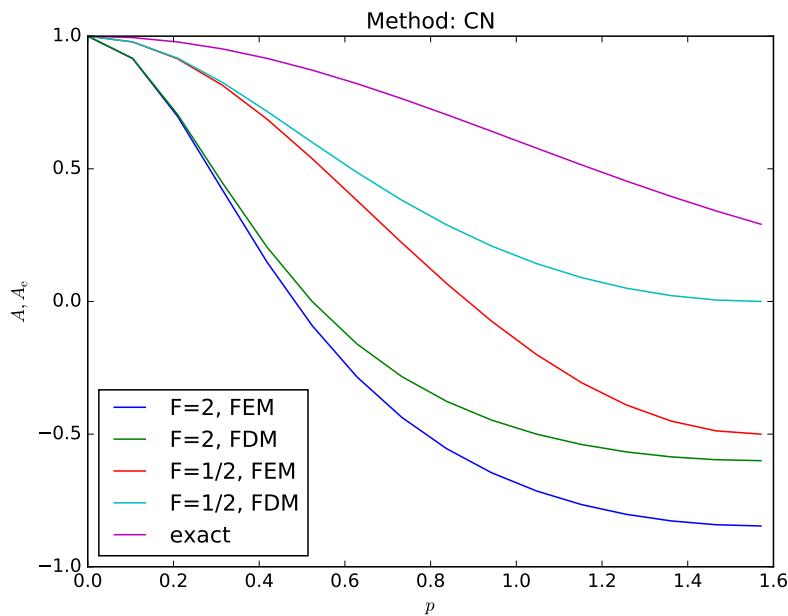


Fig. 7.4 Comparison of coarse-mesh amplification factors for Crank-Nicolson discretization of a 1D diffusion equation.

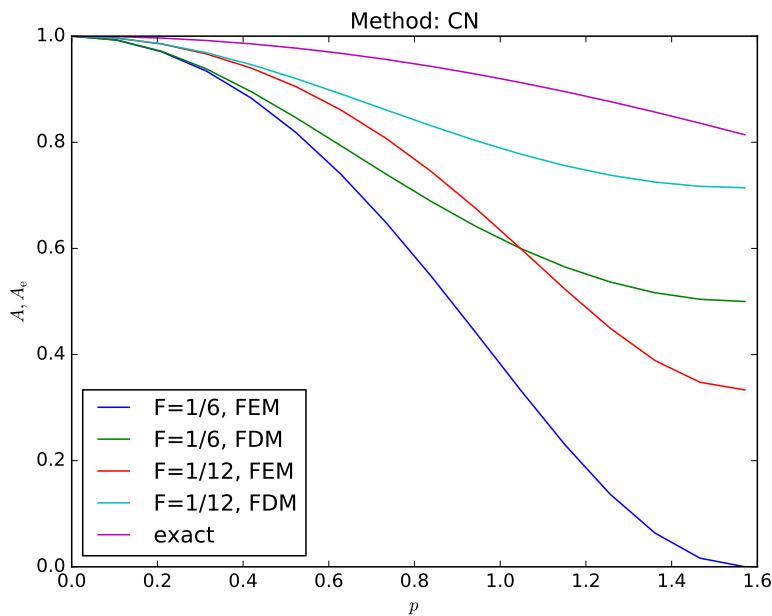


Fig. 7.5 Comparison of fine-mesh amplification factors for Backward Euler discretization of a 1D diffusion equation.

7.5 Exercises

Exercise 7.1: Analyze a Crank-Nicolson scheme for the diffusion equation

Perform the analysis in Section 7.4 for a 1D diffusion equation $u_t = \alpha u_{xx}$ discretized by the Crank-Nicolson scheme in time:

$$\frac{u^{n+1} - u^n}{\Delta t} = \alpha \frac{1}{2} \left(\frac{\partial u^{n+1}}{\partial x^2} + \frac{\partial u^n}{\partial x^2} \right),$$

or written compactly with finite difference operators,

$$[D_t u = \alpha D_x D_x \bar{u}^t]^{n+\frac{1}{2}}.$$

(From a strict mathematical point of view, the u^n and u^{n+1} in these equations should be replaced by u_e^n and u_e^{n+1} to indicate that the unknown is the exact solution of the PDE discretized in time, but not yet in space, see Section 7.1.) Filename: `fe_diffusion`.

Many mathematical models involve $m + 1$ unknown functions governed by a system of $m + 1$ differential equations. In abstract form we may denote the unknowns by $u^{(0)}, \dots, u^{(m)}$ and write the governing equations as

$$\mathcal{L}_0(u^{(0)}, \dots, u^{(m)}) = 0,$$

⋮

$$\mathcal{L}_m(u^{(0)}, \dots, u^{(m)}) = 0,$$

where \mathcal{L}_i is some differential operator defining differential equation number i .

8.1 Variational forms

There are basically two ways of formulating a variational form for a system of differential equations. The first method treats each equation independently as a scalar equation, while the other method views the total system as a vector equation with a vector function as unknown.

8.1.1 Sequence of scalar PDEs formulation

Let us start with the approach that treats one equation at a time. We multiply equation number i by some test function $v^{(i)} \in V^{(i)}$ and integrate over the domain:

$$\int_{\Omega} \mathcal{L}^{(0)}(u^{(0)}, \dots, u^{(m)}) v^{(0)} \, dx = 0, \quad (8.1)$$

$$\vdots \quad (8.2)$$

$$\int_{\Omega} \mathcal{L}^{(m)}(u^{(0)}, \dots, u^{(m)}) v^{(m)} \, dx = 0. \quad (8.3)$$

Terms with second-order derivatives may be integrated by parts, with Neumann conditions inserted in boundary integrals. Let

$$V^{(i)} = \text{span}\{\psi_0^{(i)}, \dots, \psi_{N_i}^{(i)}\},$$

such that

$$u^{(i)} = B^{(i)}(\mathbf{x}) + \sum_{j=0}^{N_i} c_j^{(i)} \psi_j^{(i)}(\mathbf{x}),$$

where $B^{(i)}$ is a boundary function to handle nonzero Dirichlet conditions. Observe that different unknowns may live in different spaces with different basis functions and numbers of degrees of freedom.

From the m equations in the variational forms we can derive m coupled systems of algebraic equations for the $\prod_{i=0}^m N_i$ unknown coefficients $c_j^{(i)}$, $j = 0, \dots, N_i$, $i = 0, \dots, m$.

8.1.2 Vector PDE formulation

The alternative method for deriving a variational form for a system of differential equations introduces a vector of unknown functions

$$\mathbf{u} = (u^{(0)}, \dots, u^{(m)}),$$

a vector of test functions

$$\mathbf{v} = (v^{(0)}, \dots, v^{(m)}),$$

with

$$\mathbf{u}, \mathbf{v} \in \mathbf{V} = V^{(0)} \times \cdots \times V^{(m)}.$$

With nonzero Dirichlet conditions, we have a vector $\mathbf{B} = (B^{(0)}, \dots, B^{(m)})$ with boundary functions and then it is $\mathbf{u} - \mathbf{B}$ that lies in \mathbf{V} , not \mathbf{u} itself.

The governing system of differential equations is written

$$\mathcal{L}(\mathbf{u}) = 0,$$

where

$$\mathcal{L}(\mathbf{u}) = (\mathcal{L}^{(0)}(\mathbf{u}), \dots, \mathcal{L}^{(m)}(\mathbf{u})).$$

The variational form is derived by taking the inner product of the vector of equations and the test function vector:

$$\int_{\Omega} \mathcal{L}(\mathbf{u}) \cdot \mathbf{v} = 0 \quad \forall \mathbf{v} \in \mathbf{V}. \quad (8.4)$$

Observe that (8.4) is one scalar equation. To derive systems of algebraic equations for the unknown coefficients in the expansions of the unknown functions, one chooses m linearly independent \mathbf{v} vectors to generate m independent variational forms from (8.4). The particular choice $\mathbf{v} = (v^{(0)}, 0, \dots, 0)$ recovers (8.1), $\mathbf{v} = (0, \dots, 0, v^{(m)})$ recovers (8.3), and $\mathbf{v} = (0, \dots, 0, v^{(i)}, 0, \dots, 0)$ recovers the variational form number i , $\int_{\Omega} \mathcal{L}^{(i)} v^{(i)} dx = 0$, in (8.1)-(8.3).

8.2 A worked example

We now consider a specific system of two partial differential equations in two space dimensions:

$$\mu \nabla^2 w = -\beta, \quad (8.5)$$

$$\kappa \nabla^2 T = -\mu \|\nabla w\|^2. \quad (8.6)$$

The unknown functions $w(x, y)$ and $T(x, y)$ are defined in a domain Ω , while μ , β , and κ are given constants. The norm in (8.6) is the standard Euclidean norm:

$$\|\nabla w\|^2 = \nabla w \cdot \nabla w = w_x^2 + w_y^2.$$

The boundary conditions associated with (8.5)-(8.6) are $w = 0$ on $\partial\Omega$ and $T = T_0$ on $\partial\Omega$. Each of the equations (8.5) and (8.6) needs one condition at each point on the boundary.

The system (8.5)-(8.6) arises from fluid flow in a straight pipe, with the z axis in the direction of the pipe. The domain Ω is a cross section of the pipe, w is the velocity in the z direction, μ is the viscosity of the fluid, β is the pressure gradient along the pipe, T is the temperature, and κ is the heat conduction coefficient of the fluid. The equation (8.5) comes from the Navier-Stokes equations, and (8.6) follows from the energy equation. The term $-\mu||\nabla w||^2$ models heating of the fluid due to internal friction.

Observe that the system (8.5)-(8.6) has only a one-way coupling: T depends on w , but w does not depend on T . Hence, we can solve (8.5) with respect to w and then (8.6) with respect to T . Some may argue that this is not a real system of PDEs, but just two scalar PDEs. Nevertheless, the one-way coupling is convenient when comparing different variational forms and different implementations.

8.3 Identical function spaces for the unknowns

Let us first apply the same function space V for w and T (or more precisely, $w \in V$ and $T - T_0 \in V$). With

$$V = \text{span}\{\psi_0(x, y), \dots, \psi_N(x, y)\},$$

we write

$$w = \sum_{j=0}^N c_j^{(w)} \psi_j, \quad T = T_0 + \sum_{j=0}^N c_j^{(T)} \psi_j. \quad (8.7)$$

Note that w and T in (8.5)-(8.6) denote the exact solution of the PDEs, while w and T in (8.7) are the discrete functions that approximate the exact solution. It should be clear from the context whether a symbol means the exact or approximate solution, but when we need both at the same time, we use a subscript e to denote the exact solution.

8.3.1 Variational form of each individual PDE

Inserting the expansions (8.7) in the governing PDEs, results in a residual in each equation,

$$R_w = \mu \nabla^2 w + \beta, \quad (8.8)$$

$$R_T = \kappa \nabla^2 T + \mu \|\nabla w\|^2. \quad (8.9)$$

A Galerkin method demands R_w and R_T do be orthogonal to V :

$$\begin{aligned} \int_{\Omega} R_w v \, dx &= 0 \quad \forall v \in V, \\ \int_{\Omega} R_T v \, dx &= 0 \quad \forall v \in V. \end{aligned}$$

Because of the Dirichlet conditions, $v = 0$ on $\partial\Omega$. We integrate the Laplace terms by parts and note that the boundary terms vanish since $v = 0$ on $\partial\Omega$:

$$\int_{\Omega} \mu \nabla w \cdot \nabla v \, dx = \int_{\Omega} \beta v \, dx \quad \forall v \in V, \quad (8.10)$$

$$\int_{\Omega} \kappa \nabla T \cdot \nabla v \, dx = \int_{\Omega} \mu \nabla w \cdot \nabla w v \, dx \quad \forall v \in V. \quad (8.11)$$

The equation R_w in (8.8) is linear in w , while the equation R_T in (8.9) is linear in T and nonlinear in w .

8.3.2 Compound scalar variational form

The alternative way of deriving the variational from is to introduce a test vector function $\mathbf{v} \in \mathbf{V} = V \times V$ and take the inner product of \mathbf{v} and the residuals, integrated over the domain:

$$\int_{\Omega} (R_w, R_T) \cdot \mathbf{v} \, dx = 0 \quad \forall \mathbf{v} \in \mathbf{V}.$$

With $\mathbf{v} = (v_0, v_1)$ we get

$$\int_{\Omega} (R_w v_0 + R_T v_1) \, dx = 0 \quad \forall \mathbf{v} \in \mathbf{V}.$$

Integrating the Laplace terms by parts results in

$$\int_{\Omega} (\mu \nabla w \cdot \nabla v_0 + \kappa \nabla T \cdot \nabla v_1) dx = \int_{\Omega} (\beta v_0 + \mu \nabla w \cdot \nabla w v_1) dx, \quad \forall \mathbf{v} \in \mathbf{V}. \quad (8.12)$$

Choosing $v_0 = v$ and $v_1 = 0$ gives the variational form (8.10), while $v_0 = 0$ and $v_1 = v$ gives (8.11).

With the inner product notation, $(p, q) = \int_{\Omega} pq dx$, we can alternatively write (8.10) and (8.11) as

$$(\mu \nabla w, \nabla v) = (\beta, v) \quad \forall v \in V,$$

$$(\kappa \nabla T, \nabla v) = (\mu \nabla w \cdot \nabla w, v) \quad \forall v \in V,$$

or since μ and κ are considered constant,

$$\mu(\nabla w, \nabla v) = (\beta, v) \quad \forall v \in V, \quad (8.13)$$

$$\kappa(\nabla T, \nabla v) = \mu(\nabla w \cdot \nabla w, v) \quad \forall v \in V. \quad (8.14)$$

Note that the left-hand side of (8.13) is again linear in w , the left-hand side of (8.14) is linear in T and the nonlinearity of w appears in the right-hand side of (8.14)

8.3.3 Decoupled linear systems

The linear systems governing the coefficients $c_j^{(w)}$ and $c_j^{(T)}$, $j = 0, \dots, N$, are derived by inserting the expansions (8.7) in (8.10) and (8.11), and choosing $v = \psi_i$ for $i = 0, \dots, N$. The result becomes

$$\sum_{j=0}^N A_{i,j}^{(w)} c_j^{(w)} = b_i^{(w)}, \quad i = 0, \dots, N, \quad (8.15)$$

$$\sum_{j=0}^N A_{i,j}^{(T)} c_j^{(T)} = b_i^{(T)}, \quad i = 0, \dots, N, \quad (8.16)$$

$$A_{i,j}^{(w)} = \mu(\nabla \psi_j, \nabla \psi_i), \quad (8.17)$$

$$b_i^{(w)} = (\beta, \psi_i), \quad (8.18)$$

$$A_{i,j}^{(T)} = \kappa(\nabla \psi_j, \nabla \psi_i), \quad (8.19)$$

$$b_i^{(T)} = \mu((\sum_j c_j^{(w)} \nabla \psi_j) \cdot (\sum_k c_k^{(w)} \nabla \psi_k), \psi_i). \quad (8.20)$$

It can also be instructive to write the linear systems using matrices and vectors. Define K as the matrix corresponding to the Laplace operator ∇^2 . That is, $K_{i,j} = (\nabla\psi_j, \nabla\psi_i)$. Let us introduce the vectors

$$b^{(w)} = (b_0^{(w)}, \dots, b_N^{(w)}),$$

$$b^{(T)} = (b_0^{(T)}, \dots, b_N^{(T)}),$$

$$c^{(w)} = (c_0^{(w)}, \dots, c_N^{(w)}),$$

$$c^{(T)} = (c_0^{(T)}, \dots, c_N^{(T)}).$$

The system (8.15)-(8.16) can now be expressed in matrix-vector form as

$$\mu K c^{(w)} = b^{(w)}, \quad (8.21)$$

$$\kappa K c^{(T)} = b^{(T)}. \quad (8.22)$$

We can solve the first system for $c^{(w)}$, and then the right-hand side $b^{(T)}$ is known such that we can solve the second system for $c^{(T)}$. Hence, the decoupling of the unknowns w and T reduces the system of nonlinear PDEs to two linear PDEs.

8.3.4 Coupled linear systems

Despite the fact that w can be computed first, without knowing T , we shall now pretend that w and T enter a two-way coupling such that we need to derive the algebraic equations as *one system* for all the unknowns $c_j^{(w)}$ and $c_j^{(T)}$, $j = 0, \dots, N$. This system is nonlinear in $c_j^{(w)}$ because of the $\nabla w \cdot \nabla w$ product. To remove this nonlinearity, imagine that we introduce an iteration method where we replace $\nabla w \cdot \nabla w$ by $\nabla w_- \cdot \nabla w$, w_- being the w computed in the previous iteration. Then the term $\nabla w_- \cdot \nabla w$ is linear in w since w_- is known. The total linear system becomes

$$\sum_{j=0}^N A_{i,j}^{(w,w)} c_j^{(w)} + \sum_{j=0}^N A_{i,j}^{(w,T)} c_j^{(T)} = b_i^{(w)}, \quad i = 0, \dots, N, \quad (8.23)$$

$$\sum_{j=0}^N A_{i,j}^{(T,w)} c_j^{(w)} + \sum_{j=0}^N A_{i,j}^{(T,T)} c_j^{(T)} = b_i^{(T)}, \quad i = 0, \dots, N, \quad (8.24)$$

$$A_{i,j}^{(w,w)} = \mu(\nabla\psi_j, \nabla\psi_i), \quad (8.25)$$

$$A_{i,j}^{(w,T)} = 0, \quad (8.26)$$

$$b_i^{(w)} = (\beta, \psi_i), \quad (8.27)$$

$$A_{i,j}^{(w,T)} = \mu((\nabla w_-) \cdot \nabla\psi_j, \psi_i), \quad (8.28)$$

$$A_{i,j}^{(T,T)} = \kappa(\nabla\psi_j, \nabla\psi_i), \quad (8.29)$$

$$b_i^{(T)} = 0. \quad (8.30)$$

This system can alternatively be written in matrix-vector form as

$$\mu K c^{(w)} = b^{(w)}, \quad (8.31)$$

$$L c^{(w)} + \kappa K c^{(T)} = 0, \quad (8.32)$$

with L as the matrix from the $\nabla w_- \cdot \nabla$ operator: $L_{i,j} = A_{i,j}^{(w,T)}$. The matrix K is $K_{i,j} = A_{i,j}^{(w,w)} = A_{i,j}^{(T,T)}$.

The matrix-vector equations are often conveniently written in block form:

$$\begin{pmatrix} \mu K & 0 \\ L & \kappa K \end{pmatrix} \begin{pmatrix} c^{(w)} \\ c^{(T)} \end{pmatrix} = \begin{pmatrix} b^{(w)} \\ 0 \end{pmatrix},$$

Note that in the general case where all unknowns enter all equations, we have to solve the compound system (8.23)-(8.24) since then we cannot utilize the special property that (8.15) does not involve T and can be solved first.

When the viscosity depends on the temperature, the $\mu\nabla^2 w$ term must be replaced by $\nabla \cdot (\mu(T)\nabla w)$, and then T enters the equation for w . Now we have a two-way coupling since both equations contain w and T and therefore must be solved simultaneously. The equation $\nabla \cdot (\mu(T)\nabla w) = -\beta$ is nonlinear, and if some iteration procedure is invoked, where we use a previously computed T_- in the viscosity ($\mu(T_-)$),

the coefficient is known, and the equation involves only one unknown, w . In that case we are back to the one-way coupled set of PDEs.

We may also formulate our PDE system as a vector equation. To this end, we introduce the vector of unknowns $\mathbf{u} = (u^{(0)}, u^{(1)})$, where $u^{(0)} = w$ and $u^{(1)} = T$. We then have

$$\nabla^2 \mathbf{u} = \begin{pmatrix} -\mu^{-1}\beta \\ -\kappa^{-1}\mu \nabla u^{(0)} \cdot \nabla u^{(0)} \end{pmatrix}.$$

8.4 Different function spaces for the unknowns

It is easy to generalize the previous formulation to the case where $w \in V^{(w)}$ and $T \in V^{(T)}$, where $V^{(w)}$ and $V^{(T)}$ can be different spaces with different numbers of degrees of freedom. For example, we may use quadratic basis functions for w and linear for T . Approximation of the unknowns by different finite element spaces is known as *mixed finite element methods*.

We write

$$\begin{aligned} V^{(w)} &= \text{span}\{\psi_0^{(w)}, \dots, \psi_{N_w}^{(w)}\}, \\ V^{(T)} &= \text{span}\{\psi_0^{(T)}, \dots, \psi_{N_T}^{(T)}\}. \end{aligned}$$

The next step is to multiply (8.5) by a test function $v^{(w)} \in V^{(w)}$ and (8.6) by a $v^{(T)} \in V^{(T)}$, integrate by parts and arrive at

$$\int_{\Omega} \mu \nabla w \cdot \nabla v^{(w)} dx = \int_{\Omega} \beta v^{(w)} dx \quad \forall v^{(w)} \in V^{(w)}, \quad (8.33)$$

$$\int_{\Omega} \kappa \nabla T \cdot \nabla v^{(T)} dx = \int_{\Omega} \mu \nabla w \cdot \nabla w v^{(T)} dx \quad \forall v^{(T)} \in V^{(T)}. \quad (8.34)$$

The compound scalar variational formulation applies a test vector function $\mathbf{v} = (v^{(w)}, v^{(T)})$ and reads

$$\int_{\Omega} (\mu \nabla w \cdot \nabla v^{(w)} + \kappa \nabla T \cdot \nabla v^{(T)}) dx = \int_{\Omega} (\beta v^{(w)} + \mu \nabla w \cdot \nabla w v^{(T)}) dx, \quad (8.35)$$

valid $\forall \mathbf{v} \in \mathbf{V} = V^{(w)} \times V^{(T)}$.

As earlier, we may decouple the system in terms of two linear PDEs as we did with (8.15)-(8.16) or linearize the coupled system by introducing the previous iterate w_- as in (8.23)-(8.24). However, we need to distinguish between $\psi_i^{(w)}$ and $\psi_i^{(T)}$, and the range in the sums over j must match the number of degrees of freedom in the spaces $V^{(w)}$ and $V^{(T)}$. The formulas become

$$\sum_{j=0}^{N_w} A_{i,j}^{(w)} c_j^{(w)} = b_i^{(w)}, \quad i = 0, \dots, N_w, \quad (8.36)$$

$$\sum_{j=0}^{N_T} A_{i,j}^{(T)} c_j^{(T)} = b_i^{(T)}, \quad i = 0, \dots, N_T, \quad (8.37)$$

$$A_{i,j}^{(w)} = \mu(\nabla \psi_j^{(w)}, \nabla \psi_i^{(w)}), \quad (8.38)$$

$$b_i^{(w)} = (\beta, \psi_i^{(w)}), \quad (8.39)$$

$$A_{i,j}^{(T)} = \kappa(\nabla \psi_j^{(T)}, \nabla \psi_i^{(T)}), \quad (8.40)$$

$$b_i^{(T)} = \mu\left(\sum_{j=0}^{N_w} c_j^{(w)} \nabla \psi_j^{(w)}\right) \cdot \left(\sum_{k=0}^{N_w} c_k^{(w)} \nabla \psi_k^{(w)}\right), \quad (8.41)$$

In the case we formulate one compound linear system involving both $c_j^{(w)}$, $j = 0, \dots, N_w$, and $c_j^{(T)}$, $j = 0, \dots, N_T$, (8.23)-(8.24) becomes

$$\sum_{j=0}^{N_w} A_{i,j}^{(w,w)} c_j^{(w)} + \sum_{j=0}^{N_T} A_{i,j}^{(w,T)} c_j^{(T)} = b_i^{(w)}, \quad i = 0, \dots, N_w, \quad (8.42)$$

$$\sum_{j=0}^{N_w} A_{i,j}^{(T,w)} c_j^{(w)} + \sum_{j=0}^{N_T} A_{i,j}^{(T,T)} c_j^{(T)} = b_i^{(T)}, \quad i = 0, \dots, N_T, \quad (8.43)$$

$$A_{i,j}^{(w,w)} = \mu(\nabla \psi_j^{(w)}, \nabla \psi_i^{(w)}), \quad (8.44)$$

$$A_{i,j}^{(w,T)} = 0, \quad (8.45)$$

$$b_i^{(w)} = (\beta, \psi_i^{(w)}), \quad (8.46)$$

$$A_{i,j}^{(w,T)} = \mu(\nabla w_- \cdot \nabla \psi_j^{(w)}), \quad (8.47)$$

$$A_{i,j}^{(T,T)} = \kappa(\nabla \psi_j^{(T)}, \nabla \psi_i^{(T)}), \quad (8.48)$$

$$b_i^{(T)} = 0. \quad (8.49)$$

Here, we have again performed a linearization by employing a previous iterate w_- . The corresponding block form

$$\begin{pmatrix} \mu K^{(w)} & 0 \\ L & \kappa K^{(T)} \end{pmatrix} \begin{pmatrix} c^{(w)} \\ c^{(T)} \end{pmatrix} = \begin{pmatrix} b^{(w)} \\ 0 \end{pmatrix},$$

has square and rectangular block matrices: $K^{(w)}$ is $N_w \times N_w$, $K^{(T)}$ is $N_T \times N_T$, while L is $N_T \times N_w$,

8.5 Computations in 1D

We can reduce the system (8.5)-(8.6) to one space dimension, which corresponds to flow in a channel between two flat plates. Alternatively, one may consider flow in a circular pipe, introduce cylindrical coordinates, and utilize the radial symmetry to reduce the equations to a one-dimensional problem in the radial coordinate. The former model becomes

$$\mu w_{xx} = -\beta, \quad (8.50)$$

$$\kappa T_{xx} = -\mu w_x^2, \quad (8.51)$$

while the model in the radial coordinate r reads

$$\mu \frac{1}{r} \frac{d}{dr} \left(r \frac{dw}{dr} \right) = -\beta, \quad (8.52)$$

$$\kappa \frac{1}{r} \frac{d}{dr} \left(r \frac{dT}{dr} \right) = -\mu \left(\frac{dw}{dr} \right)^2. \quad (8.53)$$

The domain for (8.50)-(8.51) is $\Omega = [0, H]$, with boundary conditions $w(0) = w(H) = 0$ and $T(0) = T(H) = T_0$. For (8.52)-(8.53) the domain is $[0, R]$ (R being the radius of the pipe) and the boundary conditions are $du/dr = dT/dr = 0$ for $r = 0$, $u(R) = 0$, and $T(R) = T_0$.

The exact solutions, w_e and T_e , to (8.50) and (8.51) are computed as

$$\begin{aligned} w_{e,x} &= - \int \frac{\beta}{\mu} dx + C_w, \\ w_e &= \int w_{e,x} dx + D_w, \\ T_{e,x} &= - \int \mu w_{e,x}^2 dx + C_T, \\ T_e &= \int T_{e,x} dx + D_T, \end{aligned}$$

where we determine the constants C_w , D_w , C_T , and D_T by the boundary conditions $w(0) = w(H) = 0$ and $T(0) = T(H) = T_0$. The calculations may be performed in `sympy` as

```
import sympy as sym

x, mu, beta, k, H, C, D, T0 = sym.symbols("x mu beta k H C D T0")
wx = sym.integrate(-beta/mu, (x, 0, x)) + C
w = sym.integrate(wx, x) + D
s = sym.solve([w.subs(x, 0)-0, # x=0 condition
               w.subs(x,H)-0], # x=H condition
               [C, D])           # unknowns
w = w.subs(C, s[C]).subs(D, s[D])
w = sym.simplify(sym.expand(w))

Tx = sym.integrate(-mu*sym.diff(w,x)**2, x) + C
T = sym.integrate(Tx, x) + D
s = sym.solve([T.subs(x, 0)-T0, # x=0 condition
               T.subs(x, H)-T0], # x=H condition
               [C, D])           # unknowns
T = T.subs(C, s[C]).subs(D, s[D])
T = sym.simplify(sym.expand(T))
```

We find that the solutions are

$$\begin{aligned} w_e(x) &= \frac{\beta x}{2\mu} (H - x), \\ T_e(x) &= \frac{\beta^2}{\mu} \left(\frac{H^3 x}{24} - \frac{H^2}{8} x^2 + \frac{H}{6} x^3 - \frac{x^4}{12} \right) + T_0. \end{aligned}$$

The figure 8.1 shows w computed by the finite element method using the decoupled approach with P1 elements, that is; implementing (8.15). The analytical solution w_e is a quadratic polynomial. The linear finite elements result in a poor approximation on the coarse meshes, $N = 2$ and $N = 4$, but the approximation improves fast and already at $N = 8$ the approximation appears adequate. The figure 8.1 shows the approximation

of T and also here we see that the fourth order polynomial is poorly approximated at coarse resolution, but that the approximation quickly improves.

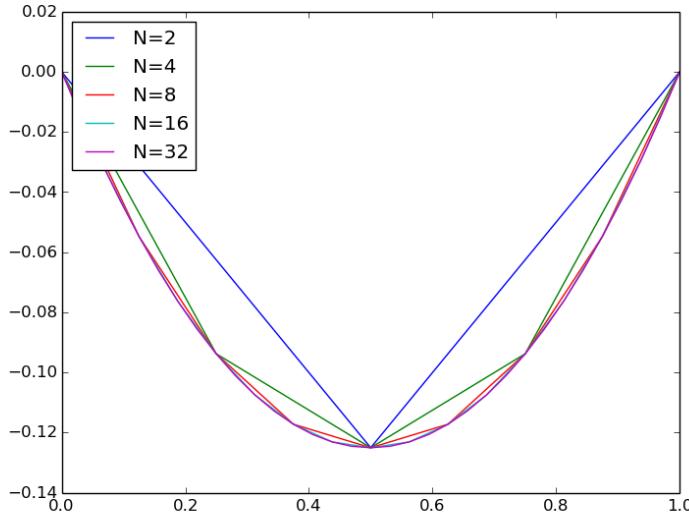


Fig. 8.1 The solution w of (8.50) with $\beta = \mu = 1$ for different mesh resolutions.

The figure 8.2 shows T for different resolutions. The same tendency is apparent although the coarse grid solutions are worse for T than for w . The solutions at $N = 16$ and $N = 32$, however, appear almost identical.

Below we include the code used to solve this problem in FEniCS and plot it using `matplotlib`.

```
def boundary(x):
    return x[0] < DOLFIN_EPS or x[0] > 1.0 - DOLFIN_EPS

from dolfin import *
import matplotlib.pyplot as plt

Ns = [2, 4, 8, 16, 32]
for N in Ns:
    mesh = UnitIntervalMesh(N)
    V = FunctionSpace(mesh, "Lagrange", 1)
    u = TrialFunction(V)
    v = TestFunction(V)

    beta = Constant(1)
    mu = Constant(1)
```

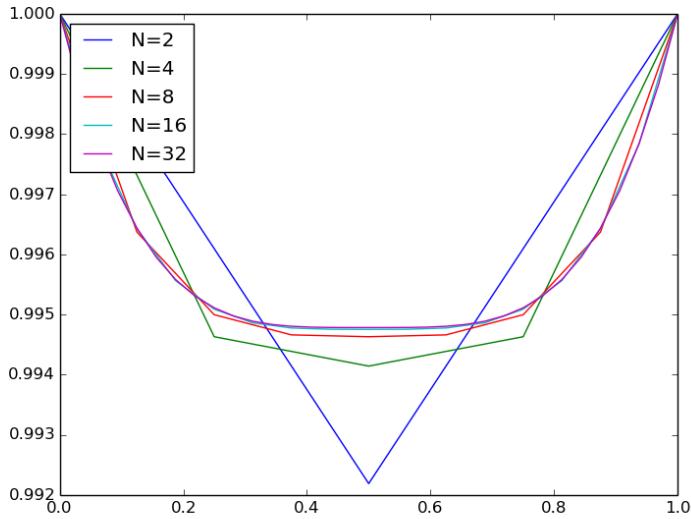


Fig. 8.2 The solution T of (8.51) for $\kappa = H = 1$.

```

bc = DirichletBC(V, Constant(0), boundary)
a = mu*inner(grad(u), grad(v))*dx
L = -beta*v*dx

w = Function(V)
solve(a == L, w, bc)

T0 = Constant(1)
kappa = Constant(1)
bc = DirichletBC(V, T0, boundary)
a = kappa*inner(grad(u), grad(v))*dx
L = -mu*inner(grad(w), grad(w))*v*dx

T = Function(V)
solve(a == L, T, bc)

x = V.tabulate_dof_coordinates()
plt.plot(x, T.vector().get_local())
plt.legend(["N=%d"%N for N in Ns], loc="upper left")
plt.show()

```

Most of the FEniCS code should be familiar to the reader, but we remark that we use the function `V.tabulate_dof_coordinates()` to obtain the coordinates of the nodal points. This is a general function

that works for any finite element implemented in FEniCS and also in a parallel setting.

The calculations for (8.52) and (8.53) are similar. The `sympy` code

```
import sympy as sym

r, R, mu, beta, C, D, T0 = sym.symbols("r R mu beta C D T0")
rwr = sym.integrate(-(beta/mu)*r, r) + C
w = sym.integrate(rwr/r, r) + D
s = sym.solve([sym.diff(w,r).subs(r, 0)-0, # r=0 condition
               w.subs(r,R)-0], # r=R condition
               [C, D]) # unknowns
w = w.subs(C, s[C]).subs(D, s[D])
w = sym.simplify(sym.expand(w))

rTr = sym.integrate(-mu*sym.diff(w,r)**2*r, r) + C
T = sym.integrate(rTr/r, r) + D
s = sym.solve([sym.diff(T,r).subs(r, 0)-T0, # r=0 condition
               T.subs(r, R)-T0], # r=R condition
               [C, D]) # unknowns
T = T.subs(C, s[C]).subs(D, s[D])
T = sym.simplify(sym.expand(T))
```

and we obtain the solutions

$$w(r) = \frac{\beta(R^2 - r^2)}{4\mu},$$

$$T(r) = \frac{1}{64\mu} \left(R^4 \beta^2 + 64T_0\mu - \beta^2 r^4 \right).$$

The radial solution corresponds to the analytical solution in 3D and is very useful for the purpose of verifying the code of multi-physics flow problems.

8.5.1 Another example in 1D

Consider the problem

$$-(au')' = 0, \quad (8.54)$$

$$u(0) = 0, \quad (8.55)$$

$$u(1) = 1. \quad (8.56)$$

For any scalar a (larger than 0), we may easily verify that the solution is $u(x) = x$. In many applications, such as for example porous media flow or heat conduction, the parameter a contains a jump that represents the transition from one material to another. Hence, let us consider the

problem where a is on the following form

$$a(x) = \begin{cases} 1 & \text{if } x \leq \frac{1}{2}, \\ a_0 & \text{if } x > \frac{1}{2}. \end{cases}$$

Notice that for such an $a(x)$, the equation (8.54) does not necessarily make sense because we cannot differentiate $a(x)$. Strictly speaking $a'(x)$ would be a Dirac's delta function in $x = \frac{1}{2}$, that is; $a'(x)$ is ∞ at $x = \frac{1}{2}$ and zero everywhere else.

Hand-calculations do however show that we may be able to compute the solution. Integrating (8.54) yields the expression

$$-(au') = C$$

A trick now is to divide by $a(x)$ on both sides to obtain

$$-u' = \frac{C}{a}$$

and since a is a piecewise constant

$$u(x) = \frac{C}{a(x)}x + D$$

The boundary conditions demand that $u(0) = 0$, which means that $D = 0$. In order to obtain an expression for $u(1) = 1$ we note that u is a piecewise linear function with $u' = C$ for $x \in (0, 0.5)$ and $u' = \frac{C}{a_0}$ for $x \in (0.5, 1)$. We may therefore express $u(1)$ in terms of $u(0)$ plus the derivatives at the midpoints of the two intervals, i.e., $u(1) = u(0) + \frac{1}{2}u'(0.25) + \frac{1}{2}u'(0.75) = 0.5(C + \frac{C}{a_0}) = 1$. In other words, $C = \frac{2a_0}{a_0+1}$ and the analytical solution becomes

$$u_e(x) = \begin{cases} \frac{2a_0}{a_0+1}x & \text{for } 0 < x \leq 0.5 \\ \frac{a_0-1}{a_0+1}x + \frac{a_0}{a_0+1} & \text{for } 0.5 < x \leq 1 \end{cases} \quad (8.57)$$

The variational problem derived from a standard Galerkin method reads: Find u such that

$$\int_{\Omega} au'v' \, dx = \int_{\Omega} fv \, dx + \int_{\partial\Omega} au'v \, ds$$

We observe that in this variational formulation, the discontinuity of a does not cause any problem as the differentiation is moved from a (and u') to v by using integration by parts (or Green's lemma). As earlier, to include the boundary conditions, we may use a boundary function such

that

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)$$

and further letting $v = \psi_i(x)$, the corresponding linear system is $\sum_j A_{i,j} c_j = b_i$ with

$$\begin{aligned} A_{i,j} &= (a\psi'_j, \psi'_i) = \int_{\Omega} a(x)\psi'_j(x)\psi'_i(x) \, dx, \\ b_i &= - \int_{\Omega} aB'v' \, dx + \int_{\partial\Omega} a \frac{\partial B'}{\partial n} v \, ds. \end{aligned}$$

In FEniCS, the linear algebra approach is used and the boundary conditions are inserted in the element matrix as described in Section 6.2.5. The solution of the problem is shown in Figure 8.3 at different mesh resolutions. The analytical solution in (8.57) is a piecewise polynomial, linear for x in $[0, \frac{1}{2})$ and $(\frac{1}{2}, 1]$ and it seems that the numerical strategy gives a good approximation of the solution. The FEniCS program generating the plot is

```
from fenics import *

class uExact(UserExpression):
    def __init__(self, **kwargs):
        super().__init__(degree=kwargs["degree"])
        self.a0 = kwargs["a0"]
        self.a = 1
    def eval(self, value, x):
        if x[0] < 0.5:
            value[0] = (2.0*self.a0 / (self.a0 +1)) / self.a * x[0]
        else:
            value[0] = ((2.0*self.a0 / (self.a0 +1)) / self.a0) * x[0] \
                + (self.a0-1)/(self.a0+1)

    class A(UserExpression):
        def __init__(self, **kwargs):
            super().__init__(degree=kwargs["degree"])
            self.a0 = kwargs["a0"]
        def eval(self, value, x):
            value[0] = 1
            if x[0] >= 0.5: value[0] = self.a0

    class DirichletBoundary(SubDomain):
        def inside(self, x, on_boundary):
            return on_boundary

p_bc = f = Expression("x[0]", degree=2)
Ns = [2, 8, 32]
```

```

a0 = 0.1
for N in Ns:
    mesh = UnitIntervalMesh(N)
    V = FunctionSpace(mesh, "CG", 1)
    Q = FunctionSpace(mesh, "DG", 0)
    u = TrialFunction(V)
    v = TestFunction(V)
    a_coeff = A(degree=2, a0=a0)
    a = a_coeff*inner(grad(u), grad(v))*dx
    f = Constant(0)
    L = f*v*dx
    bc = DirichletBC(V, p_bc, DirichletBoundary())
    u = Function(V)
    solve(a == L, u, bc)

    # plot solution on the various meshes
    plt.plot(V.tabulate_dof_coordinates(), u.vector().get_local())

# create plot for analytical solution, plot, save
u_exact = project(uExact(a0=a0, degree=1), V)
plt.plot(V.tabulate_dof_coordinates(), u_exact.vector().get_local())
legend = ["N=%d"%N for N in Ns]
legend.append("analytical solution")
plt.legend(legend, loc="upper left")
plt.savefig('darcy_a1D.png'); plt.savefig('darcy_a1D.pdf')

```

Figure 8.3 shows the solution u . Clearly we have a good approximation already on a mesh with just two elements as the solution is piecewise linear as found in (8.57).

The flux au' is often a quantity of interest. Because the flux involves differentiation with respect to x we do not have direct access to it and have to compute it. A natural approach is to take the Galerkin approximation, that is we seek a $w \approx au'$ on the form $w = \sum_{j \in \mathcal{I}_s} d_j \psi_j$ and require Galerkin orthogonality. In other words, we require that $w - au'$ is orthogonal to $\{\psi_i\}$. This is done by solving the linear system $\sum_j M_{i,j} d_j = b_i$ with

$$M_{i,j} = (a\psi_j, \psi_i) = \int_{\Omega} a(x)\psi_j(x)\psi_i(x) dx,$$

$$b_i = (au', \psi_i) = \int_{\Omega} a(x) \sum_j c_j \psi'_j(x) dx.$$

Computing the flux by taking the Galerkin projection as described in Section 4.7.1 is implemented in the `project` method in FEniCS and is obtained as

```
aux = project(-a_coeff*u.dx(0), V)
```

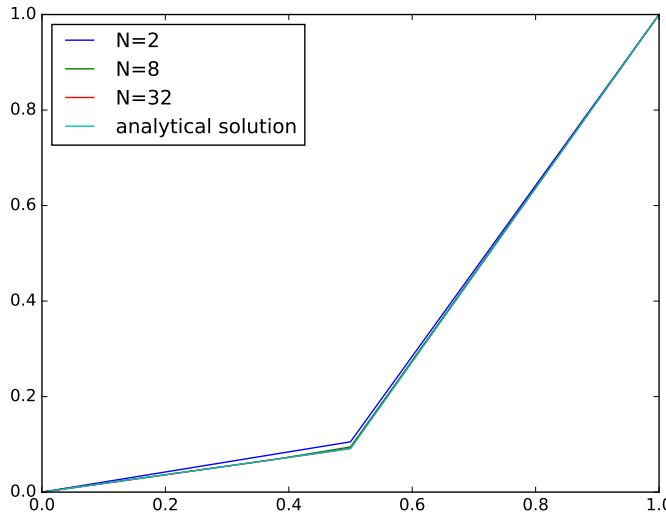


Fig. 8.3 Solution of the Darcy problem with discontinuous coefficient for different number of elements N .

As shown in Figure 8.4 this approach does not produce a good approximation of the flux. Moreover, the approximate solution does not seem to improve close to the jump as the mesh is refined. The problem is that we try to approximate a discontinuous function with a continuous basis and this may often cause spurious oscillations. In this case, we may fix the problem by alternative post-processing methods for calculating the flux. For instance, we may project onto piecewise constant functions instead. However, in general, we would still lose accuracy as the first order derivative involved in the flux calculation lower the order of accuracy by one.

An alternative method that makes the flux approximation w more accurate than the underlying u is an equivalent form of (8.54) where the flux is one of the unknowns. This formulation is usually called the mixed formulation. The equations reads:

$$\frac{\partial w}{\partial x} = 0, \quad (8.58)$$

$$w = -a \frac{\partial u}{\partial x}. \quad (8.59)$$

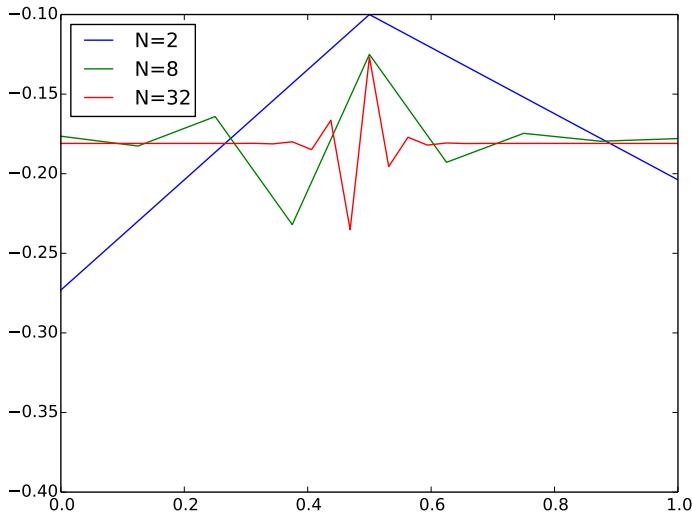


Fig. 8.4 The corresponding flux au' for the Darcy problem with discontinuous coefficient for different number of elements N .

Equation (8.59) is Darcy's law for a porous media. A straightforward calculation shows that inserting (8.59) into (8.58) yields the equation (8.54). We also note that we have replaced the second order differential equation with a system of two first order differential equations.

It is common to swap the order of the equations and also divide equation (8.59) by a . Then variational formulation of the problem, having the two unknowns w and u and corresponding test functions $v^{(w)}$ and $v^{(u)}$, becomes

$$\int_{\Omega} \frac{1}{a} w v^{(w)} + \frac{\partial u}{\partial x} v^{(w)} \, dx = 0, \quad (8.60)$$

$$\int_{\Omega} \frac{\partial w}{\partial x} v^{(u)} \, dx = 0. \quad (8.61)$$

To obtain a suitable variation formulation we perform integration by parts on the last term of (8.60) to obtain

$$\int_{\Omega} \frac{\partial u}{\partial x} v^{(w)} \, dx = - \int_{\Omega} u \frac{\partial v^{(w)}}{\partial x} \, dx + \int_{\partial\Omega} u v^{(w)} \, dx$$

Notice that the Dirichlet conditions of (8.54) becomes a Neumann condition in this mixed formulation. Vice versa, a Neumann condition in (8.54) becomes a Dirichlet condition in the mixed case.

To obtain a linear system of equation, let $u = \sum_{j \in \mathcal{I}_s} c_j \psi_j^{(u)}$, $w = \sum_{j \in \mathcal{I}_s} c_j \psi_j^{(w)}$, $v^{(u)} = \psi_i^{(u)}$, and $v^{(w)} = \psi_i^{(w)}$. We obtain the following system of linear equations

$$A c = \begin{bmatrix} A^{(w,w)} & A^{(w,u)} \\ A^{(u,w)} & 0 \end{bmatrix} \begin{bmatrix} c^{(w)} \\ c^{(u)} \end{bmatrix} = \begin{bmatrix} b^{(w)} \\ b^{(u)} \end{bmatrix} = b,$$

where

$$\begin{aligned} A_{i,j}^{(w,w)} &= \int_{\Omega} \frac{1}{a(x)} \psi_j^{(w)}(x) \psi_i^{(w)}(x) dx & i, j = 0 \dots N^{(w)} - 1, \\ A_{i,j}^{(w,u)} &= - \int_{\Omega} \frac{\partial}{\partial x} \psi_j^{(w)}(x) \psi_i^{(u)}(x) dx & i = 0 \dots N^{(w)} - 1, j = 0, \dots, N^{(u)} - 1, \\ A_{i,j}^{(u,w)} &= -A_{j,i}^{(w,u)}, \\ b_i^{(w)} &= \int_{\Omega} aB \frac{\partial}{\partial x} \psi_i^{(w)} - [aB \psi_i^{(w)}]_0^1, \\ b_i^{(u)} &= (0, \psi_i^{(u)}) = 0. \end{aligned}$$

In Figure 8.5 the solution u obtained by solving the of system (8.59) using piecewise linear elements for w and piecewise constants for u . Clearly, u converges towards the analytical solution as the mesh is refined, although in a staggered way. In Figure 8.6 we display the flux au' . The flux appears to be a constant as predicted by our analytical solution (8.57) and in our case $a_0 = 0.1$ which makes $C = \frac{2a_0}{a_0+1} \approx 0.18$, which is quite close to the estimate provided in Figure 8.6.

It is interesting to note that the standard Galerkin formulation of the problem results in a perfect approximation of u , while the flux $-au'$ is badly represented. On the other hand, for the mixed formulation, the flux is well approximated but u is approximated only to first order yielding a staircase approximation. These observations naturally suggest that we should employ P1 approximation of both u and its flux. We should then get a perfect approximation of both unknowns. This is however not possible. The linear system we obtain with P1 elements for both variables is singular.

This example shows that when we are solving systems of PDEs with several unknowns, we can not choose the approximation arbitrary. The polynomial spaces of the different unknowns have to be compatible and

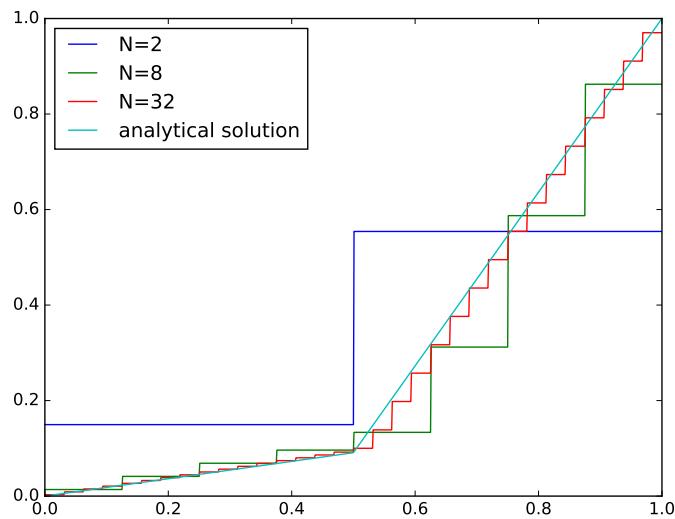


Fig. 8.5 Solution of the mixed Darcy problem with discontinuous coefficient for different number of elements N .

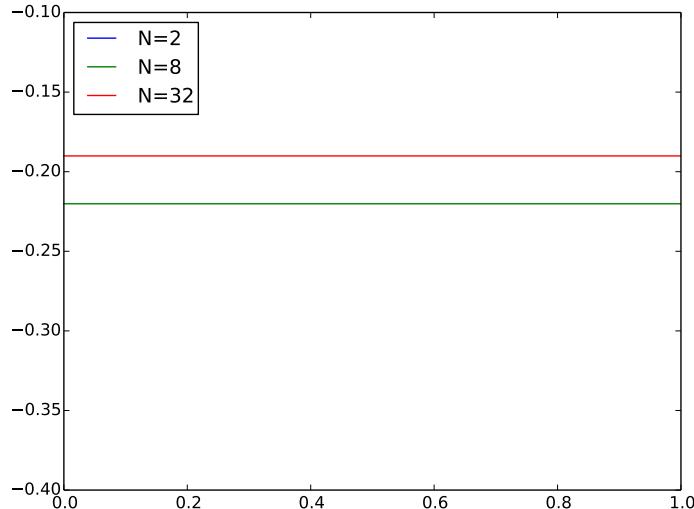


Fig. 8.6 The corresponding flux au' for the mixed Darcy problem with discontinuous coefficient for different number of elements N .

the accuracy of the different unknowns depend on each other. We will not discuss the reasons for the need of compatibility here as it is rather theoretical and beyond the scope of this book. Instead we refer the interested reader to [7, 6, 5, 9].

The complete code for the mixed Darcy example is

```

for N in Ns:
    mesh = UnitIntervalMesh(N)
    P1 = FiniteElement("CG", mesh.ufl_cell(), 1)
    P2 = FiniteElement("DG", mesh.ufl_cell(), 0)
    P1xP2 = P1 * P2
    W = FunctionSpace(mesh, P1xP2)
    u, p = TrialFunctions(W)
    v, q = TestFunctions(W)

    f = Constant(0)
    n = FacetNormal(mesh)
    a_coeff = A(degree=1, a0=a0)

    a = (1/a_coeff)*u*v*dx + u.dx(0)*q*dx - v.dx(0)*p*dx
    L = f*q*dx - p_bc*v*n[0]*ds

    up = Function(W)
    solve(a == L, up)

    u, p = up.split()

    import numpy
    a = numpy.array([0.0])
    b = numpy.array([0.0])
    xs = numpy.arange(0.0, 1.0, 0.001)
    ps = numpy.arange(0.0, 1.0, 0.001)
    for i in range(0,len(xs)):
        a[0] = xs[i]
        p.eval(b, a)
        ps[i] = b
    plt.plot(xs, ps)

CG1 = FunctionSpace(mesh, "CG", 1)
p_exact = project(uExact(a0=a0, degree=1), CG1)
p_exact4plot = numpy.array([p_exact(x) for x in xs])
plt.plot(xs, p_exact4plot)
legend = ["N=%d" % N for N in Ns]
legend.append("analytical solution")

plt.legend(legend, loc="upper left")
plt.savefig('darcy_a1D_mx.png'); plt.savefig('darcy_a1D_mx.pdf');

```

Here, we remark that we in order to plot the discontinuous solution properly we re-sampled the solution on a fine mesh. In general for

discontinuous elements one should be careful with the way the solution is plotted. Interpolating into a continuous field may not be desirable.

8.6 Exercises

Problem 8.1: Estimate order of convergence for the Cooling law

Consider the 1D Example of the fluid flow in a straight pipe coupled to heat conduction in Section 8.5. The example demonstrated fast convergence when using linear elements for both variables w and T . In this exercise we quantify the order of convergence. That is, we expect that

$$\begin{aligned}\|w - w_e\|_{L_2} &\leq C_w h^{\beta_w}, \\ \|T - T_e\|_{L_2} &\leq C_T h^{\beta_T},\end{aligned}$$

for some C_w , C_T , β_w and β_T . Assume therefore that

$$\begin{aligned}\|w - w_e\|_{L_2} &= C_w h^{\beta_w}, \\ \|T - T_e\|_{L_2} &= C_T h^{\beta_T},\end{aligned}$$

and estimate C_w , C_T , β_w and β_T .

Problem 8.2: Estimate order of convergence for the Cooling law

Repeat Exercise 8.1 with quadratic finite elements for both w and T .

Calculations to be continued...

Flexible implementations of boundary conditions

One quickly gets the impression that variational forms can handle only two types of boundary conditions: essential conditions where the unknown is prescribed, and natural conditions where flux terms integrated by parts allow specification of flux conditions. However, it is possible to treat much more general boundary conditions by adding their weak form. That is, one simply adds the variational formulation of some boundary condition $\mathcal{B}(u) = 0$: $\int_{\Omega_B} \mathcal{B}(u)v \, dx$, where Ω_B is some boundary, to the variational formulation of the PDE problem. Or using the terminology from Chapter 3: the residual of the boundary condition when the discrete solution is inserted is added to the residual of the entire problem. The present chapter shows underlying mathematical details.

9.1 Optimization with constraint

Suppose we have a function

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

and want to optimize its values, i.e., find minima and maxima. The condition for an optimum is that the derivatives vanish in all directions, which implies

$$\mathbf{n} \cdot \nabla f = 0 \quad \forall \mathbf{n} \in \mathbb{R}^2,$$

which further implies

$$\frac{\partial f}{\partial x} = 0, \quad \frac{\partial f}{\partial y} = 0.$$

These two equations are in general nonlinear and can have many solutions, one unique solution, or none. In our specific example, there is only one solution: $x = 0, y = 0$.

Now we want to optimize $f(x, y)$ under the constraint $y = 2 - x$. This means that only f values along the line $y = 2 - x$ are relevant, and we can imagine we view $f(x, y)$ along this line and want to find the optimum value.

9.1.1 Elimination of variables

Our f is obviously a function of one variable along the line. Inserting $y = 2 - x$ in $f(x, y)$ eliminates y and leads to f as function of x alone:

$$f(x, y = 2 - x) = 4 - 4x + 2x^2.$$

The condition for an optimum is

$$\frac{d}{dx}(4 - 4x + 2x^2) = -4 + 4x = 0,$$

so $x = 1$ and $y = 2 - x = 1$.

In the general case we have a scalar function $f(\mathbf{x})$, $\mathbf{x} = (x_0, \dots, x_m)$ with $n + 1$ constraints $g_i(\mathbf{x}) = 0$, $i = 0, \dots, n$. In theory, we could use the constraints to express $n + 1$ variables in terms of the remaining $m - n$ variables, but this is very seldom possible, because it requires us to solve the $g_i = 0$ symbolically with respect to $n + 1$ different variables.

9.1.2 Lagrange multiplier method

When we cannot easily eliminate variables using the constraint(s), the Lagrange multiplier method come to aid. Optimization of $f(x, y)$ under the constraint $g(x, y) = 0$ then consists in formulating the *Lagrangian*

$$\ell(x, y, \lambda) = f(x, y) + \lambda g(x, y),$$

where λ is the Lagrange multiplier, which is unknown. The conditions for an optimum is that

$$\frac{\partial \ell}{\partial x} = 0, \quad \frac{\partial \ell}{\partial y} = 0, \quad \frac{\partial \ell}{\partial \lambda} = 0.$$

In our example, we have

$$\ell(x, y, \lambda) = x^2 + y^2 + \lambda(y - 2 + x),$$

leading to the conditions

$$2x + \lambda = 0, \quad 2y + \lambda = 0, \quad y - 2 + x = 0.$$

This is a system of three linear equations in three unknowns with the solution

$$x = 1, \quad y = 1, \quad \lambda = 2.$$

In the general case with optimizing $f(\mathbf{x})$ subject to the constraints $g_i(\mathbf{x}) = 0$, $i = 0, \dots, n$, the Lagrangian becomes

$$\ell(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \sum_{j=0}^n \lambda_j g_j(\mathbf{x}),$$

with $\mathbf{x} = (x_0, \dots, x_m)$ and $\boldsymbol{\lambda} = (\lambda_0, \dots, \lambda_n)$. The conditions for an optimum are

$$\frac{\partial f}{\partial \mathbf{x}} = 0, \quad \frac{\partial f}{\partial \boldsymbol{\lambda}} = 0,$$

where

$$\frac{\partial f}{\partial \mathbf{x}} = 0 \Rightarrow \frac{\partial f}{\partial x_i} = 0, \quad i = 0, \dots, m.$$

Similarly, $\partial f / \partial \boldsymbol{\lambda} = 0$ leads to $n + 1$ equations $\partial f / \partial \lambda_i = 0$, $i = 0, \dots, n$.

9.1.3 Penalty method

Instead of incorporating the constraint exactly, as in the Lagrange multiplier method, the penalty method employs an approximation at the benefit of avoiding the extra Lagrange multiplier as unknown. The idea is to add the constraint squared, multiplied by a large prescribed number λ , called the penalty parameter,

$$\ell_\lambda(x, y) = f(x, y) + \frac{1}{2} \lambda (y - 2 + x)^2.$$

Note that λ is now a given (chosen) number. The ℓ_λ function is just a function of two variables, so the optimum is found by solving

$$\frac{\partial \ell_\lambda}{\partial x} = 0, \quad \frac{\partial \ell_\lambda}{\partial y} = 0.$$

Here we get

$$2x + \lambda(y - 2 + x) = 0, \quad 2y + \lambda(y - 2 + x) = 0.$$

The solution becomes

$$x = y = \frac{1}{1 - \frac{1}{2}\lambda^{-1}},$$

which we see approaches the correct solution $x = y = 1$ as $\lambda \rightarrow \infty$.

The penalty method for optimization of a multi-variate function $f(\mathbf{x})$ with constraints $g_i(\mathbf{x}) = 0$, $i = 0, \dots, n$, can be formulated as optimization of the unconstrained function

$$\ell_\lambda(\mathbf{x}) = f(\mathbf{x}) + \frac{1}{2}\lambda \sum_{j=0}^n (g_j(\mathbf{x}))^2.$$

Sometimes the symbol ϵ^{-1} is used for λ in the penalty method.

9.2 Optimization of functionals

The methods above for optimization of scalar functions of a finite number of variables can be generalized to optimization of functionals (functions of functions). We start with the specific example of optimizing

$$F(u) = \int_{\Omega} ||\nabla u||^2 dx - \int_{\Omega} fu dx - \int_{\partial\Omega_N} gu ds, \quad u \in V, \quad (9.1)$$

where $\Omega \subset \mathbb{R}^2$, and u and f are functions of x and y in Ω . The norm $||\nabla u||^2$ is defined as $u_x^2 + u_y^2$, with u_x denoting the derivative with respect to x . The vector space V contains the relevant functions for this problem, and more specifically, V is the Hilbert space H_0^1 consisting of all functions for which $\int_{\Omega} (u^2 + ||\nabla u||^2) dx$ is finite and $u = 0$ on $\partial\Omega_D$, which is some part of the boundary $\partial\Omega$ of Ω . The remaining part of the boundary is denoted by $\partial\Omega_N$ ($\partial\Omega_N \cup \partial\Omega_D = \partial\Omega$, $\partial\Omega_N \cap \partial\Omega_D = \emptyset$), over which $F(u)$

involves a line integral. Note that F is a mapping from any $u \in V$ to a real number in \mathbb{R} .

9.2.1 Classical calculus of variations

Optimization of the functional F makes use of the machinery from [variational calculus](#). The essence is to demand that the functional derivative of F with respect to u is zero. Technically, this is carried out by writing a general function $\tilde{u} \in V$ as $\tilde{u} = u + \epsilon v$, where u is the exact solution of the optimization problem, v is an arbitrary function in V , and ϵ is a scalar parameter. The functional derivative in the direction of v (also known as the [Gateaux derivative](#)) is defined as

$$\frac{\delta F}{\delta u} = \lim_{\epsilon \rightarrow 0} \frac{d}{d\epsilon} F(u + \epsilon v). \quad (9.2)$$

As an example, the functional derivative to the term $\int_{\Omega} f u \, dx$ in $F(u)$ is computed by finding

$$\frac{d}{d\epsilon} \int_{\Omega} f \cdot (u + \epsilon v) \, dx = \int_{\Omega} f v \, dx, \quad (9.3)$$

and then let ϵ go to zero (not strictly needed in this case because the term is linear in ϵ), which just results in $\int_{\Omega} f v \, dx$. The functional derivative of the other area integral becomes

$$\frac{d}{d\epsilon} \int_{\Omega} ((u_x + \epsilon v_x)^2 + (u_y + \epsilon v_y)^2) \, dx = \int_{\Omega} (2(u_x + \epsilon v_x)v_x + 2(u_y + \epsilon v_y)v_y) \, dx,$$

which leads to

$$\int_{\Omega} (u_x v_x + u_y v_y) \, dx = \int_{\Omega} \nabla u \cdot \nabla v \, dx, \quad (9.4)$$

as $\epsilon \rightarrow 0$.

The functional derivative of the boundary term becomes

$$\frac{d}{d\epsilon} \int_{\partial\Omega_N} g \cdot (u + \epsilon v) \, ds = \int_{\partial\Omega_N} g v \, ds, \quad (9.5)$$

for any ϵ . From (9.3)-(9.5) we then get the result

$$\frac{\delta F}{\delta u} = \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\Omega} fv \, dx - \int_{\partial\Omega_N} gv \, ds = 0. \quad (9.6)$$

Since v is arbitrary, this equation must hold $\forall v \in V$. Many will recognize (9.6) as the variational formulation of a Poisson problem, which can be directly discretized and solved by a finite element method.

Variational calculus goes one more step and derives a partial differential equation problem from (9.6), known as the [Euler-Lagrange equation](#) corresponding to optimization of $F(u)$. To find the differential equation, one manipulates the variational form (9.6) such that no derivatives of v appear and the equation (9.6) can be written as $\int_{\Omega} \mathcal{L}v \, dx = 0$, $\forall v \in V$, from which it follows that $\mathcal{L} = 0$ is the differential equation.

Performing integration by parts of the term $\int_{\Omega} \nabla u \cdot \nabla v \, dx$ in (9.6) moves the derivatives of v over to u :

$$\begin{aligned} \int_{\Omega} \nabla u \cdot \nabla v \, dx &= - \int_{\Omega} (\nabla^2 u)v \, dx + \int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds \\ &= - \int_{\Omega} (\nabla^2 u)v \, dx + \int_{\partial\Omega_D} \frac{\partial u}{\partial n} v \, ds + \int_{\partial\Omega_N} \frac{\partial u}{\partial n} v \, ds \\ &= - \int_{\Omega} (\nabla^2 u)v \, dx + \int_{\partial\Omega_D} \frac{\partial u}{\partial n} 0 \, ds + \int_{\partial\Omega_N} \frac{\partial u}{\partial n} v \, ds \\ &= - \int_{\Omega} (\nabla^2 u)v \, dx + \int_{\partial\Omega_N} \frac{\partial u}{\partial n} v \, ds. \end{aligned}$$

Using this rewrite in (9.6) gives

$$- \int_{\Omega} (\nabla^2 u)v \, dx + \int_{\partial\Omega_N} \frac{\partial u}{\partial n} v \, ds - \int_{\Omega} fv \, dx - \int_{\partial\Omega_N} gv \, ds,$$

which equals

$$\int_{\Omega} (\nabla^2 u + f)v \, dx + \int_{\partial\Omega_N} \left(\frac{\partial u}{\partial n} - g \right) v \, ds = 0.$$

This is to hold for any $v \in V$, which means that the integrands must vanish, and we get the famous Poisson problem

$$\begin{aligned} -\nabla^2 u &= f, \quad (x, y) \in \Omega, \\ u &= 0, \quad (x, y) \in \partial\Omega_D, \\ \frac{\partial u}{\partial n} &= g, \quad (x, y) \in \partial\Omega_N. \end{aligned}$$

Some remarks.

- Specifying u on some part of the boundary ($\partial\Omega_D$) implies a specification of $\partial u/\partial n$ on the rest of the boundary. In particular, if such a specification is not explicitly done, the mathematics above implies $\partial u/\partial n = 0$ on $\partial\Omega_N$.
- If a non-zero condition on $u = u_0$ on $\partial\Omega_D$ is wanted, one can write $u = u_0 + \bar{u}$ and express the functional F in terms of \bar{u} , which obviously must vanish on $\partial\Omega_D$ since u is the exact solution that is u_0 on $\partial\Omega_D$.
- The boundary conditions on u must be implemented in the space V , i.e., we can only work with functions that *must* be zero on $\partial\Omega_D$ (so-called *essential boundary condition*). The condition involving $\partial u/\partial n$ is easier to implement since it is just a matter of computing a line integral.
- The solution is not unique if $\partial\Omega_D = \emptyset$ (any solution $u + \text{const}$ is also a solution).

9.2.2 Penalty and Nitsche's methods for optimization with constraints

The attention is now on optimization of a functional $F(u)$ with a given constraint that $u = u_N$ on $\partial\Omega_N$. That is, we want to set Dirichlet conditions weakly on the Neumann part of the boundary. We could, of course, just extend the Dirichlet condition on u in the previous set-up by saying that $\partial\Omega_D$ is the complete boundary $\partial\Omega$ and that u takes on the values of 0 and u_N at the different parts of the boundary. However, this also implies that all the functions in V must vanish on the entire boundary. We want to relax this condition (and by relaxing it, we will derive a method that can be used for many other types of boundary conditions!). The goal is, therefore, to incorporate $u = u_N$ on $\partial\Omega_N$

without demanding anything from the functions in V . We can achieve this by enforcing the constraint

$$\int_{\partial\Omega_N} |u - u_N| \, ds = 0. \quad (9.7)$$

However, this constraint is cumbersome to implement. Note that the absolute sign here is needed as in general there are many functions u such that $\int_{\partial\Omega_N} u - u_N \, ds = 0$.

A penalty method. The idea is to add a penalization term $\frac{1}{2}\lambda(u - u_N)^2$, integrated over the boundary $\partial\Omega_N$, to the functional $F(u)$, just as we do in the penalty method (the factor $\frac{1}{2}$ can be incorporated in λ , but we keep it because it makes the final result look nicer).

The condition $\partial u / \partial n = g$ on $\partial\Omega_N$ is no longer relevant, so we replace the g by the unknown $\partial u / \partial n$ in the boundary integral term in (9.1). The new functional becomes

$$F(u) = \int_{\Omega} ||\nabla u||^2 \, dx - \int_{\Omega} f u \, dx - \frac{1}{2} \int_{\partial\Omega_N} \lambda(u - u_N)^2 \, ds, \quad u \in V, \quad (9.8)$$

In $F(\tilde{u})$, insert $\tilde{u} = u + \epsilon v$, differentiate with respect to ϵ , and let $\epsilon \rightarrow 0$. The result becomes

$$\frac{\delta F}{\delta u} = \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\Omega} f v \, dx - \int_{\partial\Omega_N} \lambda(u - u_N)v \, ds = 0. \quad (9.9)$$

We may then ask ourselves which equation and which boundary conditions is solved for when minimizing this functional. We therefore do integration by parts in order to obtain the strong formulation. We remember the Gauss-Green's lemma:

$$\int_{\Omega} (\nabla^2 u)v \, dx = - \int_{\Omega} \nabla u \cdot \nabla v \, dx + \int_{\partial\Omega_N} \frac{\partial u}{\partial n} v \, ds.$$

Hence, from (9.9) and using Gauss-Green's lemma, we obtain:

$$\frac{\delta F}{\delta u} = \int_{\Omega} -\Delta u v \, dx - \int_{\Omega} f v \, dx - \int_{\partial\Omega_N} \lambda(u - u_N)v \, ds + \int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds = 0. \quad (9.10)$$

In other words, our problem on strong form reads:

$$\begin{aligned} -\Delta u &= f, \quad x \in \Omega, \\ \frac{\partial u}{\partial n} &= \lambda(u - u_N), \quad x \in \partial\Omega_N. \end{aligned}$$

This means that the minimizing problem corresponds to solving a problem with Robin conditions.

Nitsche's method consists of changing the above functional in order to obtain the *true Dirichlet conditions*. As we saw in the previous calculations, integration by parts introduced the term $\frac{\partial u}{\partial n}$ in the strong formulations. Hence, a natural idea is to subtract such a term from the functional.

$$F(u) = \int_{\Omega} ||\nabla u||^2 dx - \int_{\Omega} fu dx - \int_{\partial\Omega_N} \frac{\partial u}{\partial n}(u - u_N) ds + \frac{1}{2} \int_{\partial\Omega_N} \lambda(u - u_N)^2 ds. \quad (9.11)$$

In $F(\tilde{u})$, insert $\tilde{u} = u + \epsilon v$, differentiate with respect to ϵ , and let $\epsilon \rightarrow 0$. The result becomes

$$\begin{aligned} \frac{\delta F}{\delta u} &= \int_{\Omega} \nabla u \cdot \nabla v dx - \int_{\Omega} fv dx \\ &\quad - \int_{\partial\Omega_N} \frac{\partial u}{\partial n} v ds + \int_{\partial\Omega_N} \frac{\partial v}{\partial n}(u - u_N) ds + \int_{\partial\Omega_N} \lambda(u - u_N)v ds = 0. \end{aligned}$$

If we again perform integration by parts to obtain the strong form and boundary conditions, we get

$$\frac{\delta F}{\delta u} = \int_{\Omega} -\Delta u v dx - \int_{\Omega} fv dx - \int_{\partial\Omega_N} \frac{\partial v}{\partial n}(u - u_N) ds + \int_{\partial\Omega_N} \lambda(u - u_N)v ds = 0.$$

In other words, our problem on strong form reads:

$$\begin{aligned} -\Delta u &= f, \quad x \in \Omega, \\ u &= u_N, \quad x \in \partial\Omega_N. \end{aligned}$$

and the condition $u = u_N$ is enforced both in terms of the penalty parameter λ by the term $\int_{\partial\Omega_N} \lambda(u - u_N)v ds$ and in terms of equations

involving the derivatives of the test function v , $\int_{\partial\Omega_N} \frac{\partial v}{\partial n}(u - u_N) ds$.

One may question why two terms are needed in order to enforce the boundary condition. In general this may not be needed and the penalty term may sometimes be dropped. However, the advantage of including the penalty term is that it keeps the functional convex and the bilinear form becomes both positive and symmetric.

We summarize the final formulation in terms of a weak formulation:

$$a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega_N} \frac{\partial u}{\partial n} v \, ds - \int_{\partial\Omega_N} \frac{\partial v}{\partial n} u \, ds + \int_{\partial\Omega_N} \lambda u v \, ds, \quad (9.12)$$

$$L(v) = \int_{\Omega} fv \, dx - \int_{\partial\Omega_N} \frac{\partial v}{\partial n} u_N \, ds + \int_{\partial\Omega_N} \lambda u_N v \, ds. \quad (9.13)$$

9.2.3 Lagrange multiplier method for optimization with constraints

We consider the same problem as in Section 9.2.2, but this time we want to apply a Lagrange multiplier method so we can solve for a *multiplier function* rather than specifying a large number for a penalty parameter and getting an approximate result.

The functional to be optimized reads

$$F(u) = \int_{\Omega} \|\nabla u\|^2 \, dx - \int_{\Omega} fu \, dx - \int_{\partial\Omega_N} u_N \, ds + \int_{\partial\Omega_N} \lambda(u - u_N) \, ds, \quad u \in V.$$

Here we have two unknown functions: $u \in V$ in Ω and $\lambda \in Q$ on $\partial\Omega_N$. The optimization criteria are

$$\frac{\delta F}{\delta u} = 0, \quad \frac{\delta F}{\delta \lambda} = 0.$$

We write $\tilde{u} = u + \epsilon_u v$ and $\tilde{\lambda} = \lambda + \epsilon_\lambda p$, where v is an arbitrary function in V and p is an arbitrary function in Q . Notice that V is here a usual function space with functions defined on Ω , while on the other hand is a function space defined only on the surface Ω_N . We insert the expressions for \tilde{u} and $\tilde{\lambda}$ for u and λ and compute

$$\begin{aligned}\frac{\delta F}{\delta u} &= \lim_{\epsilon_u \rightarrow 0} \frac{dF}{d\epsilon_u} = \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\Omega} fv \, dx - \int_{\partial\Omega_N} \frac{\partial u}{\partial n} v \, ds + \int_{\partial\Omega_N} \lambda(u - u_N) \, ds = 0, \\ \frac{\delta F}{\delta \lambda} &= \lim_{\epsilon_\lambda \rightarrow 0} \frac{dF}{d\epsilon_\lambda} = \int_{\partial\Omega_N} (u - u_N)p \, ds = 0.\end{aligned}$$

These equations can be written as a linear system of equations: Find $u, \lambda \in V \times Q$ such that

$$\begin{aligned}a(u, v) + b(\lambda, v) &= L_u(v), \\ b(u, p) &= L_\lambda(\lambda),\end{aligned}$$

for all test functions $v \in V$ and $p \in Q$ and

$$\begin{aligned}a(u, v) &= \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega_N} \frac{\partial u}{\partial n} v \, ds, \\ b(\lambda, v) &= \int_{\partial\Omega_N} \lambda v \, ds, \\ L_u(v) &= \int_{\Omega} fv \, dx, \\ L_\lambda(\lambda) &= \int_{\partial\Omega_N} u_N \lambda \, ds.\end{aligned}$$

Letting $u = \sum_{j \in \mathcal{I}_s} c_j \psi_j^{(u)}$, $\lambda = \sum_{j \in \mathcal{I}_s} c_j \psi_j^{(\lambda)}$, $v = \psi_i^{(v)}$, and $p = \psi_i^{(p)}$, we obtain the following system of linear equations

$$A c = \begin{bmatrix} A^{(u,u)} & A^{(\lambda,u)} \\ A^{(u,\lambda)} & 0 \end{bmatrix} \begin{bmatrix} c^{(u)} \\ c^{(\lambda)} \end{bmatrix} = \begin{bmatrix} b^{(u)} \\ b^{(\lambda)} \end{bmatrix} = b,$$

where

$$\begin{aligned}A_{i,j}^{(u,u)} &= a(\psi_j^{(u)}, \psi_i^{(u)}), \\ A_{i,j}^{(u,\lambda)} &= b(\psi_j^{(u)}, \psi_i^{(\lambda)}), \\ A_{i,j}^{(\lambda,u)} &= A_{j,i}^{(u,\lambda)}, \\ b_i^{(u)} &= L_u(\psi_i^{(u)}), \\ b_i^{(\lambda)} &= L_\lambda(\psi_i^{(\lambda)}), .\end{aligned}$$

9.2.4 Example: 1D problem

Nitsche method. Let us do hand calculations to demonstrate weakly enforced boundary conditions via a Nitsche's method and via the Lagrange multiplier method. We study the simple problem $-u'' = 2$ on $[0, 1]$, c.f. (9.12)-(9.13), with boundary conditions $u(0) = 0$ and $u(1) = 1$.

$$a(u, v) = \int_0^1 u_x v_x \, dx - [u_x v]_0^1 - [v_x u]_0^1 + [\lambda u v]_0^1$$

$$L(v) = \int_0^1 f v \, dx - [v_x u_N]_0^1 + [\lambda u_N v]_0^1.$$

A uniform mesh with nodes $x_i = i\Delta x$ is introduced, numbered from left to right: $i = 0, \dots, N_x$. The approximate value of u at x_i is denoted by c_i , and in general the approximation to u is $\sum_{i=0}^{N_x} \varphi_i(x)c_i$.

The elements at the boundaries needs special attention. Let us consider the element 0 defined on $[0, h]$. The basis functions are $\varphi_0(x) = 1 - x/h$ and $\varphi_1(x) = x/h$. Hence, $\varphi_0|_{x=0} = 1$, $\varphi'_0|_{x=0} = -1/h$, $\varphi_1|_{x=0} = 0$, and $\varphi'_1|_{x=0} = 1/h$. Therefore, for element 0 we obtain the element matrix

$$A_{0,0}^{(0)} = \lambda + \frac{3}{h},$$

$$A_{0,1}^{(0)} = -\frac{2}{h},$$

$$A_{1,0}^{(0)} = -\frac{2}{h},$$

$$A_{1,1}^{(0)} = \frac{1}{h}.$$

The interior elements ($e = 1 \dots N_e - 2$) result in the following element matrices

$$A_{0,0}^{(e)} = \frac{1}{h}, \quad A_{0,1}^{(e)} = -\frac{1}{h},$$

$$A_{1,0}^{(e)} = -\frac{1}{h}, \quad A_{1,1}^{(e)} = \frac{1}{h}.$$

While the element at the boundary $x = 1$ result in a element matrix similar to A^0 except that 0 and 1 are swapped. The calculations are straightforward in `sympy`

```
import sympy as sym
x, h, lam = sym.symbols("x h \lambda")
basis = [1 - x/h, x/h]
```

```

for i in range(len(basis)):
    phi_i = basis[i]
    for j in range(len(basis)):
        phi_j = basis[j]
        a = sym.integrate(sym.diff(phi_i, x)*sym.diff(phi_j, x), (x, 0, h))
        a -= (sym.diff(phi_i, x)*phi_j).subs(x,0)
        a -= (sym.diff(phi_j, x)*phi_i).subs(x,0)
        a += (lam*phi_j*phi_i).subs(x,0)

```

In the symmetric variant of Nitsche's method that we have presented here, there is a need for a positive penalty parameter λ in order for the method to work. A natural question is therefore how sensitive the results are to this penalty parameter. The following code implements Nitsche's method in FEniCS and tests various penalty parameters.

```

import matplotlib.pyplot as plt
from dolfin import *

mesh = UnitIntervalMesh(100)
V = FunctionSpace(mesh, "Lagrange", 1)
u = TrialFunction(V)
v = TestFunction(V)

lams = [1.001, 1.01, 1.1, 2, 10, 100]
for lam in lams:
    lam = Constant(lam)
    h = CellDiameter(mesh)
    n = FacetNormal(mesh)
    f = Expression("-12*pow(x[0], 2)", degree=2)
    u0 = Expression("pow(x[0],4)", degree=4)

    a = dot(grad(v), grad(u))*dx \
        - dot(grad(v), u*n)*ds \
        - dot(v*n, grad(u))*ds \
        + (lam/h)*v*u*ds
    L = v*f*dx - u0*dot(grad(v), n)*ds + (lam/h)*u0*v*ds

    U = Function(V)
    solve(a == L, U)

    plt.plot(V.tabulate_dof_coordinates(), U.vector().get_local())

plt.legend(["lam=%4.3f" %lam for lam in lams], loc=2)
plt.show()

```

Figure 9.1 displays the results obtained by running the above script. As we see in Figure 9.1 and the zoom in Figure 9.2, Nitsche's method is not very sensitive to the value of the penalty parameter as long as it is above a certain threshold. In our 1D example, the threshold seems to be $1/h$. Setting the parameter to $1/h$ or lower makes the solution blow up

and there are some artifacts when setting the parameter very close to $1/h$ but we see that $2/h$, $10/h$ and $100/h$ gives produce visually identical solutions. This is generally, the case with Nitsche's method although the threshold may depend on the application.

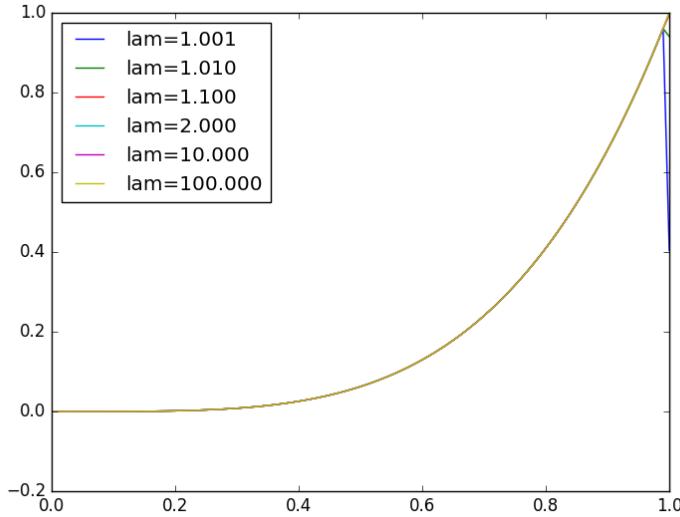


Fig. 9.1 Solution of the Poisson problem using Nitsche's method for various penalty parameters.

Lagrange multiplier method. For the Lagrange multiplier method we need a function space Q defined on the boundary of the domain. In 1D with $\Omega = (0, 1)$ the boundary is $x = 0$ and $x = 1$. Hence, Q can be spanned by two basis functions λ_0 and λ_1 . These functions should be such that $\lambda_0 = 1$ for $x = 0$ and zero everywhere else, while $\lambda_1 = 1$ for $x = 1$ and zero everywhere else. Hence, we may use the following function

$$\lambda(x) = \lambda_0 \varphi_0(x) + \lambda_{N_x} \varphi_{N_x}(x).$$

9.2.5 Example: adding a constraint in a Neumann problem

The Poisson Neumann problem reads:

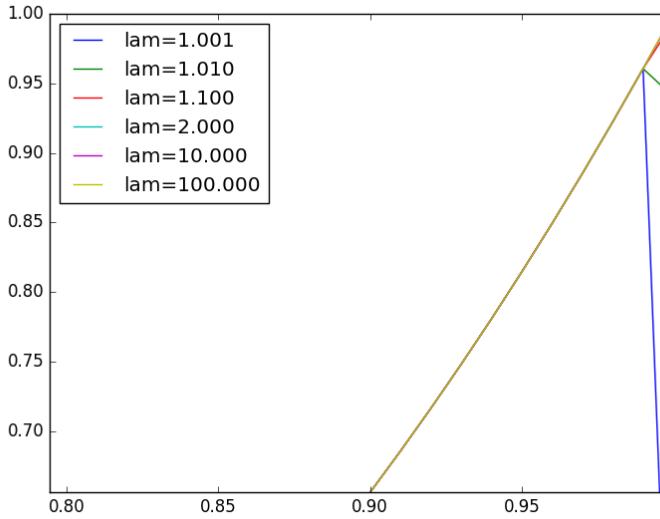


Fig. 9.2 A zoom towards the right boundary of the figure in 9.1.

$$\begin{aligned} -\Delta u &= f, \quad x \in \Omega \\ \frac{\partial u}{\partial n} &= 0, \quad x \in \partial\Omega \end{aligned},$$

It is a singular problem with a one-dimensional kernel. To see this, we remark that if u is a solution to the problem then $\hat{u} = u + C$, where C is any number, is also a solution since $-\Delta \hat{u} = -\Delta u - \Delta C = -\Delta u = f$ and $\frac{\partial \hat{u}}{\partial n} = \frac{\partial u}{\partial n} + \frac{\partial C}{\partial n} = \frac{\partial u}{\partial n} = 0$. As the PDE is singular, also the corresponding finite element matrix will be singular and this frequently (but not always) cause problems when solving the linear system.

There are two main remedies for this problem 1) to add an equation that fixates the solution in one point to the linear system, i.e., set $u(x) = 0$ in some point x either at the boundary or in the interior and 2) to enforce that $\int_{\Omega} u \, dx = 0$ by using a Lagrange multiplier. The first method is the most used method as it is easy to implement. The method often works well but it is not a bullet-proof procedure, as we will illustrate.

The following code implements the Poisson Neumann problem in FEniCS and fixates the solution in one point.

```
from dolfin import *
def boundary(x, on_boundary):
```

```

if near(x[0],0.3) and near(x[1],0): return True
return False

mesh = UnitSquareMesh(10,10)
V = FunctionSpace(mesh, "Lagrange", 1)
u = TrialFunction(V)
v = TestFunction(V)

n = FacetNormal(mesh)
f = Expression("pow(3.14,2)*cos(3.14*x[0])+1", degree=4)
ue = Expression("cos(3.14*x[0])-1", degree=4)
du = Expression(["3.14*sin(3.14*x[0])", "0"], degree=4)

a = dot(grad(v), grad(u))*dx
L = v*f*dx + inner(du,n)*v*ds

point_condition = DirichletBC(V, ue, boundary, "pointwise")
u = Function(V, name="u")
solve(a == L, u, point_condition)

```

We remark that we fixate the solution in one point here by using `DirichletBC` where we specify `pointwise` application and further that we in this example know that $(0.3, 0)$ is a vertex in the mesh.

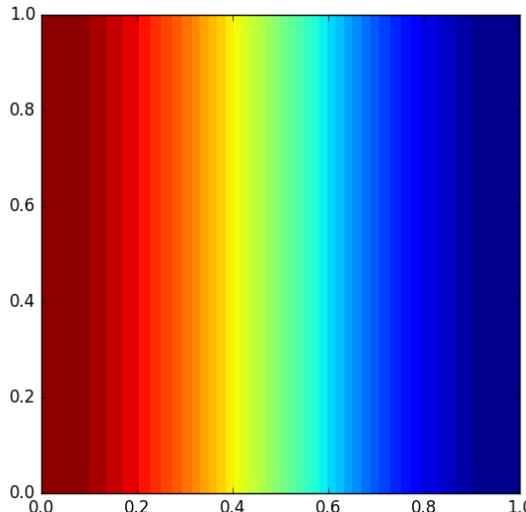


Fig. 9.3 The exact solution of the Poisson Neumann problem.

Figure 9.3 displays the exact solution of the Poisson problem while Figure 9.4 shows the resulting solution of the problem implemented in

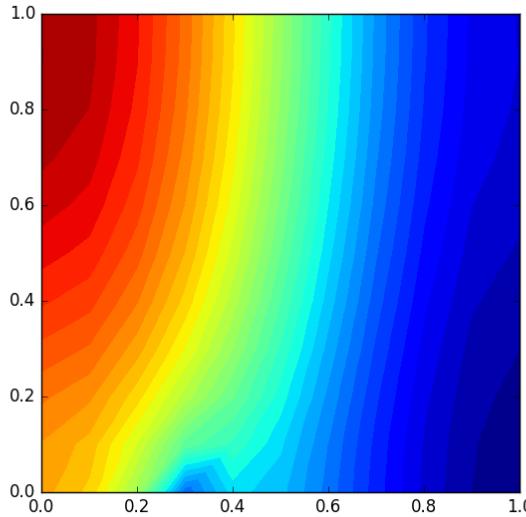


Fig. 9.4 The computed solution of the Poisson Neumann problem with u being fixated at $(0.3, 0)$.

the FEniCS code listed above. Clearly, the numerical solution is wrong and our approach of fixating the solution in $(0.3, 0)$ has destroyed the solution in large parts of the domain. In our case the problem is however easily fixed by ensuring that the true solution u and right-hand side f satisfy $\int_{\Omega} u \, dx = 0$ and $\int_{\Omega} f \, dx = 0$, respectively, and that the fixation is compatible with this. While ensuring compatibility is quite easy for scalar PDE problems, it may be more difficult for systems of PDEs where determining the appropriate conditions is more involved.

A more general strategy is to remove the kernel by a Lagrange multiplier, requiring that $\int_{\Omega} u \, dx = 0$. The resulting equations can be written as a linear system of equations: Find $u, \lambda \in V \times \mathbb{R}$ such that

$$\begin{aligned} a(u, v) + b(v, c) &= L_u(v), \\ b(u, d) &= L_c(d), \end{aligned}$$

for all test functions $v \in V$ and $d \in \mathbb{R}$ and

$$\begin{aligned} a(u, v) &= \int_{\Omega} \nabla u \cdot \nabla v \, dx \\ b(v, c) &= \int_{\Omega} cv \, dx, \\ L_u(v) &= \int_{\Omega} fv \, dx, \\ L_c(d) &= 0. \end{aligned}$$

Letting $u = \sum_{j \in \mathcal{I}_s} c_j \psi_j^{(u)}$, $v = \psi_i^{(v)}$, and c, d be two arbitrary constants, we obtain the following system of linear equations

$$A c = \begin{bmatrix} A^{(u,u)} & A^{(c,u)} \\ A^{(u,c)} & 0 \end{bmatrix} \begin{bmatrix} c^{(u)} \\ c \end{bmatrix} = \begin{bmatrix} b^{(u)} \\ 0 \end{bmatrix} = b,$$

where

$$\begin{aligned} A_{i,j}^{(u,u)} &= a(\psi_j^{(u)}, \psi_i^{(u)}), \\ A_{i,j}^{(u,c)} &= b(\psi_j^{(u)}, c), \\ A_{i,j}^{(c,u)} &= A_{j,i}^{(u,c)}, \\ b_i^{(u)} &= L_u(\psi_i^{(u)}). \end{aligned}$$

The corresponding code in FEniCS resembles the mathematical problem:

```
from dolfin import *

mesh = UnitSquareMesh(10,10)
L = FiniteElement("Lagrange", mesh.ufl_cell(), 1)
R = FiniteElement("R", mesh.ufl_cell(), 0)
W = FunctionSpace(mesh, L*R)

(u,c) = TrialFunction(W)
(v,d) = TestFunction(W)

n = FacetNormal(mesh)

f = Expression("pow(3.14,2)*cos(3.14*x[0])+1", degree=4)
ue = Expression("cos(3.14*x[0])-1", degree=4)
du = Expression(["3.14*sin(3.14*x[0])", "0"], degree=4)

a = dot(grad(v), grad(u))*dx + c*v*dx + d*u*dx
L = v*f*dx + inner(du,n)*v*ds
```

```
w = Function(W)
solve(a == L, w)
(u,c) = w.split()
```

The solution produced by running this script is visually identical to the exact solution.

10.1 Introduction of basic concepts

10.1.1 Linear versus nonlinear equations

Algebraic equations. A linear, scalar, algebraic equation in x has the form

$$ax + b = 0,$$

for arbitrary real constants a and b . The unknown is a number x . All other algebraic equations, e.g., $x^2 + ax + b = 0$, are nonlinear. The typical feature in a nonlinear algebraic equation is that the unknown appears in products with itself, like x^2 or in functions that are infinite sums of products, like $e^x = 1 + x + \frac{1}{2}x^2 + \frac{1}{3!}x^3 + \dots$.

We know how to solve a linear algebraic equation, $x = -b/a$, but there are no general closed formulas for finding the exact solutions of nonlinear algebraic equations, except for very special cases (quadratic equations constitute a primary example). A nonlinear algebraic equation may have no solution, one solution, or many solutions. The tools for solving nonlinear algebraic equations are *iterative methods*, where we construct a series of linear equations, which we know how to solve, and hope that the solutions of the linear equations converge to a solution of the nonlinear equation we want to solve. Typical methods for nonlinear algebraic equation equations are Newton's method, the Bisection method, and the Secant method.

Differential equations. The unknown in a differential equation is a function and not a number. In a linear differential equation, all terms involving the unknown function are linear in the unknown function or its derivatives. Linear here means that the unknown function, or a derivative of it, is multiplied by a number or a known function. All other differential equations are non-linear.

The easiest way to see if an equation is nonlinear, is to spot nonlinear terms where the unknown function or its derivatives are multiplied by each other. For example, in

$$u'(t) = -a(t)u(t) + b(t),$$

the terms involving the unknown function u are linear: u' contains the derivative of the unknown function multiplied by unity, and au contains the unknown function multiplied by a known function. However,

$$u'(t) = u(t)(1 - u(t)),$$

is nonlinear because of the term $-u^2$ where the unknown function is multiplied by itself. Also

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = 0,$$

is nonlinear because of the term uu_x where the unknown function appears in a product with its derivative. (Note here that we use different notations for derivatives: u' or du/dt for a function $u(t)$ of one variable, $\frac{\partial u}{\partial t}$ or u_t for a function of more than one variable.)

Another example of a nonlinear equation is

$$u'' + \sin(u) = 0,$$

because $\sin(u)$ contains products of u , which becomes clear if we expand the function in a Taylor series:

$$\sin(u) = u - \frac{1}{3}u^3 + \dots$$

Mathematical proof of linearity

To really prove mathematically that some differential equation in an unknown u is linear, show for each term $T(u)$ that with $u = au_1 + bu_2$ for constants a and b ,

$$T(au_1 + bu_2) = aT(u_1) + bT(u_2).$$

For example, the term $T(u) = (\sin^2 t)u'(t)$ is linear because

$$\begin{aligned} T(au_1 + bu_2) &= (\sin^2 t)(au_1(t) + bu_2(t))' \\ &= a(\sin^2 t)u'_1(t) + b(\sin^2 t)u'_2(t) \\ &= aT(u_1) + bT(u_2). \end{aligned}$$

However, $T(u) = \sin u$ is nonlinear because

$$T(au_1 + bu_2) = \sin(au_1 + bu_2) \neq a \sin u_1 + b \sin u_2.$$

10.1.2 A simple model problem

A series of forthcoming examples will explain how to tackle nonlinear differential equations with various techniques. We start with the (scaled) logistic equation as model problem:

$$u'(t) = u(t)(1 - u(t)). \quad (10.1)$$

This is a nonlinear ordinary differential equation (ODE) which will be solved by different strategies in the following. Depending on the chosen time discretization of (10.1), the mathematical problem to be solved at every time level will either be a linear algebraic equation or a nonlinear algebraic equation. In the former case, the time discretization method transforms the nonlinear ODE into linear subproblems at each time level, and the solution is straightforward to find since linear algebraic equations are easy to solve. However, when the time discretization leads to nonlinear algebraic equations, we cannot (except in very rare cases) solve these without turning to approximate, iterative solution methods.

The next subsections introduce various methods for solving nonlinear differential equations, using (10.1) as model. We shall go through the following set of cases:

- explicit time discretization methods (with no need to solve nonlinear algebraic equations)
- implicit Backward Euler time discretization, leading to nonlinear algebraic equations solved by
 - an exact analytical technique
 - Picard iteration based on manual linearization
 - a single Picard step
 - Newton's method
- implicit Crank-Nicolson time discretization and linearization via a geometric mean formula

Thereafter, we compare the performance of the various approaches. Despite the simplicity of (10.1), the conclusions reveal typical features of the various methods in much more complicated nonlinear PDE problems.

10.1.3 Linearization by explicit time discretization

Time discretization methods are divided into explicit and implicit methods. Explicit methods lead to a closed-form formula for finding new values of the unknowns, while implicit methods give a linear or nonlinear system of equations that couples (all) the unknowns at a new time level. Here we shall demonstrate that explicit methods may constitute an efficient way to deal with nonlinear differential equations.

The Forward Euler method is an explicit method. When applied to (10.1), sampled at $t = t_n$, it results in

$$\frac{u^{n+1} - u^n}{\Delta t} = u^n(1 - u^n),$$

which is a *linear* algebraic equation for the unknown value u^{n+1} that we can easily solve:

$$u^{n+1} = u^n + \Delta t u^n(1 - u^n).$$

The nonlinearity in the original equation poses in this case no difficulty in the discrete algebraic equation. Any other explicit scheme in time will also give only linear algebraic equations to solve. For example, a typical 2nd-order Runge-Kutta method for (10.1) leads to the following formulas:

$$\begin{aligned} u^* &= u^n + \Delta t u^n (1 - u^n), \\ u^{n+1} &= u^n + \Delta t \frac{1}{2} (u^n (1 - u^n) + u^* (1 - u^*)) . \end{aligned}$$

The first step is linear in the unknown u^* . Then u^* is known in the next step, which is linear in the unknown u^{n+1} . For this scheme we have an explicit formula for the unknown and the scheme is therefore called an *explicit scheme*.

10.1.4 Exact solution of nonlinear algebraic equations

Switching to a Backward Euler scheme for (10.1),

$$\frac{u^n - u^{n-1}}{\Delta t} = u^n (1 - u^n), \quad (10.2)$$

results in a nonlinear algebraic equation for the unknown value u^n . The equation is of quadratic type:

$$\Delta t (u^n)^2 + (1 - \Delta t) u^n - u^{n-1} = 0,$$

and may be solved exactly by the well-known formula for such equations. Before we do so, however, we will introduce a shorter, and often cleaner, notation for nonlinear algebraic equations at a given time level. The notation is inspired by the natural notation (i.e., variable names) used in a program, especially in more advanced partial differential equation problems. The unknown in the algebraic equation is denoted by u , while $u^{(1)}$ is the value of the unknown at the previous time level (in general, $u^{(\ell)}$ is the value of the unknown ℓ levels back in time). The notation will be frequently used in later sections. What is meant by u should be evident from the context: u may be 1) the exact solution of the ODE/PDE problem, 2) the numerical approximation to the exact solution, or 3) the unknown solution at a certain time level.

The quadratic equation for the unknown u^n in (10.2) can, with the new notation, be written

$$F(u) = \Delta t u^2 + (1 - \Delta t) u - u^{(1)} = 0. \quad (10.3)$$

The solution is readily found to be

$$u = \frac{1}{2\Delta t} \left(-1 + \Delta t \pm \sqrt{(1 - \Delta t)^2 - 4\Delta t u^{(1)}} \right). \quad (10.4)$$

Now we encounter a fundamental challenge with nonlinear algebraic equations: the equation may have more than one solution. How do we pick the right solution? This is in general a hard problem. In the present simple case, however, we can analyze the roots mathematically and provide an answer. The idea is to expand the roots in a series in Δt and truncate after the linear term since the Backward Euler scheme will introduce an error proportional to Δt anyway. Using `sympy` we find the following Taylor series expansions of the roots:

```
>>> import sympy as sym
>>> dt, u_1, u = sym.symbols('dt u_1 u')
>>> r1, r2 = sym.solve(dt*u**2 + (1-dt)*u - u_1, u) # find roots
>>> r1
(dt - sqrt(dt**2 + 4*dt*u_1 - 2*dt + 1) - 1)/(2*dt)
>>> r2
(dt + sqrt(dt**2 + 4*dt*u_1 - 2*dt + 1) - 1)/(2*dt)
>>> print(r1.series(dt, 0, 2)) # 2 terms in dt, around dt=0
-1/dt + 1 - u_1 + dt*(u_1**2 - u_1) + O(dt**2)
>>> print(r2.series(dt, 0, 2))
u_1 + dt*(-u_1**2 + u_1) + O(dt**2)
```

We see that the `r1` root, corresponding to a minus sign in front of the square root in (10.4), behaves as $1/\Delta t$ and will therefore blow up as $\Delta t \rightarrow 0$! Since we know that u takes on finite values, actually it is less than or equal to 1, only the `r2` root is of relevance in this case: as $\Delta t \rightarrow 0$, $u \rightarrow u^{(1)}$, which is the expected result.

For those who are not well experienced with approximating mathematical formulas by series expansion, an alternative method of investigation is simply to compute the limits of the two roots as $\Delta t \rightarrow 0$ and see if a limit unreasonable:

```
>>> print(r1.limit(dt, 0))
-oo
>>> print(r2.limit(dt, 0))
u_1
```

10.1.5 Linearization

When the time integration of an ODE results in a nonlinear algebraic equation, we must normally find its solution by defining a sequence of linear equations and hope that the solutions of these linear equations converge to the desired solution of the nonlinear algebraic equation. Usually, this means solving the linear equation repeatedly in an iterative fashion.

Alternatively, the nonlinear equation can sometimes be approximated by one linear equation, and consequently there is no need for iteration.

Constructing a linear equation from a nonlinear one requires *linearization* of each nonlinear term. This can be done manually as in Picard iteration, or fully algorithmically as in Newton's method. Examples will best illustrate how to linearize nonlinear problems.

10.1.6 Picard iteration

Let us write (10.3) in a more compact form

$$F(u) = au^2 + bu + c = 0,$$

with $a = \Delta t$, $b = 1 - \Delta t$, and $c = -u^{(1)}$. Let u^- be an available approximation of the unknown u .

Then we can linearize the term u^2 simply by writing $u^- u$. The resulting equation, $\hat{F}(u) = 0$, is now linear and hence easy to solve:

$$F(u) \approx \hat{F}(u) = au^- u + bu + c = 0.$$

Since the equation $\hat{F} = 0$ is only approximate, the solution u does not equal the exact solution u_e of the exact equation $F(u_e) = 0$, but we can hope that u is closer to u_e than u^- is, and hence it makes sense to repeat the procedure, i.e., set $u^- = u$ and solve $\hat{F}(u) = 0$ again. There is no guarantee that u is closer to u_e than u^- , but this approach has proven to be effective in a wide range of applications.

The idea of turning a nonlinear equation into a linear one by using an approximation u^- of u in the nonlinear terms is a widely used approach that goes under many names: *fixed-point iteration*, the method of *successive substitutions*, *nonlinear Richardson iteration*, and *Picard iteration*. We will stick to the latter name.

Picard iteration for solving the nonlinear equation arising from the Backward Euler discretization of the logistic equation can be written as

$$u = -\frac{c}{au^- + b}, \quad u^- \leftarrow u.$$

The \leftarrow symbol means assignment (we set u^- equal to the value of u). The iteration is started with the value of the unknown at the previous time level: $u^- = u^{(1)}$.

Some prefer an explicit iteration counter as superscript in the mathematical notation. Let u^k be the computed approximation to the solution in iteration k . In iteration $k + 1$ we want to solve

$$au^k u^{k+1} + bu^{k+1} + c = 0 \quad \Rightarrow \quad u^{k+1} = -\frac{c}{au^k + b}, \quad k = 0, 1, \dots$$

Since we need to perform the iteration at every time level, the time level counter is often also included:

$$au^{n,k} u^{n,k+1} + bu^{n,k+1} - u^{n-1} = 0 \quad \Rightarrow \quad u^{n,k+1} = \frac{u^{n-1}}{au^{n,k} + b}, \quad k = 0, 1, \dots,$$

with the start value $u^{n,0} = u^{n-1}$ and the final converged value $u^n = u^{n,k}$ for sufficiently large k .

However, we will normally apply a mathematical notation in our final formulas that is as close as possible to what we aim to write in a computer code and then it becomes natural to use u and u^- instead of u^{k+1} and u^k or $u^{n,k+1}$ and $u^{n,k}$.

Stopping criteria. The iteration method can typically be terminated when the change in the solution is smaller than a tolerance ϵ_u :

$$|u - u^-| \leq \epsilon_u,$$

or when the residual in the equation is sufficiently small ($< \epsilon_r$),

$$|F(u)| = |au^2 + bu + c| < \epsilon_r .$$

A single Picard iteration. Instead of iterating until a stopping criterion is fulfilled, one may iterate a specific number of times. Just one Picard iteration is popular as this corresponds to the intuitive idea of approximating a nonlinear term like $(u^n)^2$ by $u^{n-1}u^n$. This follows from the linearization u^-u^n and the initial choice of $u^- = u^{n-1}$ at time level t_n . In other words, a single Picard iteration corresponds to using the solution at the previous time level to linearize nonlinear terms. The resulting discretization becomes (using proper values for a , b , and c)

$$\frac{u^n - u^{n-1}}{\Delta t} = u^n(1 - u^{n-1}), \quad (10.5)$$

which is a linear algebraic equation in the unknown u^n , making it easy to solve for u^n without any need for any alternative notation.

We shall later refer to the strategy of taking one Picard step, or equivalently, linearizing terms with use of the solution at the previous time step, as the *Picard1* method. It is a widely used approach in science and technology, but with some limitations if Δt is not sufficiently small (as will be illustrated later).

Notice

Equation (10.5) does not correspond to a “pure” finite difference method where the equation is sampled at a point and derivatives replaced by differences (because the u^{n-1} term on the right-hand side must then be u^n). The best interpretation of the scheme (10.5) is a Backward Euler difference combined with a single (perhaps insufficient) Picard iteration at each time level, with the value at the previous time level as start for the Picard iteration.

10.1.7 Linearization by a geometric mean

We consider now a Crank-Nicolson discretization of (10.1). This means that the time derivative is approximated by a centered difference,

$$[D_t u = u(1 - u)]^{n+\frac{1}{2}},$$

written out as

$$\frac{u^{n+1} - u^n}{\Delta t} = u^{n+\frac{1}{2}} - (u^{n+\frac{1}{2}})^2. \quad (10.6)$$

The first term $u^{n+\frac{1}{2}}$ is normally approximated by an arithmetic mean,

$$u^{n+\frac{1}{2}} \approx \frac{1}{2}(u^n + u^{n+1}),$$

such that the scheme involves the unknown function only at the time levels where we actually compute it. The same arithmetic mean applied to the second term gives

$$(u^{n+\frac{1}{2}})^2 \approx \frac{1}{4}(u^n + u^{n+1})^2,$$

which is nonlinear in the unknown u^{n+1} . However, using a *geometric mean* for $(u^{n+\frac{1}{2}})^2$ is a way of linearizing the nonlinear term in (10.6):

$$(u^{n+\frac{1}{2}})^2 \approx u^n u^{n+1}.$$

Using an arithmetic mean on the linear $u^{n+\frac{1}{2}}$ term in (10.6) and a geometric mean for the second term, results in a linearized equation for the unknown u^{n+1} :

$$\frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{2}(u^n + u^{n+1}) - u^n u^{n+1},$$

which can readily be solved:

$$u^{n+1} = \frac{1 + \frac{1}{2}\Delta t}{1 + \Delta t u^n - \frac{1}{2}\Delta t} u^n.$$

This scheme can be coded directly, and since there is no nonlinear algebraic equation to iterate over, we skip the simplified notation with u for u^{n+1} and $u^{(1)}$ for u^n . The technique with using a geometric average is an example of transforming a nonlinear algebraic equation to a linear one, without any need for iterations.

The geometric mean approximation is often very effective for linearizing quadratic nonlinearities. Both the arithmetic and geometric mean approximations have truncation errors of order Δt^2 and are therefore compatible with the truncation error $\mathcal{O}(\Delta t^2)$ of the centered difference approximation for u' in the Crank-Nicolson method.

Applying the operator notation for the means and finite differences, the linearized Crank-Nicolson scheme for the logistic equation can be compactly expressed as

$$[D_t u = \bar{u}^t + \bar{u}^{2^{t,g}}]^{n+\frac{1}{2}}.$$

Remark

If we use an arithmetic instead of a geometric mean for the nonlinear term in (10.6), we end up with a nonlinear term $(u^{n+1})^2$. This term can be linearized as $u^- u^{n+1}$ in a Picard iteration approach and in particular as $u^n u^{n+1}$ in a Picard1 iteration approach. The latter gives a scheme almost identical to the one arising from a geometric mean (the difference in u^{n+1} being $\frac{1}{4}\Delta t u^n(u^{n+1} - u^n) \approx \frac{1}{4}\Delta t^2 u' u$, i.e., a difference of size Δt^2).

10.1.8 Newton's method

The Backward Euler scheme (10.2) for the logistic equation leads to a nonlinear algebraic equation (10.3). Now we write any nonlinear algebraic equation in the general and compact form

$$F(u) = 0.$$

Newton's method linearizes this equation by approximating $F(u)$ by its Taylor series expansion around a computed value u^- and keeping only the linear part:

$$\begin{aligned} F(u) &= F(u^-) + F'(u^-)(u - u^-) + \frac{1}{2}F''(u^-)(u - u^-)^2 + \dots \\ &\approx F(u^-) + F'(u^-)(u - u^-) = \hat{F}(u). \end{aligned}$$

The linear equation $\hat{F}(u) = 0$ has the solution

$$u = u^- - \frac{F(u^-)}{F'(u^-)}.$$

Expressed with an iteration index in the unknown, Newton's method takes on the more familiar mathematical form

$$u^{k+1} = u^k - \frac{F(u^k)}{F'(u^k)}, \quad k = 0, 1, \dots$$

When the method converges, it can be shown that the error in iteration $k + 1$ of Newton's method is proportional to the square of the error in iteration k , a result referred to as *quadratic convergence*. This means that for small errors the method converges very fast, and in particular much faster than Picard iteration and other iteration methods. (The proof of this result is found in most textbooks on numerical analysis.) However, the quadratic convergence appears only if u^k is sufficiently close to the solution. Further away from the solution the method can easily converge very slowly or diverge. The reader is encouraged to do Exercise 10.3 to get a better understanding for the behavior of the method.

Application of Newton's method to the logistic equation discretized by the Backward Euler method is straightforward as we have

$$F(u) = au^2 + bu + c, \quad a = \Delta t, \quad b = 1 - \Delta t, \quad c = -u^{(1)},$$

and then

$$F'(u) = 2au + b.$$

The iteration method becomes

$$u = u^- + \frac{a(u^-)^2 + bu^- + c}{2au^- + b}, \quad u^- \leftarrow u. \quad (10.7)$$

At each time level, we start the iteration by setting $u^- = u^{(1)}$. Stopping criteria as listed for the Picard iteration can be used also for Newton's method.

An alternative mathematical form, where we write out a , b , and c , and use a time level counter n and an iteration counter k , takes the form

$$u^{n,k+1} = u^{n,k} + \frac{\Delta t(u^{n,k})^2 + (1 - \Delta t)u^{n,k} - u^{n-1}}{2\Delta t u^{n,k} + 1 - \Delta t}, \quad u^{n,0} = u^{n-1}, \quad (10.8)$$

for $k = 0, 1, \dots$. A program implementation is much closer to (10.7) than to (10.8), but the latter is better aligned with the established mathematical notation used in the literature.

10.1.9 Relaxation

One iteration in Newton's method or Picard iteration consists of solving a linear problem $\hat{F}(u) = 0$. Sometimes convergence problems arise because the new solution u of $\hat{F}(u) = 0$ is “too far away” from the previously computed solution u^- . A remedy is to introduce a relaxation, meaning that we first solve $\hat{F}(u^*) = 0$ for a suggested value u^* and then we take u as a weighted mean of what we had, u^- , and what our linearized equation $\hat{F} = 0$ suggests, u^* :

$$u = \omega u^* + (1 - \omega)u^-.$$

The parameter ω is known as a *relaxation parameter*, and a choice $\omega < 1$ may prevent divergent iterations.

Relaxation in Newton's method can be directly incorporated in the basic iteration formula:

$$u = u^- - \omega \frac{F(u^-)}{F'(u^-)}. \quad (10.9)$$

10.1.10 Implementation and experiments

The program `logistic.py` contains implementations of all the methods described above. Below is an extract of the file showing how the Picard and Newton methods are implemented for a Backward Euler discretization of the logistic equation.

```
def BE_logistic(u0, dt, Nt, choice='Picard',
                eps_r=1E-3, omega=1, max_iter=1000):
    if choice == 'Picard1':
        choice = 'Picard'
        max_iter = 1

    u = np.zeros(Nt+1)
    iterations = []
    u[0] = u0
    for n in range(1, Nt+1):
        a = dt
        b = 1 - dt
        c = -u[n-1]

        if choice == 'Picard':

            def F(u):
                return a*u**2 + b*u + c

            u_ = u[n-1]
            k = 0
            while abs(F(u_)) > eps_r and k < max_iter:
                u_ = omega*(-c/(a*u_ + b)) + (1-omega)*u_
                k += 1
            u[n] = u_
            iterations.append(k)

        elif choice == 'Newton':

            def F(u):
                return a*u**2 + b*u + c

            def dF(u):
                return 2*a*u + b

            u_ = u[n-1]
            k = 0
            while abs(F(u_)) > eps_r and k < max_iter:
                u_ = u_ - F(u_)/dF(u_)
                k += 1
            u[n] = u_
            iterations.append(k)

    return u, iterations
```

The Crank-Nicolson method utilizing a linearization based on the geometric mean gives a simpler algorithm:

```
def CN_logistic(u0, dt, Nt):
    u = np.zeros(Nt+1)
    u[0] = u0
    for n in range(0, Nt):
        u[n+1] = (1 + 0.5*dt)/(1 + dt*u[n] - 0.5*dt)*u[n]
    return u
```

We may run experiments with the model problem (10.1) and the different strategies for dealing with nonlinearities as described above. For a quite coarse time resolution, $\Delta t = 0.9$, use of a tolerance $\epsilon_r = 0.05$ in the stopping criterion introduces an iteration error, especially in the Picard iterations, that is visibly much larger than the time discretization error due to a large Δt . This is illustrated by comparing the upper two plots in Figure 10.1. The one to the right has a stricter tolerance $\epsilon = 10^{-3}$, which leads to all the curves involving backward Euler, using iterative solution by either Picard or Newton iterations, to be on top of each other (and no changes can be visually observed by reducing ϵ_r further). The reason why Newton's method does much better than Picard iteration in the upper left plot is that Newton's method with one step comes far below the ϵ_r tolerance, while the Picard iteration needs on average 7 iterations to bring the residual down to $\epsilon_r = 10^{-1}$, which gives insufficient accuracy in the solution of the nonlinear equation. It is obvious that the Picard1 method gives significant errors in addition to the time discretization unless the time step is as small as in the lower right plot.

The *BE exact* curve corresponds to using the exact solution of the quadratic equation at each time level, so this curve is only affected by the Backward Euler time discretization. The *CN gm* curve corresponds to the theoretically more accurate Crank-Nicolson discretization, combined with a geometric mean for linearization. Visually, this curve appears more accurate in all the plots, especially if we take the plot in the lower right with a small Δt and an appropriately small ϵ_r value as the reference curve.

When it comes to the need for iterations, Figure 10.2 displays the number of iterations required at each time level for Newton's method and Picard iteration. The smaller Δt is, the better starting value we have for the iteration, and the faster the convergence is. With $\Delta t = 0.9$ Picard iteration requires on average 32 iterations per time step for the stricter convergence criterion, but this number is dramatically reduced as Δt is reduced.

However, introducing relaxation and a parameter $\omega = 0.8$ immediately reduces the average of 32 to 7, indicating that for the large $\Delta t = 0.9$,

Picard iteration takes too long steps. An approximately optimal value for ω in this case is 0.5, which results in an average of only 2 iterations! An even more dramatic impact of ω appears when $\Delta t = 1$: Picard iteration does not converge in 1000 iterations, but $\omega = 0.5$ again brings the average number of iterations down to 2.

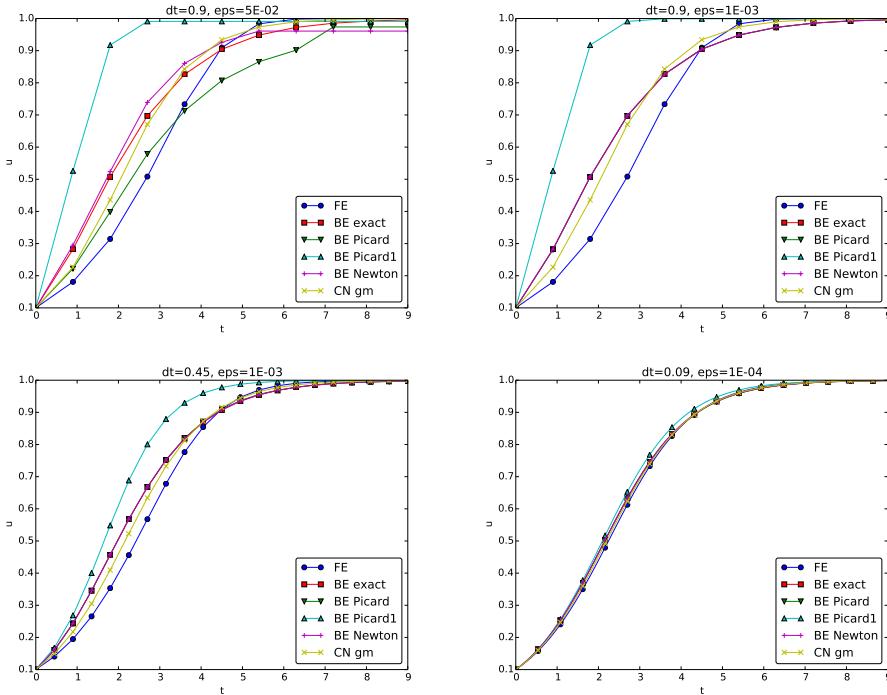


Fig. 10.1 Impact of solution strategy and time step length on the solution.

10.1.11 Generalization to a general nonlinear ODE

Let us see how the various methods in the previous sections can be applied to the more generic model

$$u' = f(u, t), \quad (10.10)$$

where f is a nonlinear function of u .

Explicit time discretization. Explicit ODE methods like the Forward Euler scheme, various Runge-Kutta methods, Adams-Basforth methods

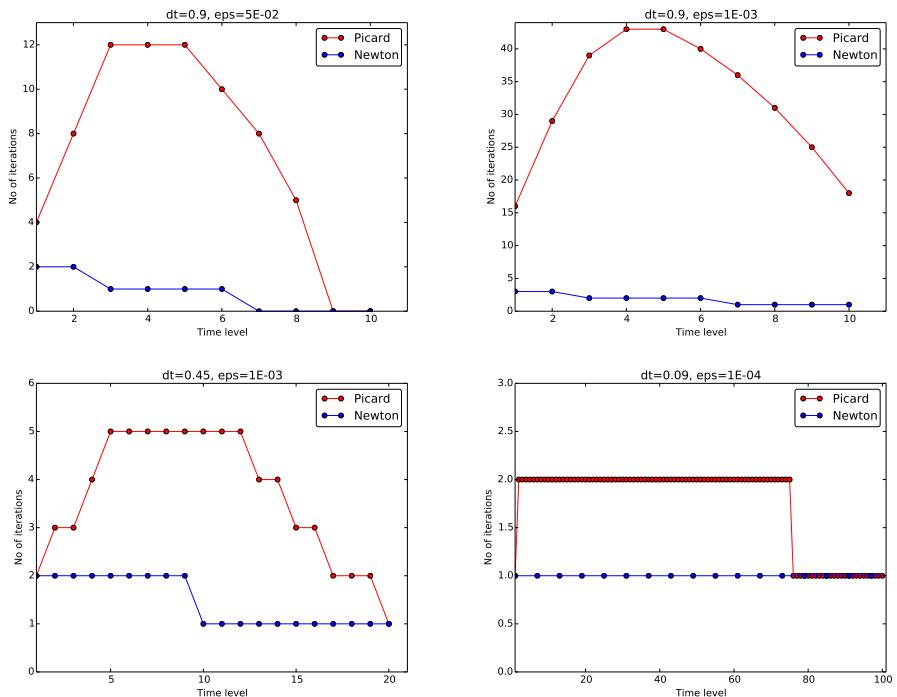


Fig. 10.2 Comparison of the number of iterations at various time levels for Picard and Newton iteration.

all evaluate f at time levels where u is already computed, so nonlinearities in f do not pose any difficulties.

Backward Euler discretization. Approximating u' by a backward difference leads to a Backward Euler scheme, which can be written as

$$F(u^n) = u^n - \Delta t f(u^n, t_n) - u^{n-1} = 0,$$

or alternatively

$$F(u) = u - \Delta t f(u, t_n) - u^{(1)} = 0.$$

A simple Picard iteration, not knowing anything about the nonlinear structure of f , must approximate $f(u, t_n)$ by $f(u^-, t_n)$:

$$\hat{F}(u) = u - \Delta t f(u^-, t_n) - u^{(1)}.$$

The iteration starts with $u^- = u^{(1)}$ and proceeds with repeating

$$u^* = \Delta t f(u^-, t_n) + u^{(1)}, \quad u = \omega u^* + (1 - \omega)u^-, \quad u^- \leftarrow u,$$

until a stopping criterion is fulfilled.

Explicit vs implicit treatment of nonlinear terms

Evaluating f for a known u^- is referred to as *explicit* treatment of f , while if $f(u, t)$ has some structure, say $f(u, t) = u^3$, parts of f can involve the unknown u , as in the manual linearization like $(u^-)^2u$, and then the treatment of f is “more implicit” and “less explicit”. This terminology is inspired by time discretization of $u' = f(u, t)$, where evaluating f for known u values gives explicit formulas for the unknown and hence explicit schemes, while treating f or parts of f implicitly, meaning that equations must be solved in terms of the unknown, makes f contribute to the unknown terms in the equation at the new time level.

Explicit treatment of f usually means stricter conditions on Δt to achieve stability of time discretization schemes. The same applies to iteration techniques for nonlinear algebraic equations: the “less” we linearize f (i.e., the more we keep of u in the original formula), the faster the convergence may be.

We may say that $f(u, t) = u^3$ is treated explicitly if we evaluate f as $(u^-)^3$, semi or partially implicit if we linearize as $(u^-)^2u$ and fully implicit if we represent f by u^3 . (Of course, the fully implicit representation will require further linearization, but with $f(u, t) = u^2$ a fully implicit treatment is possible if the resulting quadratic equation is solved with a formula.)

For the ODE $u' = -u^3$ with $f(u, t) = -u^3$ and coarse time resolution $\Delta t = 0.4$, Picard iteration with $(u^-)^2u$ requires 8 iterations with $\epsilon_r = 10^{-3}$ for the first time step, while $(u^-)^3$ leads to 22 iterations. After about 10 time steps both approaches are down to about 2 iterations per time step, but this example shows a potential of treating f more implicitly.

A trick to treat f implicitly in Picard iteration is to evaluate it as $f(u^-, t)u/u^-$. For a polynomial f , $f(u, t) = u^m$, this corresponds to $(u^-)^m u/u^- = (u^-)^{m-1}u$. Sometimes this more implicit treatment has no effect, as with $f(u, t) = \exp(-u)$ and $f(u, t) = \ln(1 + u)$, but with $f(u, t) = \sin(2(u + 1))$, the $f(u^-, t)u/u^-$ trick leads to 7, 9, and 11 iterations during the first three steps, while $f(u^-, t)$

demands 17, 21, and 20 iterations. (Experiments can be done with the code `ODE_Picard_tricks.py`.)

Newton's method applied to a Backward Euler discretization of $u' = f(u, t)$ requires the computation of the derivative

$$F'(u) = 1 - \Delta t \frac{\partial f}{\partial u}(u, t_n).$$

Starting with the solution at the previous time level, $u^- = u^{(1)}$, we can just use the standard formula

$$u = u^- - \omega \frac{F(u^-)}{F'(u^-)} = u^- - \omega \frac{u^- - \Delta t f(u^-, t_n) - u^{(1)}}{1 - \Delta t \frac{\partial}{\partial u} f(u^-, t_n)}. \quad (10.11)$$

Crank-Nicolson discretization. The standard Crank-Nicolson scheme with arithmetic mean approximation of f takes the form

$$\frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{2}(f(u^{n+1}, t_{n+1}) + f(u^n, t_n)).$$

We can write the scheme as a nonlinear algebraic equation

$$F(u) = u - u^{(1)} - \Delta t \frac{1}{2} f(u, t_{n+1}) - \Delta t \frac{1}{2} f(u^{(1)}, t_n) = 0. \quad (10.12)$$

A Picard iteration scheme must in general employ the linearization

$$\hat{F}(u) = u - u^{(1)} - \Delta t \frac{1}{2} f(u^-, t_{n+1}) - \Delta t \frac{1}{2} f(u^{(1)}, t_n),$$

while Newton's method can apply the general formula (10.11) with $F(u)$ given in (10.12) and

$$F'(u) = 1 - \frac{1}{2} \Delta t \frac{\partial f}{\partial u}(u, t_{n+1}).$$

10.1.12 Systems of ODEs

We may write a system of ODEs

$$\begin{aligned}\frac{d}{dt}u_0(t) &= f_0(u_0(t), u_1(t), \dots, u_N(t), t), \\ \frac{d}{dt}u_1(t) &= f_1(u_0(t), u_1(t), \dots, u_N(t), t), \\ &\vdots \\ \frac{d}{dt}u_m(t) &= f_m(u_0(t), u_1(t), \dots, u_N(t), t),\end{aligned}$$

as

$$u' = f(u, t), \quad u(0) = U_0, \quad (10.13)$$

if we interpret u as a vector $u = (u_0(t), u_1(t), \dots, u_N(t))$ and f as a vector function with components $(f_0(u, t), f_1(u, t), \dots, f_N(u, t))$.

Most solution methods for scalar ODEs, including the Forward and Backward Euler schemes and the Crank-Nicolson method, generalize in a straightforward way to systems of ODEs simply by using vector arithmetics instead of scalar arithmetics, which corresponds to applying the scalar scheme to each component of the system. For example, here is a backward difference scheme applied to each component,

$$\begin{aligned}\frac{u_0^n - u_0^{n-1}}{\Delta t} &= f_0(u^n, t_n), \\ \frac{u_1^n - u_1^{n-1}}{\Delta t} &= f_1(u^n, t_n), \\ &\vdots \\ \frac{u_N^n - u_N^{n-1}}{\Delta t} &= f_N(u^n, t_n),\end{aligned}$$

which can be written more compactly in vector form as

$$\frac{u^n - u^{n-1}}{\Delta t} = f(u^n, t_n).$$

This is a *system of algebraic equations*,

$$u^n - \Delta t f(u^n, t_n) - u^{n-1} = 0,$$

or written out

$$u_0^n - \Delta t f_0(u^n, t_n) - u_0^{n-1} = 0,$$

⋮

$$u_N^n - \Delta t f_N(u^n, t_n) - u_N^{n-1} = 0.$$

Example. We shall address the 2×2 ODE system for oscillations of a pendulum subject to gravity and air drag. The system can be written as

$$\dot{\omega} = -\sin \theta - \beta \omega |\omega|, \quad (10.14)$$

$$\dot{\theta} = \omega, \quad (10.15)$$

where β is a dimensionless parameter (this is the scaled, dimensionless version of the original, physical model). The unknown components of the system are the angle $\theta(t)$ and the angular velocity $\omega(t)$. We introduce $u_0 = \omega$ and $u_1 = \theta$, which leads to

$$\begin{aligned} u'_0 &= f_0(u, t) = -\sin u_1 - \beta u_0 |u_0|, \\ u'_1 &= f_1(u, t) = u_0. \end{aligned}$$

A Crank-Nicolson scheme reads

$$\begin{aligned} \frac{u_0^{n+1} - u_0^n}{\Delta t} &= -\sin u_1^{n+\frac{1}{2}} - \beta u_0^{n+\frac{1}{2}} |u_0^{n+\frac{1}{2}}| \\ &\approx -\sin \left(\frac{1}{2}(u_1^{n+1} + u_1^n) \right) - \beta \frac{1}{4}(u_0^{n+1} + u_0^n) |u_0^{n+1} + u_0^n|, \end{aligned} \quad (10.16)$$

$$\frac{u_1^{n+1} - u_1^n}{\Delta t} = u_0^{n+\frac{1}{2}} \approx \frac{1}{2}(u_0^{n+1} + u_0^n). \quad (10.17)$$

This is a *coupled system* of two nonlinear algebraic equations in two unknowns u_0^{n+1} and u_1^{n+1} .

Using the notation u_0 and u_1 for the unknowns u_0^{n+1} and u_1^{n+1} in this system, writing $u_0^{(1)}$ and $u_1^{(1)}$ for the previous values u_0^n and u_1^n , multiplying by Δt and moving the terms to the left-hand sides, gives

$$u_0 - u_0^{(1)} + \Delta t \sin \left(\frac{1}{2}(u_1 + u_1^{(1)}) \right) + \frac{1}{4} \Delta t \beta(u_0 + u_0^{(1)}) |u_0 + u_0^{(1)}| = 0, \quad (10.18)$$

$$u_1 - u_1^{(1)} - \frac{1}{2} \Delta t (u_0 + u_0^{(1)}) = 0. \quad (10.19)$$

Obviously, we have a need for solving systems of nonlinear algebraic equations, which is the topic of the next section.

10.2 Systems of nonlinear algebraic equations

Implicit time discretization methods for a system of ODEs, or a PDE, lead to *systems* of nonlinear algebraic equations, written compactly as

$$F(u) = 0,$$

where u is a vector of unknowns $u = (u_0, \dots, u_N)$, and F is a vector function: $F = (F_0, \dots, F_N)$. The system at the end of Section 10.1.12 fits this notation with $N = 2$, $F_0(u)$ given by the left-hand side of (10.18), while $F_1(u)$ is the left-hand side of (10.19).

Sometimes the equation system has a special structure because of the underlying problem, e.g.,

$$A(u)u = b(u),$$

with $A(u)$ as an $(N+1) \times (N+1)$ matrix function of u and b as a vector function: $b = (b_0, \dots, b_N)$.

We shall next explain how Picard iteration and Newton's method can be applied to systems like $F(u) = 0$ and $A(u)u = b(u)$. The exposition has a focus on ideas and practical computations. More theoretical considerations, including quite general results on convergence properties of these methods, can be found in Kelley [14].

10.2.1 Picard iteration

We cannot apply Picard iteration to nonlinear equations unless there is some special structure. For the commonly arising case $A(u)u = b(u)$ we can linearize the product $A(u)u$ to $A(u^-)u$ and $b(u)$ as $b(u^-)$. That is,

we use the most previously computed approximation in A and b to arrive at a *linear system* for u :

$$A(u^-)u = b(u^-).$$

A relaxed iteration takes the form

$$A(u^-)u^* = b(u^-), \quad u = \omega u^* + (1 - \omega)u^-.$$

In other words, we solve a system of nonlinear algebraic equations as a sequence of linear systems.

Algorithm for relaxed Picard iteration

Given $A(u)u = b(u)$ and an initial guess u^- , iterate until convergence:

1. solve $A(u^-)u^* = b(u^-)$ with respect to u^*
2. $u = \omega u^* + (1 - \omega)u^-$
3. $u^- \leftarrow u$

“Until convergence” means that the iteration is stopped when the change in the unknown, $\|u - u^-\|$, or the residual $\|A(u)u - b\|$, is sufficiently small, see Section 10.2.3 for more details.

10.2.2 Newton’s method

The natural starting point for Newton’s method is the general nonlinear vector equation $F(u) = 0$. As for a scalar equation, the idea is to approximate F around a known value u^- by a linear function \hat{F} , calculated from the first two terms of a Taylor expansion of F . In the multi-variate case these two terms become

$$F(u^-) + J(u^-) \cdot (u - u^-),$$

where J is the *Jacobian* of F , defined by

$$J_{i,j} = \frac{\partial F_i}{\partial u_j}.$$

So, the original nonlinear system is approximated by

$$\hat{F}(u) = F(u^-) + J(u^-) \cdot (u - u^-) = 0,$$

which is linear in u and can be solved in a two-step procedure: first solve $J\delta u = -F(u^-)$ with respect to the vector δu and then update $u = u^- + \delta u$. A relaxation parameter can easily be incorporated:

$$u = \omega(u^- + \delta u) + (1 - \omega)u^- = u^- + \omega\delta u.$$

Algorithm for Newton's method

Given $F(u) = 0$ and an initial guess u^- , iterate until convergence:

1. solve $J\delta u = -F(u^-)$ with respect to δu
2. $u = u^- + \omega\delta u$
3. $u^- \leftarrow u$

For the special system with structure $A(u)u = b(u)$,

$$F_i = \sum_k A_{i,k}(u)u_k - b_i(u),$$

one gets

$$J_{i,j} = \sum_k \frac{\partial A_{i,k}}{\partial u_j} u_k + A_{i,j} - \frac{\partial b_i}{\partial u_j}. \quad (10.20)$$

We realize that the Jacobian needed in Newton's method consists of $A(u^-)$ as in the Picard iteration plus two additional terms arising from the differentiation. Using the notation $A'(u)$ for $\partial A / \partial u$ (a quantity with three indices: $\partial A_{i,k} / \partial u_j$), and $b'(u)$ for $\partial b / \partial u$ (a quantity with two indices: $\partial b_i / \partial u_j$), we can write the linear system to be solved as

$$(A + A'u + b')\delta u = -Au + b,$$

or

$$(A(u^-) + A'(u^-)u^- + b'(u^-))\delta u = -A(u^-)u^- + b(u^-).$$

Rearranging the terms demonstrates the difference from the system solved in each Picard iteration:

$$\underbrace{A(u^-)(u^- + \delta u) - b(u^-)}_{\text{Picard system}} + \gamma(A'(u^-)u^- + b'(u^-))\delta u = 0.$$

Here we have inserted a parameter γ such that $\gamma = 0$ gives the Picard system and $\gamma = 1$ gives the Newton system. Such a parameter can be handy in software to easily switch between the methods.

Combined algorithm for Picard and Newton iteration

Given $A(u)$, $b(u)$, and an initial guess u^- , iterate until convergence:

1. solve $(A + \gamma(A'(u^-)u^- + b'(u^-)))\delta u = -A(u^-)u^- + b(u^-)$ with respect to δu
2. $u = u^- + \omega\delta u$
3. $u^- \leftarrow u$

$\gamma = 1$ gives a Newton method while $\gamma = 0$ corresponds to Picard iteration.

10.2.3 Stopping criteria

Let $\|\cdot\|$ be the standard Euclidean vector norm. Four termination criteria are much in use:

- Absolute change in solution: $\|u - u^-\| \leq \epsilon_u$
- Relative change in solution: $\|u - u^-\| \leq \epsilon_u \|u_0\|$, where u_0 denotes the start value of u^- in the iteration
- Absolute residual: $\|F(u)\| \leq \epsilon_r$
- Relative residual: $\|F(u)\| \leq \epsilon_r \|F(u_0)\|$

To prevent divergent iterations to run forever, one terminates the iterations when the current number of iterations k exceeds a maximum value k_{\max} .

For stationary problems, the relative criteria are most used since they are not sensitive to the characteristic size of u , which may depend on the underlying mesh and its resolution. Nevertheless, the relative criteria can be misleading when the initial start value for the iteration is very close to the solution, since an unnecessary reduction in the error measure is enforced. For time-dependent problems, if the time-step is small then

the previous solution may be a quite good guess for the unknown and in such cases the absolute criteria works better. It is common to combine the absolute and relative measures of the size of the residual, as in

$$\|F(u)\| \leq \epsilon_{rr}\|F(u_0)\| + \epsilon_{ra}, \quad (10.21)$$

where ϵ_{rr} is the tolerance in the relative criterion and ϵ_{ra} is the tolerance in the absolute criterion. With a very good initial guess for the iteration (typically the solution of a differential equation at the previous time level), the term $\|F(u_0)\|$ is small and ϵ_{ra} is the dominating tolerance. Otherwise, $\epsilon_{rr}\|F(u_0)\|$ and the relative criterion dominates.

With the change in solution as criterion we can formulate a combined absolute and relative measure of the change in the solution:

$$\|\delta u\| \leq \epsilon_{ur}\|u_0\| + \epsilon_{ua}, \quad (10.22)$$

The ultimate termination criterion, combining the residual and the change in solution with a test on the maximum number of iterations, can be expressed as

$$\|F(u)\| \leq \epsilon_{rr}\|F(u_0)\| + \epsilon_{ra} \quad \text{or} \quad \|\delta u\| \leq \epsilon_{ur}\|u_0\| + \epsilon_{ua} \quad \text{or} \quad k > k_{\max}. \quad (10.23)$$

10.2.4 Example: A nonlinear ODE model from epidemiology

The simplest model spreading of a disease, such as a flu, takes the form of a 2×2 ODE system

$$S' = -\beta SI, \quad (10.24)$$

$$I' = \beta SI - \nu I, \quad (10.25)$$

where $S(t)$ is the number of people who can get ill (susceptibles) and $I(t)$ is the number of people who are ill (infected). The constants $\beta > 0$ and $\nu > 0$ must be given along with initial conditions $S(0)$ and $I(0)$.

Implicit time discretization. A Crank-Nicolson scheme leads to a 2×2 system of nonlinear algebraic equations in the unknowns S^{n+1} and I^{n+1} :

$$\frac{S^{n+1} - S^n}{\Delta t} = -\beta[SI]^{n+\frac{1}{2}} \approx -\frac{\beta}{2}(S^n I^n + S^{n+1} I^{n+1}), \quad (10.26)$$

$$\frac{I^{n+1} - I^n}{\Delta t} = \beta[SI]^{n+\frac{1}{2}} - \nu I^{n+\frac{1}{2}} \approx \frac{\beta}{2}(S^n I^n + S^{n+1} I^{n+1}) - \frac{\nu}{2}(I^n + I^{n+1}). \quad (10.27)$$

Introducing S for S^{n+1} , $S^{(1)}$ for S^n , I for I^{n+1} , $I^{(1)}$ for I^n , we can rewrite the system as

$$F_S(S, I) = S - S^{(1)} + \frac{1}{2}\Delta t\beta(S^{(1)}I^{(1)} + SI) = 0, \quad (10.28)$$

$$F_I(S, I) = I - I^{(1)} - \frac{1}{2}\Delta t\beta(S^{(1)}I^{(1)} + SI) + \frac{1}{2}\Delta t\nu(I^{(1)} + I) = 0. \quad (10.29)$$

A Picard iteration. We assume that we have approximations S^- and I^- to S and I , respectively. A way of linearizing the only nonlinear term SI is to write I^-S in the $F_S = 0$ equation and S^-I in the $F_I = 0$ equation, which also *decouples* the equations. Solving the resulting linear equations with respect to the unknowns S and I gives

$$S = \frac{S^{(1)} - \frac{1}{2}\Delta t\beta S^{(1)}I^{(1)}}{1 + \frac{1}{2}\Delta t\beta I^-},$$

$$I = \frac{I^{(1)} + \frac{1}{2}\Delta t\beta S^{(1)}I^{(1)} - \frac{1}{2}\Delta t\nu I^{(1)}}{1 - \frac{1}{2}\Delta t\beta S^- + \frac{1}{2}\Delta t\nu}.$$

Before a new iteration, we must update $S^- \leftarrow S$ and $I^- \leftarrow I$.

Newton's method. The nonlinear system (10.28)-(10.29) can be written as $F(u) = 0$ with $F = (F_S, F_I)$ and $u = (S, I)$. The Jacobian becomes

$$J = \begin{pmatrix} \frac{\partial}{\partial S} F_S & \frac{\partial}{\partial I} F_S \\ \frac{\partial}{\partial S} F_I & \frac{\partial}{\partial I} F_I \end{pmatrix} = \begin{pmatrix} 1 + \frac{1}{2}\Delta t\beta I & \frac{1}{2}\Delta t\beta S \\ -\frac{1}{2}\Delta t\beta I & 1 - \frac{1}{2}\Delta t\beta S + \frac{1}{2}\Delta t\nu \end{pmatrix}.$$

The Newton system $J(u^-)\delta u = -F(u^-)$ to be solved in each iteration is then

$$\begin{pmatrix} 1 + \frac{1}{2}\Delta t\beta I^- & \frac{1}{2}\Delta t\beta S^- \\ -\frac{1}{2}\Delta t\beta I^- & 1 - \frac{1}{2}\Delta t\beta S^- + \frac{1}{2}\Delta t\nu \end{pmatrix} \begin{pmatrix} \delta S \\ \delta I \end{pmatrix} = \begin{pmatrix} S^- - S^{(1)} + \frac{1}{2}\Delta t\beta(S^{(1)}I^{(1)} + S^-I^-) \\ I^- - I^{(1)} - \frac{1}{2}\Delta t\beta(S^{(1)}I^{(1)} + S^-I^-) + \frac{1}{2}\Delta t\nu(I^{(1)} + I^-) \end{pmatrix}$$

Remark. For this particular system of ODEs, explicit time integration methods work very well. Even a Forward Euler scheme is fine, but (as also experienced more generally) the 4-th order Runge-Kutta method is an excellent balance between high accuracy, high efficiency, and simplicity.

10.3 Linearization at the differential equation level

The attention is now turned to nonlinear partial differential equations (PDEs) and application of the techniques explained above for ODEs. The model problem is a nonlinear diffusion equation for $u(\mathbf{x}, t)$:

$$\frac{\partial u}{\partial t} = \nabla \cdot (\alpha(u)\nabla u) + f(u), \quad \mathbf{x} \in \Omega, \quad t \in (0, T], \quad (10.30)$$

$$-\alpha(u)\frac{\partial u}{\partial n} = g, \quad \mathbf{x} \in \partial\Omega_N, \quad t \in (0, T], \quad (10.31)$$

$$u = u_0, \quad \mathbf{x} \in \partial\Omega_D, \quad t \in (0, T]. \quad (10.32)$$

In the present section, our aim is to discretize this problem in time and then present techniques for linearizing the time-discrete PDE problem “at the PDE level” such that we transform the nonlinear stationary PDE problem at each time level into a sequence of linear PDE problems, which can be solved using any method for linear PDEs. This strategy avoids the solution of systems of nonlinear algebraic equations. In Section 10.4 we shall take the opposite (and more common) approach: discretize the nonlinear problem in time and space first, and then solve the resulting nonlinear algebraic equations at each time level by the methods of Section 10.2. Very often, the two approaches are mathematically identical, so there is no preference from a computational efficiency point of view. The details of the ideas sketched above will hopefully become clear through the forthcoming examples.

10.3.1 Explicit time integration

The nonlinearities in the PDE are trivial to deal with if we choose an explicit time integration method for (10.30), such as the Forward Euler method:

$$[D_t^+ u = \nabla \cdot (\alpha(u) \nabla u) + f(u)]^n,$$

or written out,

$$\frac{u^{n+1} - u^n}{\Delta t} = \nabla \cdot (\alpha(u^n) \nabla u^n) + f(u^n),$$

which is a linear equation in the unknown u^{n+1} with solution

$$u^{n+1} = u^n + \Delta t \nabla \cdot (\alpha(u^n) \nabla u^n) + \Delta t f(u^n).$$

The disadvantage with this discretization is the strict stability criterion, e.g., $\Delta t \leq h^2/(6 \max \alpha)$ for the case $f = 0$ and a standard 2nd-order finite difference discretization in 3D space with mesh cell sizes $h = \Delta x = \Delta y = \Delta z$.

10.3.2 Backward Euler scheme and Picard iteration

A Backward Euler scheme for (10.30) reads

$$[D_t^- u = \nabla \cdot (\alpha(u) \nabla u) + f(u)]^n.$$

Written out,

$$\frac{u^n - u^{n-1}}{\Delta t} = \nabla \cdot (\alpha(u^n) \nabla u^n) + f(u^n). \quad (10.33)$$

This is a nonlinear PDE for the unknown function $u^n(\mathbf{x})$. Such a PDE can be viewed as a time-independent PDE where $u^{n-1}(\mathbf{x})$ is a known function.

We introduce a Picard iteration with k as iteration counter. A typical linearization of the $\nabla \cdot (\alpha(u^n) \nabla u^n)$ term in iteration $k + 1$ is to use the previously computed $u^{n,k}$ approximation in the diffusion coefficient: $\alpha(u^{n,k})$. The nonlinear source term is treated similarly: $f(u^{n,k})$. The unknown function $u^{n,k+1}$ then fulfills the linear PDE

$$\frac{u^{n,k+1} - u^{n-1}}{\Delta t} = \nabla \cdot (\alpha(u^{n,k}) \nabla u^{n,k+1}) + f(u^{n,k}). \quad (10.34)$$

The initial guess for the Picard iteration at this time level can be taken as the solution at the previous time level: $u^{n,0} = u^{n-1}$.

We can alternatively apply the implementation-friendly notation where u corresponds to the unknown we want to solve for, i.e., $u^{n,k+1}$ above, and u^- is the most recently computed value, $u^{n,k}$ above. Moreover, $u^{(1)}$ denotes the unknown function at the previous time level, u^{n-1} above. The PDE to be solved in a Picard iteration then looks like

$$\frac{u - u^{(1)}}{\Delta t} = \nabla \cdot (\alpha(u^-) \nabla u) + f(u^-). \quad (10.35)$$

At the beginning of the iteration we start with the value from the previous time level: $u^- = u^{(1)}$, and after each iteration, u^- is updated to u .

Remark on notation

The previous derivations of the numerical scheme for time discretizations of PDEs have, strictly speaking, a somewhat sloppy notation, but it is much used and convenient to read. A more precise notation must distinguish clearly between the exact solution of the PDE problem, here denoted $u_e(\mathbf{x}, t)$, and the exact solution of the spatial problem, arising after time discretization at each time level, where (10.33) is an example. The latter is here represented as $u^n(\mathbf{x})$ and is an approximation to $u_e(\mathbf{x}, t_n)$. Then we have another approximation $u^{n,k}(\mathbf{x})$ to $u^n(\mathbf{x})$ when solving the nonlinear PDE problem for u^n by iteration methods, as in (10.34).

In our notation, u is a synonym for $u^{n,k+1}$ and $u^{(1)}$ is a synonym for u^{n-1} , inspired by what are natural variable names in a code. We will usually state the PDE problem in terms of u and quickly redefine the symbol u to mean the numerical approximation, while u_e is not explicitly introduced unless we need to talk about the exact solution and the approximate solution at the same time.

10.3.3 Backward Euler scheme and Newton's method

At time level n , we have to solve the stationary PDE (10.33). In the previous section, we saw how this can be done with Picard iterations. Another alternative is to apply the idea of Newton's method in a clever way. Normally, Newton's method is defined for systems of *algebraic*

equations, but the idea of the method can be applied at the PDE level too.

Linearization via Taylor expansions. Let $u^{n,k}$ be an approximation to the unknown u^n . We seek a better approximation on the form

$$u^n = u^{n,k} + \delta u. \quad (10.36)$$

The idea is to insert (10.36) in (10.33), Taylor expand the nonlinearities and keep only the terms that are linear in δu (which makes (10.36) an approximation for u^n). Then we can solve a linear PDE for the correction δu and use (10.36) to find a new approximation

$$u^{n,k+1} = u^{n,k} + \delta u$$

to u^n . Repeating this procedure gives a sequence $u^{n,k+1}$, $k = 0, 1, \dots$ that hopefully converges to the goal u^n .

Let us carry out all the mathematical details for the nonlinear diffusion PDE discretized by the Backward Euler method. Inserting (10.36) in (10.33) gives

$$\frac{u^{n,k} + \delta u - u^{n-1}}{\Delta t} = \nabla \cdot (\alpha(u^{n,k} + \delta u) \nabla(u^{n,k} + \delta u)) + f(u^{n,k} + \delta u). \quad (10.37)$$

We can Taylor expand $\alpha(u^{n,k} + \delta u)$ and $f(u^{n,k} + \delta u)$:

$$\begin{aligned} \alpha(u^{n,k} + \delta u) &= \alpha(u^{n,k}) + \frac{d\alpha}{du}(u^{n,k})\delta u + \mathcal{O}(\delta u^2) \approx \alpha(u^{n,k}) + \alpha'(u^{n,k})\delta u, \\ f(u^{n,k} + \delta u) &= f(u^{n,k}) + \frac{df}{du}(u^{n,k})\delta u + \mathcal{O}(\delta u^2) \approx f(u^{n,k}) + f'(u^{n,k})\delta u. \end{aligned}$$

Inserting the linear approximations of α and f in (10.37) results in

$$\begin{aligned} \frac{u^{n,k} + \delta u - u^{n-1}}{\Delta t} &= \nabla \cdot (\alpha(u^{n,k}) \nabla u^{n,k}) + f(u^{n,k}) + \\ &\quad \nabla \cdot (\alpha(u^{n,k}) \nabla \delta u) + \nabla \cdot (\alpha'(u^{n,k}) \delta u \nabla u^{n,k}) + \\ &\quad \nabla \cdot (\alpha'(u^{n,k}) \delta u \nabla \delta u) + f'(u^{n,k}) \delta u. \quad (10.38) \end{aligned}$$

The term $\alpha'(u^{n,k}) \delta u \nabla \delta u$ is of order δu^2 and is therefore omitted since we expect the correction δu to be small ($\delta u \gg \delta u^2$). Reorganizing the equation gives a PDE for δu that we can write in short form as

$$\delta F(\delta u; u^{n,k}) = -F(u^{n,k}),$$

where

$$F(u^{n,k}) = \frac{u^{n,k} - u^{n-1}}{\Delta t} - \nabla \cdot (\alpha(u^{n,k}) \nabla u^{n,k}) + f(u^{n,k}), \quad (10.39)$$

$$\begin{aligned} \delta F(\delta u; u^{n,k}) &= -\frac{1}{\Delta t} \delta u + \nabla \cdot (\alpha(u^{n,k}) \nabla \delta u) + \\ &\quad \nabla \cdot (\alpha'(u^{n,k}) \delta u \nabla u^{n,k}) + f'(u^{n,k}) \delta u. \end{aligned} \quad (10.40)$$

Note that δF is a linear function of δu , and F contains only terms that are known, such that the PDE for δu is indeed linear.

Observations

The notational form $\delta F = -F$ resembles the Newton system $J\delta u = -F$ for systems of algebraic equations, with δF as $J\delta u$. The unknown vector in a linear system of algebraic equations enters the system as a linear operator in terms of a matrix-vector product ($J\delta u$), while at the PDE level we have a linear differential operator instead (δF).

Similarity with Picard iteration. We can rewrite the PDE for δu in a slightly different way too if we define $u^{n,k} + \delta u$ as $u^{n,k+1}$.

$$\begin{aligned} \frac{u^{n,k+1} - u^{n-1}}{\Delta t} &= \nabla \cdot (\alpha(u^{n,k}) \nabla u^{n,k+1}) + f(u^{n,k}) \\ &\quad + \nabla \cdot (\alpha'(u^{n,k}) \delta u \nabla u^{n,k}) + f'(u^{n,k}) \delta u. \end{aligned} \quad (10.41)$$

Note that the first line is the same PDE as arises in the Picard iteration, while the remaining terms arise from the differentiations that are an inherent ingredient in Newton's method.

Implementation. For coding we want to introduce u for u^n , u^- for $u^{n,k}$ and $u^{(1)}$ for u^{n-1} . The formulas for F and δF are then more clearly written as

$$F(u^-) = \frac{u^- - u^{(1)}}{\Delta t} - \nabla \cdot (\alpha(u^-) \nabla u^-) + f(u^-), \quad (10.42)$$

$$\begin{aligned} \delta F(\delta u; u^-) = & -\frac{1}{\Delta t} \delta u + \nabla \cdot (\alpha(u^-) \nabla \delta u) + \\ & \nabla \cdot (\alpha'(u^-) \delta u \nabla u^-) + f'(u^-) \delta u. \end{aligned} \quad (10.43)$$

The form that orders the PDE as the Picard iteration terms plus the Newton method's derivative terms becomes

$$\begin{aligned} \frac{u - u^{(1)}}{\Delta t} = & \nabla \cdot (\alpha(u^-) \nabla u) + f(u^-) + \\ & \gamma (\nabla \cdot (\alpha'(u^-) (u - u^-) \nabla u^-) + f'(u^-) (u - u^-)). \end{aligned} \quad (10.44)$$

The Picard and full Newton versions correspond to $\gamma = 0$ and $\gamma = 1$, respectively.

Derivation with alternative notation. Some may prefer to derive the linearized PDE for δu using the more compact notation. We start with inserting $u^n = u^- + \delta u$ to get

$$\frac{u^- + \delta u - u^{n-1}}{\Delta t} = \nabla \cdot (\alpha(u^- + \delta u) \nabla (u^- + \delta u)) + f(u^- + \delta u).$$

Taylor expanding,

$$\begin{aligned} \alpha(u^- + \delta u) & \approx \alpha(u^-) + \alpha'(u^-) \delta u, \\ f(u^- + \delta u) & \approx f(u^-) + f'(u^-) \delta u, \end{aligned}$$

and inserting these expressions gives a less cluttered PDE for δu :

$$\begin{aligned} \frac{u^- + \delta u - u^{n-1}}{\Delta t} = & \nabla \cdot (\alpha(u^-) \nabla u^-) + f(u^-) + \\ & \nabla \cdot (\alpha(u^-) \nabla \delta u) + \nabla \cdot (\alpha'(u^-) \delta u \nabla u^-) + \\ & \nabla \cdot (\alpha'(u^-) \delta u \nabla \delta u) + f'(u^-) \delta u. \end{aligned}$$

10.3.4 Crank-Nicolson discretization

A Crank-Nicolson discretization of (10.30) applies a centered difference at $t_{n+\frac{1}{2}}$:

$$[D_t u = \nabla \cdot (\alpha(u) \nabla u) + f(u)]^{n+\frac{1}{2}}.$$

The standard technique is to apply an arithmetic average for quantities defined between two mesh points, e.g.,

$$u^{n+\frac{1}{2}} \approx \frac{1}{2}(u^n + u^{n+1}).$$

However, with nonlinear terms we have many choices of formulating an arithmetic mean:

$$[f(u)]^{n+\frac{1}{2}} \approx f\left(\frac{1}{2}(u^n + u^{n+1})\right) = [f(\bar{u}^t)]^{n+\frac{1}{2}}, \quad (10.45)$$

$$[f(u)]^{n+\frac{1}{2}} \approx \frac{1}{2}(f(u^n) + f(u^{n+1})) = [\overline{f(u)}^t]^{n+\frac{1}{2}}, \quad (10.46)$$

$$[\alpha(u) \nabla u]^{n+\frac{1}{2}} \approx \alpha\left(\frac{1}{2}(u^n + u^{n+1})\right) \nabla\left(\frac{1}{2}(u^n + u^{n+1})\right) = [\alpha(\bar{u}^t) \nabla \bar{u}^t]^{n+\frac{1}{2}}, \quad (10.47)$$

$$[\alpha(u) \nabla u]^{n+\frac{1}{2}} \approx \frac{1}{2}(\alpha(u^n) + \alpha(u^{n+1})) \nabla\left(\frac{1}{2}(u^n + u^{n+1})\right) = [\overline{\alpha(u)}^t \nabla \bar{u}^t]^{n+\frac{1}{2}}, \quad (10.48)$$

$$[\alpha(u) \nabla u]^{n+\frac{1}{2}} \approx \frac{1}{2}(\alpha(u^n) \nabla u^n + \alpha(u^{n+1}) \nabla u^{n+1}) = [\overline{\alpha(u) \nabla u}^t]^{n+\frac{1}{2}}. \quad (10.49)$$

A big question is whether there are significant differences in accuracy between taking the products of arithmetic means or taking the arithmetic mean of products. Exercise 10.6 investigates this question, and the answer is that the approximation is $\mathcal{O}(\Delta t^2)$ in both cases.

10.4 1D stationary nonlinear differential equations

Section 10.3 presented methods for linearizing time-discrete PDEs directly prior to discretization in space. We can alternatively carry out the discretization in space of the time-discrete nonlinear PDE problem and get a system of nonlinear algebraic equations, which can be solved by Picard iteration or Newton's method as presented in Section 10.2. This latter approach will now be described in detail.

We shall work with the 1D problem

$$-(\alpha(u)u')' + au = f(u), \quad x \in (0, L), \quad \alpha(u(0))u'(0) = C, \quad u(L) = D. \quad (10.50)$$

The problem (10.50) arises from the stationary limit of a diffusion equation,

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x} \left(\alpha(u) \frac{\partial u}{\partial x} \right) - au + f(u), \quad (10.51)$$

as $t \rightarrow \infty$ and $\partial u / \partial t \rightarrow 0$. Alternatively, the problem (10.50) arises at each time level from implicit time discretization of (10.51). For example, a Backward Euler scheme for (10.51) leads to

$$\frac{u^n - u^{n-1}}{\Delta t} = \frac{d}{dx} \left(\alpha(u^n) \frac{du^n}{dx} \right) - au^n + f(u^n). \quad (10.52)$$

Introducing $u(x)$ for $u^n(x)$, $u^{(1)}$ for u^{n-1} , and defining $f(u)$ in (10.50) to be $f(u)$ in (10.52) plus $u^{n-1}/\Delta t$, gives (10.50) with $a = 1/\Delta t$.

10.4.1 Finite difference discretization

Since the technical steps in finite difference discretization in space are so much simpler than the steps in the finite element method, we start with finite difference to illustrate the concept of handling this nonlinear problem and minimize the spatial discretization details.

The nonlinearity in the differential equation (10.50) poses no more difficulty than a variable coefficient, as in the term $(\alpha(x)u')'$. We can therefore use a standard finite difference approach to discretizing the Laplace term with a variable coefficient:

$$[-D_x \alpha D_x u + au = f]_i.$$

Writing this out for a uniform mesh with points $x_i = i\Delta x$, $i = 0, \dots, N_x$, leads to

$$-\frac{1}{\Delta x^2} \left(\alpha_{i+\frac{1}{2}}(u_{i+1} - u_i) - \alpha_{i-\frac{1}{2}}(u_i - u_{i-1}) \right) + au_i = f(u_i). \quad (10.53)$$

This equation is valid at all the mesh points $i = 0, 1, \dots, N_x - 1$. At $i = N_x$ we have the Dirichlet condition $u_i = D$. The only difference from the case with $(\alpha(x)u')'$ and $f(x)$ is that now α and f are functions of u and not only of x : $(\alpha(u(x))u')'$ and $f(u(x))$.

The quantity $\alpha_{i+\frac{1}{2}}$, evaluated between two mesh points, needs a comment. Since α depends on u and u is only known at the mesh points, we need to express $\alpha_{i+\frac{1}{2}}$ in terms of u_i and u_{i+1} . For this purpose we use an arithmetic mean, although a harmonic mean is also common in this context if α features large jumps. There are two choices of arithmetic means:

$$\alpha_{i+\frac{1}{2}} \approx \alpha\left(\frac{1}{2}(u_i + u_{i+1})\right) = [\alpha(\bar{u}^x)]^{i+\frac{1}{2}}, \quad (10.54)$$

$$\alpha_{i+\frac{1}{2}} \approx \frac{1}{2}(\alpha(u_i) + \alpha(u_{i+1})) = [\overline{\alpha(u)}^x]^{i+\frac{1}{2}} \quad (10.55)$$

Equation (10.53) with the latter approximation then looks like

$$\begin{aligned} -\frac{1}{2\Delta x^2} ((\alpha(u_i) + \alpha(u_{i+1}))(u_{i+1} - u_i) - (\alpha(u_{i-1}) + \alpha(u_i))(u_i - u_{i-1})) \\ + au_i = f(u_i), \end{aligned} \quad (10.56)$$

or written more compactly,

$$[-D_x \overline{\alpha}^x D_x u + au = f]_i.$$

At mesh point $i = 0$ we have the boundary condition $\alpha(u)u' = C$, which is discretized by

$$[\alpha(u)D_{2x}u = C]_0,$$

meaning

$$\alpha(u_0) \frac{u_1 - u_{-1}}{2\Delta x} = C. \quad (10.57)$$

The fictitious value u_{-1} can be eliminated with the aid of (10.56) for $i = 0$. Formally, (10.56) should be solved with respect to u_{i-1} and that value (for $i = 0$) should be inserted in (10.57), but it is algebraically much easier to do it the other way around. Alternatively, one can use a ghost cell $[-\Delta x, 0]$ and update the u_{-1} value in the ghost cell according to (10.57) after every Picard or Newton iteration. Such an approach means that we use a known u_{-1} value in (10.56) from the previous iteration.

10.4.2 Solution of algebraic equations

The structure of the equation system. The nonlinear algebraic equations (10.56) are of the form $A(u)u = b(u)$ with

$$\begin{aligned} A_{i,i} &= \frac{1}{2\Delta x^2}(\alpha(u_{i-1}) + 2\alpha(u_i) + \alpha(u_{i+1})) + a, \\ A_{i,i-1} &= -\frac{1}{2\Delta x^2}(\alpha(u_{i-1}) + \alpha(u_i)), \\ A_{i,i+1} &= -\frac{1}{2\Delta x^2}(\alpha(u_i) + \alpha(u_{i+1})), \\ b_i &= f(u_i). \end{aligned}$$

The matrix $A(u)$ is tridiagonal: $A_{i,j} = 0$ for $j > i + 1$ and $j < i - 1$.

The above expressions are valid for internal mesh points $1 \leq i \leq N_x - 1$. For $i = 0$ we need to express $u_{i-1} = u_{-1}$ in terms of u_1 using (10.57):

$$u_{-1} = u_1 - \frac{2\Delta x}{\alpha(u_0)}C. \quad (10.58)$$

This value must be inserted in $A_{0,0}$. The expression for $A_{i,i+1}$ applies for $i = 0$, and $A_{i,i-1}$ does not enter the system when $i = 0$.

Regarding the last equation, its form depends on whether we include the Dirichlet condition $u(L) = D$, meaning $u_{N_x} = D$, in the nonlinear algebraic equation system or not. Suppose we choose $(u_0, u_1, \dots, u_{N_x-1})$ as unknowns, later referred to as *systems without Dirichlet conditions*. The last equation corresponds to $i = N_x - 1$. It involves the boundary value u_{N_x} , which is substituted by D . If the unknown vector includes the boundary value, $(u_0, u_1, \dots, u_{N_x})$, later referred to as *system including Dirichlet conditions*, the equation for $i = N_x - 1$ just involves the unknown u_{N_x} , and the final equation becomes $u_{N_x} = D$, corresponding to $A_{i,i} = 1$ and $b_i = D$ for $i = N_x$.

Picard iteration. The obvious Picard iteration scheme is to use previously computed values of u_i in $A(u)$ and $b(u)$, as described more in detail in Section 10.2. With the notation u^- for the most recently computed value of u , we have the system $F(u) \approx \hat{F}(u) = A(u^-)u - b(u^-)$, with $F = (F_0, F_1, \dots, F_m)$, $u = (u_0, u_1, \dots, u_m)$. The index m is N_x if the system includes the Dirichlet condition as a separate equation and $N_x - 1$ otherwise. The matrix $A(u^-)$ is tridiagonal, so the solution procedure is to fill a tridiagonal matrix data structure and the right-hand side

vector with the right numbers and call a Gaussian elimination routine for tridiagonal linear systems.

Mesh with two cells. It helps on the understanding of the details to write out all the mathematics in a specific case with a small mesh, say just two cells ($N_x = 2$). We use u_i^- for the i -th component in u^- .

The starting point is the basic expressions for the nonlinear equations at mesh point $i = 0$ and $i = 1$ are

$$A_{0,-1}u_{-1} + A_{0,0}u_0 + A_{0,1}u_1 = b_0, \quad (10.59)$$

$$A_{1,0}u_0 + A_{1,1}u_1 + A_{1,2}u_2 = b_1. \quad (10.60)$$

Equation (10.59) written out reads

$$\begin{aligned} & \frac{1}{2\Delta x^2}(-(\alpha(u_{-1}) + \alpha(u_0))u_{-1} + \\ & (\alpha(u_{-1}) + 2\alpha(u_0) + \alpha(u_1))u_0 - \\ & (\alpha(u_0) + \alpha(u_1)))u_1 + au_0 = f(u_0). \end{aligned}$$

We must then replace u_{-1} by (10.58). With Picard iteration we get

$$\begin{aligned} & \frac{1}{2\Delta x^2}(-(\alpha(u_{-1}^-) + 2\alpha(u_0^-) + \alpha(u_1^-))u_1 + \\ & (\alpha(u_{-1}^-) + 2\alpha(u_0^-) + \alpha(u_1^-))u_0 + au_0 \\ & = f(u_0^-) - \frac{1}{\alpha(u_0^-)\Delta x}(\alpha(u_{-1}^-) + \alpha(u_0^-))C, \end{aligned}$$

where

$$u_{-1}^- = u_1^- - \frac{2\Delta x}{\alpha(u_0^-)}C.$$

Equation (10.60) contains the unknown u_2 for which we have a Dirichlet condition. In case we omit the condition as a separate equation, (10.60) with Picard iteration becomes

$$\begin{aligned} \frac{1}{2\Delta x^2} & (-(\alpha(u_0^-) + \alpha(u_1^-))u_0 + \\ & (\alpha(u_0^-) + 2\alpha(u_1^-) + \alpha(u_2^-))u_1 - \\ & (\alpha(u_1^-) + \alpha(u_2^-)))u_2 + au_1 = f(u_1^-). \end{aligned}$$

We must now move the u_2 term to the right-hand side and replace all occurrences of u_2 by D :

$$\begin{aligned} \frac{1}{2\Delta x^2} & (-(\alpha(u_0^-) + \alpha(u_1^-))u_0 + \\ & (\alpha(u_0^-) + 2\alpha(u_1^-) + \alpha(D))u_1 + au_1 \\ & = f(u_1^-) + \frac{1}{2\Delta x^2}(\alpha(u_1^-) + \alpha(D))D. \end{aligned}$$

The two equations can be written as a 2×2 system:

$$\begin{pmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \end{pmatrix},$$

where

$$B_{0,0} = \frac{1}{2\Delta x^2}(\alpha(u_{-1}^-) + 2\alpha(u_0^-) + \alpha(u_1^-)) + a \quad (10.61)$$

$$B_{0,1} = -\frac{1}{2\Delta x^2}(\alpha(u_{-1}^-) + 2\alpha(u_0^-) + \alpha(u_1^-)), \quad (10.62)$$

$$B_{1,0} = -\frac{1}{2\Delta x^2}(\alpha(u_0^-) + \alpha(u_1^-)), \quad (10.63)$$

$$B_{1,1} = \frac{1}{2\Delta x^2}(\alpha(u_0^-) + 2\alpha(u_1^-) + \alpha(D)) + a, \quad (10.64)$$

$$d_0 = f(u_0^-) - \frac{1}{\alpha(u_0^-)\Delta x}(\alpha(u_{-1}^-) + \alpha(u_0^-))C, \quad (10.65)$$

$$d_1 = f(u_1^-) + \frac{1}{2\Delta x^2}(\alpha(u_1^-) + \alpha(D))D. \quad (10.66)$$

The system with the Dirichlet condition becomes

$$\begin{pmatrix} B_{0,0} & B_{0,1} & 0 \\ B_{1,0} & B_{1,1} & B_{1,2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ D \end{pmatrix},$$

with

$$B_{1,1} = \frac{1}{2\Delta x^2}(\alpha(u_0^-) + 2\alpha(u_1^-) + \alpha(u_2)) + a, \quad (10.67)$$

$$B_{1,2} = -\frac{1}{2\Delta x^2}(\alpha(u_1^-) + \alpha(u_2)), \quad (10.68)$$

$$d_1 = f(u_1^-). \quad (10.69)$$

Other entries are as in the 2×2 system.

Newton's method. The Jacobian must be derived in order to use Newton's method. Here it means that we need to differentiate $F(u) = A(u)u - b(u)$ with respect to the unknown parameters u_0, u_1, \dots, u_m ($m = N_x$ or $m = N_x - 1$, depending on whether the Dirichlet condition is included in the nonlinear system $F(u) = 0$ or not). Nonlinear equation number i of (10.56) has the structure

$$F_i = A_{i,i-1}(u_{i-1}, u_i)u_{i-1} + A_{i,i}(u_{i-1}, u_i, u_{i+1})u_i + A_{i,i+1}(u_i, u_{i+1})u_{i+1} - b_i(u_i).$$

Computing the Jacobian requires careful differentiation. For example,

$$\begin{aligned} \frac{\partial}{\partial u_i} (A_{i,i}(u_{i-1}, u_i, u_{i+1})u_i) &= \frac{\partial A_{i,i}}{\partial u_i} u_i + A_{i,i} \frac{\partial u_i}{\partial u_i} \\ &= \frac{\partial}{\partial u_i} \left(\frac{1}{2\Delta x^2}(\alpha(u_{i-1}) + 2\alpha(u_i) + \alpha(u_{i+1})) + a \right) u_i + \\ &\quad \frac{1}{2\Delta x^2}(\alpha(u_{i-1}) + 2\alpha(u_i) + \alpha(u_{i+1})) + a \\ &= \frac{1}{2\Delta x^2}(2\alpha'(u_i)u_i + \alpha(u_{i-1}) + 2\alpha(u_i) + \alpha(u_{i+1})) + a. \end{aligned}$$

The complete Jacobian becomes

$$\begin{aligned}
F_{i,i} &= \frac{\partial F_i}{\partial u_i} = \frac{\partial A_{i,i-1}}{\partial u_i} u_{i-1} + \frac{\partial A_{i,i}}{\partial u_i} u_i + A_{i,i} + \frac{\partial A_{i,i+1}}{\partial u_i} u_{i+1} - \frac{\partial b_i}{\partial u_i} \\
&= \frac{1}{2\Delta x^2} (-\alpha'(u_i)u_{i-1} + 2\alpha'(u_i)u_i + \alpha(u_{i-1}) + 2\alpha(u_i) + \alpha(u_{i+1})) + \\
&\quad a - \frac{1}{2\Delta x^2} \alpha'(u_i)u_{i+1} - b'(u_i), \\
F_{i,i-1} &= \frac{\partial F_i}{\partial u_{i-1}} = \frac{\partial A_{i,i-1}}{\partial u_{i-1}} u_{i-1} + A_{i-1,i} + \frac{\partial A_{i,i}}{\partial u_{i-1}} u_i - \frac{\partial b_i}{\partial u_{i-1}} \\
&= \frac{1}{2\Delta x^2} (-\alpha'(u_{i-1})u_{i-1} - (\alpha(u_{i-1}) + \alpha(u_i)) + \alpha'(u_{i-1})u_i), \\
F_{i,i+1} &= \frac{\partial F_i}{\partial u_{i+1}} = \frac{\partial A_{i,i+1}}{\partial u_{i+1}} u_{i+1} + A_{i+1,i} + \frac{\partial A_{i,i}}{\partial u_{i+1}} u_i - \frac{\partial b_i}{\partial u_{i+1}} \\
&= \frac{1}{2\Delta x^2} (-\alpha'(u_{i+1})u_{i+1} - (\alpha(u_i) + \alpha(u_{i+1})) + \alpha'(u_{i+1})u_i).
\end{aligned}$$

The explicit expression for nonlinear equation number i , $F_i(u_0, u_1, \dots)$, arises from moving the $f(u_i)$ term in (10.56) to the left-hand side:

$$\begin{aligned}
F_i &= -\frac{1}{2\Delta x^2} ((\alpha(u_i) + \alpha(u_{i+1}))(u_{i+1} - u_i) - (\alpha(u_{i-1}) + \alpha(u_i))(u_i - u_{i-1})) \\
&\quad + au_i - f(u_i) = 0.
\end{aligned} \tag{10.70}$$

At the boundary point $i = 0$, u_{-1} must be replaced using the formula (10.58). When the Dirichlet condition at $i = N_x$ is not a part of the equation system, the last equation $F_m = 0$ for $m = N_x - 1$ involves the quantity u_{N_x-1} which must be replaced by D . If u_{N_x} is treated as an unknown in the system, the last equation $F_m = 0$ has $m = N_x$ and reads

$$F_{N_x}(u_0, \dots, u_{N_x}) = u_{N_x} - D = 0.$$

Similar replacement of u_{-1} and u_{N_x} must be done in the Jacobian for the first and last row. When u_{N_x} is included as an unknown, the last row in the Jacobian must help implement the condition $\delta u_{N_x} = 0$, since we assume that u contains the right Dirichlet value at the beginning of the iteration ($u_{N_x} = D$), and then the Newton update should be zero for $i = 0$, i.e., $\delta u_{N_x} = 0$. This also forces the right-hand side to be $b_i = 0$, $i = N_x$.

We have seen, and can see from the present example, that the linear system in Newton's method contains all the terms present in the system that arises in the Picard iteration method. The extra terms in Newton's method can be multiplied by a factor such that it is easy to program

one linear system and set this factor to 0 or 1 to generate the Picard or Newton system.

10.4.3 Galerkin-type discretization

For a Galerkin-type discretization, which may be developed into a finite element method, we first need to derive the variational problem. Let V be an appropriate function space with basis functions $\{\psi_i\}_{i \in \mathcal{I}_s}$. Because of the Dirichlet condition at $x = L$ we require $\psi_i(L) = 0$, $i \in \mathcal{I}_s$. The approximate solution is written as $u = D + \sum_{j \in \mathcal{I}_s} c_j \psi_j$, where the term D can be viewed as a boundary function needed to implement the Dirichlet condition $u(L) = D$. We remark that the boundary function is D rather than Dx/L because of the Neumann condition at $x = 0$.

Using Galerkin's method, we multiply the differential equation by any $v \in V$ and integrate terms with second-order derivatives by parts:

$$\int_0^L \alpha(u) u' v' \, dx + \int_0^L a u v \, dx = \int_0^L f(u) v \, dx + [\alpha(u) u' v]_0^L, \quad \forall v \in V.$$

The Neumann condition at the boundary $x = 0$ is inserted in the boundary term:

$$[\alpha(u) u' v]_0^L = \alpha(u(L)) u'(L) v(L) - \alpha(u(0)) u'(0) v(0) = 0 - Cv(0) = -Cv(0).$$

(Recall that since $\psi_i(L) = 0$, any linear combination v of the basis functions also vanishes at $x = L$: $v(L) = 0$.) The variational problem is then: find $u \in V$ such that

$$\int_0^L \alpha(u) u' v' \, dx + \int_0^L a u v \, dx = \int_0^L f(u) v \, dx - Cv(0), \quad \forall v \in V. \tag{10.71}$$

To derive the algebraic equations, we note that $\forall v \in V$ is equivalent with $v = \psi_i$ for $i \in \mathcal{I}_s$. Setting $u = D + \sum_j c_j \psi_j$ and sorting terms results in the linear system

$$\begin{aligned} & \sum_{j \in \mathcal{I}_s} \left(\int_0^L \left(\alpha(D + \sum_{k \in \mathcal{I}_s} c_k \psi_k) \psi'_j \psi'_i + a \psi_i \psi_j \right) dx \right) c_j \\ &= \int_0^L f(D + \sum_{k \in \mathcal{I}_s} c_k \psi_k) \psi_i dx - C \psi_i(0), \quad i \in \mathcal{I}_s. \end{aligned} \quad (10.72)$$

Fundamental integration problem. Methods that use the Galerkin or weighted residual method face a fundamental difficulty in nonlinear problems: how can we integrate terms like $\int_0^L \alpha(\sum_k c_k \psi_k) \psi'_i \psi'_j dx$ and $\int_0^L f(\sum_k c_k \psi_k) \psi_i dx$ when we do not know the c_k coefficients in the argument of the α function? We can resort to numerical integration, provided an approximate $\sum_k c_k \psi_k$ can be used for the argument u in f and α . This is the approach used in computer programs.

However, if we want to look more mathematically into the structure of the algebraic equations generated by the finite element method in nonlinear problems, and compare such equations with those arising in the finite difference method, we need techniques that enable integration of expressions like $\int_0^L f(\sum_k c_k \psi_k) \psi_i dx$ *by hand*. Two such techniques will be shown: the group finite element and numerical integration based on the nodes only. Both techniques are approximate, but they allow us to see the difference equations in the finite element method. The details are worked out in Appendix 10.6. Some readers will prefer to dive into these symbolic calculations to gain more understanding of nonlinear finite element equations, while others will prefer to continue with computational algorithms (in the next two sections) rather than analysis.

10.4.4 Picard iteration defined from the variational form

Consider the problem (10.50) with the corresponding variational form (10.71). Our aim is to define a Picard iteration based on this variational form without any attempt to compute integrals symbolically as in the previous three sections. The idea in Picard iteration is to use a previously computed u value in the nonlinear functions $\alpha(u)$ and $f(u)$. Let u^- be the available approximation to u from the previous iteration. The linearized variational form for Picard iteration is then

$$\int_0^L (\alpha(u^-) u' v' + a u v) dx = \int_0^L f(u^-) v dx - C v(0), \quad \forall v \in V. \quad (10.73)$$

This is a linear problem $a(u, v) = L(v)$ with bilinear and linear forms

$$a(u, v) = \int_0^L (\alpha(u^-)u'v' + auv) dx, \quad L(v) = \int_0^L f(u^-)v dx - Cv(0).$$

Make sure to distinguish the coefficient a in auv from the differential equation from the a in the abstract bilinear form notation $a(\cdot, \cdot)$.

The linear system associated with (10.73) is computed the standard way. Technically, we are back to solving $-(\alpha(x)u')' + au = f(x)$. The unknown u is sought on the form $u = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j$, with $B(x) = D$ and $\psi_i = \varphi_{\nu(i)}$, $\nu(i) = i + 1$, and $\mathcal{I}_s = \{0, 1, \dots, N = N_n - 2\}$.

10.4.5 Newton's method defined from the variational form

Application of Newton's method to the nonlinear variational form (10.71) arising from the problem (10.50) requires identification of the nonlinear algebraic equations $F_i = 0$. Although we originally denoted the unknowns in nonlinear algebraic equations by u_0, \dots, u_N , it is in the present context most natural to have the unknowns as c_0, \dots, c_N and write

$$F_i(c_0, \dots, c_N) = 0, \quad i \in \mathcal{I}_s,$$

and define the Jacobian as $J_{i,j} = \partial F_i / \partial c_j$ for $i, j \in \mathcal{I}_s$.

The specific form of the equations $F_i = 0$ follows from the variational form

$$\int_0^L (\alpha(u)u'v' + auv) dx = \int_0^L f(u)v dx - Cv(0), \quad \forall v \in V,$$

by choosing $v = \psi_i$, $i \in \mathcal{I}_s$, and setting $u = \sum_{j \in \mathcal{I}_s} c_j \psi_j$, maybe with a boundary function to incorporate Dirichlet conditions, we get

$$F_i = \int_0^L (\alpha(u)u'\psi'_i + a u \psi_i - f(u)\psi_i) dx + C\psi_i(0) = 0, \quad i \in \mathcal{I}_s. \quad (10.74)$$

In the differentiations leading to the Jacobian we will frequently use the results

$$\frac{\partial u}{\partial c_j} = \frac{\partial}{\partial c_j} \sum_k c_k \psi_k = \psi_j, \quad \frac{\partial u'}{\partial c_j} = \frac{\partial}{\partial c_j} \sum_k c_k \psi'_k = \psi'_j.$$

The derivation of the Jacobian of (10.74) goes as

$$\begin{aligned}
 J_{i,j} &= \frac{\partial F_i}{\partial c_j} = \int_0^L \frac{\partial}{\partial c_j} (\alpha(u)u'\psi'_i + au\psi_i - f(u)\psi_i) dx \\
 &= \int_0^L ((\alpha'(u)\frac{\partial u}{\partial c_j}u' + \alpha(u)\frac{\partial u'}{\partial c_j})\psi'_i + a\frac{\partial u}{\partial c_j}\psi_i - f'(u)\frac{\partial u}{\partial c_j}\psi_i) dx \\
 &= \int_0^L ((\alpha'(u)\psi_j u' + \alpha(u)\psi'_j)\psi'_i + a\psi_j\psi_i - f'(u)\psi_j\psi_i) dx \\
 &= \int_0^L (\alpha'(u)u'\psi'_i\psi_j + \alpha(u)\psi'_i\psi'_j + (a - f(u))\psi_i\psi_j) dx
 \end{aligned} \tag{10.75}$$

One must be careful about the prime symbol as differentiation!

In α' the derivative is with respect to the independent variable in the α function, and that is u , so

$$\alpha' = \frac{d\alpha}{du},$$

while in u' the differentiation is with respect to x , so

$$u' = \frac{du}{dx}.$$

Similarly, f is a function of u , so f' means df/du .

When calculating the right-hand side vector F_i and the coefficient matrix $J_{i,j}$ in the linear system to be solved in each Newton iteration, one must use a previously computed u , denoted by u^- , for the symbol u in (10.74) and (10.75). With this notation we have

$$F_i = \int_0^L (\alpha(u^-)u'^-\psi'_i + (a - f(u^-))\psi_i) dx - C\psi_i(0), \quad i \in \mathcal{I}_s, \tag{10.76}$$

$$J_{i,j} = \int_0^L (\alpha'(u^-)u'^-\psi'_i\psi_j + \alpha(u^-)\psi'_i\psi'_j + (a - f(u^-))\psi_i\psi_j) dx, \quad i, j \in \mathcal{I}_s. \tag{10.77}$$

These expressions can be used for any basis $\{\psi_i\}_{i \in \mathcal{I}_s}$. Choosing finite element functions for ψ_i , one will normally want to compute the integral contribution cell by cell, working in a reference cell. To this end, we restrict the integration to one cell and transform the cell to $[-1, 1]$. The most recently computed approximation u^- to u becomes $\tilde{u}^- = \sum_t \tilde{u}_t^{-1} \tilde{\varphi}_t(X)$ over the reference element, where \tilde{u}_t^{-1} is the value of u^- at global node (or degree of freedom) $q(e, t)$ corresponding to the local node t (or degree of freedom). The formulas (10.76) and (10.77) then change to

$$\tilde{F}_r^{(e)} = \int_{-1}^1 (\alpha(\tilde{u}^-) \tilde{u}^{-\prime} \tilde{\varphi}'_r + (a - f(\tilde{u}^-)) \tilde{\varphi}_r) \det J \, dX - C \tilde{\varphi}_r(0), \quad (10.78)$$

$$\tilde{J}_{r,s}^{(e)} = \int_{-1}^1 (\alpha'(\tilde{u}^-) \tilde{u}^{-\prime} \tilde{\varphi}'_r \tilde{\varphi}_s + \alpha(\tilde{u}^-) \tilde{\varphi}'_r \tilde{\varphi}'_s + (a - f(\tilde{u}^-)) \tilde{\varphi}_r \tilde{\varphi}_s) \det J \, dX, \quad (10.79)$$

with $r, s \in I_d$ runs over the local degrees of freedom.

Many finite element programs require the user to provide F_i and $J_{i,j}$. Some programs, like FEniCS, are capable of automatically deriving $J_{i,j}$ if F_i is specified.

Dirichlet conditions. Incorporation of the Dirichlet values by assembling contributions from all degrees of freedom and then modifying the linear system can obviously be applied to Picard iteration as that method involves a standard linear system. In the Newton system, however, the unknown is a correction δu to the solution. Dirichlet conditions are implemented by inserting them in the initial guess u^- for the Newton iteration and implementing $\delta u_i = 0$ for all known degrees of freedom. The manipulation of the linear system follows exactly the algorithm in the linear problems, the only difference being that the known values are zero.

10.5 Multi-dimensional PDE problems

The fundamental ideas in the derivation of F_i and $J_{i,j}$ in the 1D model problem are easily generalized to multi-dimensional problems. Nevertheless, the expressions involved are slightly different, with derivatives in x replaced by ∇ , so we present some examples below in detail.

10.5.1 Finite element discretization

As an example, a Backward Euler discretization of the PDE

$$u_t = \nabla \cdot (\alpha(u) \nabla u) + f(u), \quad (10.80)$$

gives the nonlinear time-discrete PDEs

$$u^n - \Delta t \nabla \cdot (\alpha(u^n) \nabla u^n) + f(u^n) = u^{n-1}.$$

We may alternatively write these equations with u for u^n and $u^{(1)}$ for u^{n-1} :

$$u - \Delta t \nabla \cdot (\alpha(u) \nabla u) - \Delta t f(u) = u^{(1)}.$$

Understand the meaning of the symbol u in various formulas!

Note that the mathematical meaning of the symbol u changes in the above equations: $u(\mathbf{x}, t)$ is the exact solution of (10.80), $u^n(\mathbf{x})$ is an approximation to the exact solution at $t = t_n$, while $u(\mathbf{x})$ in the latter equation is a synonym for u^n . Below, this $u(\mathbf{x})$ will be approximated by a new $u = \sum_k c_k \psi_k(\mathbf{x})$ in space, and then the actual u symbol used in the Picard and Newton iterations is a further approximation of $\sum_k c_k \psi_k$ arising from the nonlinear iteration algorithm.

Much literature reserves u for the exact solution, uses $u_h(x, t)$ for the finite element solution that varies continuously in time, introduces perhaps u_h^n as the approximation of u_h at time t_n , arising from some time discretization, and then finally applies $u_h^{n,k}$ for the approximation to u_h^n in the k -th iteration of a Picard or Newton method. The converged solution at the previous time step can be called u_h^{n-1} , but then this quantity is an approximate solution of the nonlinear equations (at the previous time level), while the counterpart u_h^n is formally the exact solution of the nonlinear equations at the current time level. The various symbols in the mathematics can in this way be clearly distinguished. However, we favor to use u for the quantity that is most naturally called u in the code, and that is the most recent approximation to the solution, e.g., named $u_h^{n,k}$ above. This is also the key quantity of interest in mathematical

derivations of algorithms as well. Choosing u this way makes the most important mathematical cleaner than using more cluttered notation as $u_h^{n,k}$. We therefore introduce other symbols for other versions of the unknown function. It is most appropriate for us to say that $u_e(\mathbf{x}, t)$ is the exact solution, u^n in the equation above is the approximation to $u_e(\mathbf{x}, t_n)$ after time discretization, and u is the spatial approximation to u^n from the most recent iteration in a nonlinear iteration method.

Let us assume homogeneous Neumann conditions on the entire boundary for simplicity in the boundary term. The variational form becomes: find $u \in V$ such that

$$\int_{\Omega} (uv + \Delta t \alpha(u) \nabla u \cdot \nabla v - \Delta t f(u)v - u^{(1)}v) dx = 0, \quad \forall v \in V. \quad (10.81)$$

The nonlinear algebraic equations follow from setting $v = \psi_i$ and using the representation $u = \sum_k c_k \psi_k$, which we just write as

$$F_i = \int_{\Omega} (u\psi_i + \Delta t \alpha(u) \nabla u \cdot \nabla \psi_i - \Delta t f(u)\psi_i - u^{(1)}\psi_i) dx. \quad (10.82)$$

Picard iteration needs a linearization where we use the most recent approximation u^- to u in α and f :

$$F_i \approx \hat{F}_i = \int_{\Omega} (u\psi_i + \Delta t \alpha(u^-) \nabla u \cdot \nabla \psi_i - \Delta t f(u^-)\psi_i - u^{(1)}\psi_i) dx. \quad (10.83)$$

The equations $\hat{F}_i = 0$ are now linear and we can easily derive a linear system $\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i$, $i \in \mathcal{I}_s$, for the unknown coefficients $\{c_i\}_{i \in \mathcal{I}_s}$ by inserting $u = \sum_j c_j \psi_j$. We get

$$A_{i,j} = \int_{\Omega} (\psi_j \psi_i + \Delta t \alpha(u^-) \nabla \psi_j \cdot \nabla \psi_i) dx, \quad b_i = \int_{\Omega} (\Delta t f(u^-)\psi_i + u^{(1)}\psi_i) dx.$$

In Newton's method we need to evaluate F_i with the known value u^- for u :

$$F_i \approx \hat{F}_i = \int_{\Omega} (u^- \psi_i + \Delta t \alpha(u^-) \nabla u^- \cdot \nabla \psi_i - \Delta t f(u^-) \psi_i - u^{(1)} \psi_i) dx. \quad (10.84)$$

The Jacobian is obtained by differentiating (10.82) and using

$$\frac{\partial u}{\partial c_j} = \sum_k \frac{\partial}{\partial c_j} c_k \psi_k = \psi_j, \quad (10.85)$$

$$\frac{\partial \nabla u}{\partial c_j} = \sum_k \frac{\partial}{\partial c_j} c_k \nabla \psi_k = \nabla \psi_j. \quad (10.86)$$

The result becomes

$$J_{i,j} = \frac{\partial F_i}{\partial c_j} = \int_{\Omega} (\psi_j \psi_i + \Delta t \alpha'(u) \psi_j \nabla u \cdot \nabla \psi_i + \Delta t \alpha(u) \nabla \psi_j \cdot \nabla \psi_i - \Delta t f'(u) \psi_j \psi_i) dx. \quad (10.87)$$

The evaluation of $J_{i,j}$ as the coefficient matrix in the linear system in Newton's method applies the known approximation u^- for u :

$$J_{i,j} = \int_{\Omega} (\psi_j \psi_i + \Delta t \alpha'(u^-) \psi_j \nabla u^- \cdot \nabla \psi_i + \Delta t \alpha(u^-) \nabla \psi_j \cdot \nabla \psi_i - \Delta t f'(u^-) \psi_j \psi_i) dx. \quad (10.88)$$

Hopefully, this example also shows how convenient the notation with u and u^- is: the unknown to be computed is always u and linearization by inserting known (previously computed) values is a matter of adding an underscore. One can take great advantage of this quick notation in software [25].

Non-homogeneous Neumann conditions. A natural physical flux condition for the PDE (10.80) takes the form of a non-homogeneous Neumann condition

$$-\alpha(u) \frac{\partial u}{\partial n} = g, \quad \mathbf{x} \in \partial \Omega_N, \quad (10.89)$$

where g is a prescribed function and $\partial \Omega_N$ is a part of the boundary of the domain Ω . From integrating $\int_{\Omega} \nabla \cdot (\alpha \nabla u) dx$ by parts, we get a boundary term

$$\int_{\partial\Omega_N} \alpha(u) \frac{\partial u}{\partial u} v \, ds. \quad (10.90)$$

Inserting the condition (10.89) into this term results in an integral over prescribed values:

$$-\int_{\partial\Omega_N} g v \, ds.$$

The nonlinearity in the $\alpha(u)$ coefficient condition (10.89) therefore does not contribute with a nonlinearity in the variational form.

Robin conditions. Heat conduction problems often apply a kind of Newton's cooling law, also known as a Robin condition, at the boundary:

$$-\alpha(u) \frac{\partial u}{\partial u} = h(u)(u - T_s(t)), \quad \mathbf{x} \in \partial\Omega_R, \quad (10.91)$$

where $h(u)$ is a heat transfer coefficient between the body (Ω) and its surroundings, T_s is the temperature of the surroundings, and $\partial\Omega_R$ is a part of the boundary where this Robin condition applies. The boundary integral (10.90) now becomes

$$\int_{\partial\Omega_R} h(u)(u - T_s(T))v \, ds.$$

In many physical applications, $h(u)$ can be taken as constant, and then the boundary term is linear in u , otherwise it is nonlinear and contributes to the Jacobian in a Newton method. Linearization in a Picard method will typically use a known value in h , but keep the u in $u - T_s$ as unknown: $h(u^-)(u - T_s(t))$. Exercise 10.15 asks you to carry out the details.

10.5.2 Finite difference discretization

A typical diffusion equation

$$u_t = \nabla \cdot (\alpha(u) \nabla u) + f(u),$$

can be discretized by (e.g.) a Backward Euler scheme, which in 2D can be written

$$[D_t^- u = D_x \overline{\alpha(u)}^x D_x u + D_y \overline{\alpha(u)}^y D_y u + f(u)]_{i,j}^n.$$

We do not dive into the details of handling boundary conditions now. Dirichlet and Neumann conditions are handled as in a corresponding linear, variable-coefficient diffusion problems.

Writing the scheme out, putting the unknown values on the left-hand side and known values on the right-hand side, and introducing $\Delta x = \Delta y = h$ to save some writing, one gets

$$\begin{aligned} u_{i,j}^n - \frac{\Delta t}{h^2} & \left(\frac{1}{2}(\alpha(u_{i,j}^n) + \alpha(u_{i+1,j}^n))(u_{i+1,j}^n - u_{i,j}^n) \right. \\ & - \frac{1}{2}(\alpha(u_{i-1,j}^n) + \alpha(u_{i,j}^n))(u_{i,j}^n - u_{i-1,j}^n) \\ & + \frac{1}{2}(\alpha(u_{i,j}^n) + \alpha(u_{i,j+1}^n))(u_{i,j+1}^n - u_{i,j}^n) \\ & \left. - \frac{1}{2}(\alpha(u_{i,j-1}^n) + \alpha(u_{i,j}^n))(u_{i,j}^n - u_{i,j-1}^n) \right) - \Delta t f(u_{i,j}^n) = u_{i,j}^{n-1} \end{aligned}$$

This defines a nonlinear algebraic system on the form $A(u)u = b(u)$.

Picard iteration. The most recently computed values u^- of u^n can be used in α and f for a Picard iteration, or equivalently, we solve $A(u^-)u = b(u^-)$. The result is a linear system of the same type as arising from $u_t = \nabla \cdot (\alpha(\mathbf{x})\nabla u) + f(\mathbf{x}, t)$.

The Picard iteration scheme can also be expressed in operator notation:

$$[D_t^- u = D_x \overline{\alpha(u^-)}^x D_x u + D_y \overline{\alpha(u^-)}^y D_y u + f(u^-)]_{i,j}^n.$$

Newton's method. As always, Newton's method is technically more involved than Picard iteration. We first define the nonlinear algebraic equations to be solved, drop the superscript n (use u for u^n), and introduce $u^{(1)}$ for u^{n-1} :

$$\begin{aligned} F_{i,j} = u_{i,j} - \frac{\Delta t}{h^2} & \left(\frac{1}{2}(\alpha(u_{i,j}) + \alpha(u_{i+1,j}))(u_{i+1,j} - u_{i,j}) - \right. \\ & \frac{1}{2}(\alpha(u_{i-1,j}) + \alpha(u_{i,j}))(u_{i,j} - u_{i-1,j}) + \\ & \frac{1}{2}(\alpha(u_{i,j}) + \alpha(u_{i,j+1}))(u_{i,j+1} - u_{i,j}) - \\ & \left. \frac{1}{2}(\alpha(u_{i,j-1}) + \alpha(u_{i,j}))(u_{i,j} - u_{i,j-1}) - \Delta t f(u_{i,j}) - u_{i,j}^{(1)} \right) = 0. \end{aligned}$$

It is convenient to work with two indices i and j in 2D finite difference discretizations, but it complicates the derivation of the Jacobian, which then gets four indices. (Make sure you really understand the 1D version of this problem as treated in Section 10.4.1.) The left-hand expression of an equation $F_{i,j} = 0$ is to be differentiated with respect to each of the unknowns $u_{r,s}$ (recall that this is short notation for $u_{r,s}^n$), $r \in \mathcal{I}_x$, $s \in \mathcal{I}_y$:

$$J_{i,j,r,s} = \frac{\partial F_{i,j}}{\partial u_{r,s}}.$$

The Newton system to be solved in each iteration can be written as

$$\sum_{r \in \mathcal{I}_x} \sum_{s \in \mathcal{I}_y} J_{i,j,r,s} \delta u_{r,s} = -F_{i,j}, \quad i \in \mathcal{I}_x, j \in \mathcal{I}_y.$$

Given i and j , only a few r and s indices give nonzero contribution to the Jacobian since $F_{i,j}$ contains $u_{i\pm 1,j}$, $u_{i,j\pm 1}$, and $u_{i,j}$. This means that $J_{i,j,r,s}$ has nonzero contributions only if $r = i \pm 1$, $s = j \pm 1$, as well as $r = i$ and $s = j$. The corresponding terms in $J_{i,j,r,s}$ are $J_{i,j,i-1,j}$, $J_{i,j,i+1,j}$, $J_{i,j,i,j-1}$, $J_{i,j,i,j+1}$, and $J_{i,j,i,j}$. Therefore, the left-hand side of the Newton system, $\sum_r \sum_s J_{i,j,r,s} \delta u_{r,s}$ collapses to

$$\begin{aligned} J_{i,j,r,s} \delta u_{r,s} &= J_{i,j,i,j} \delta u_{i,j} + J_{i,j,i-1,j} \delta u_{i-1,j} + J_{i,j,i+1,j} \delta u_{i+1,j} + J_{i,j,i,j-1} \delta u_{i,j-1} \\ &\quad + J_{i,j,i,j+1} \delta u_{i,j+1} \end{aligned}$$

The specific derivatives become

$$\begin{aligned}
J_{i,j,i-1,j} &= \frac{\partial F_{i,j}}{\partial u_{i-1,j}} \\
&= \frac{1}{2} \frac{\Delta t}{h^2} (\alpha'(u_{i-1,j})(u_{i,j} - u_{i-1,j}) + (\alpha(u_{i-1,j}) + \alpha(u_{i,j}))(-1)), \\
J_{i,j,i+1,j} &= \frac{\partial F_{i,j}}{\partial u_{i+1,j}} \\
&= \frac{1}{2} \frac{\Delta t}{h^2} (-\alpha'(u_{i+1,j})(u_{i+1,j} - u_{i,j}) - (\alpha(u_{i-1,j}) + \alpha(u_{i,j}))), \\
J_{i,j,i,j-1} &= \frac{\partial F_{i,j}}{\partial u_{i,j-1}} \\
&= \frac{1}{2} \frac{\Delta t}{h^2} (\alpha'(u_{i,j-1})(u_{i,j} - u_{i,j-1}) + (\alpha(u_{i,j-1}) + \alpha(u_{i,j}))(-1)), \\
J_{i,j,i,j+1} &= \frac{\partial F_{i,j}}{\partial u_{i,j+1}} \\
&= \frac{1}{2} \frac{\Delta t}{h^2} (-\alpha'(u_{i,j+1})(u_{i,j+1} - u_{i,j}) - (\alpha(u_{i,j-1}) + \alpha(u_{i,j}))).
\end{aligned}$$

The $J_{i,j,i,j}$ entry has a few more terms and is left as an exercise. Inserting the most recent approximation u^- for u in the J and F formulas and then forming $J\delta u = -F$ gives the linear system to be solved in each Newton iteration. Boundary conditions will affect the formulas when any of the indices coincide with a boundary value of an index.

10.5.3 Continuation methods

Picard iteration or Newton's method may diverge when solving PDEs with severe nonlinearities. Relaxation with $\omega < 1$ may help, but in highly nonlinear problems it can be necessary to introduce a *continuation parameter* Λ in the problem: $\Lambda = 0$ gives a version of the problem that is easy to solve, while $\Lambda = 1$ is the target problem. The idea is then to increase Λ in steps, $\Lambda_0 = 0, \Lambda_1 < \dots < \Lambda_n = 1$, and use the solution from the problem with Λ_{i-1} as initial guess for the iterations in the problem corresponding to Λ_i .

The continuation method is easiest to understand through an example. Suppose we intend to solve

$$-\nabla \cdot (||\nabla u||^q \nabla u) = f,$$

which is an equation modeling the flow of a non-Newtonian fluid through a channel or pipe. For $q = 0$ we have the Poisson equation (corresponding to

a Newtonian fluid) and the problem is linear. A typical value for pseudo-plastic fluids may be $q_n = -0.8$. We can introduce the continuation parameter $\Lambda \in [0, 1]$ such that $q = q_n \Lambda$. Let $\{\Lambda_\ell\}_{\ell=0}^n$ be the sequence of Λ values in $[0, 1]$, with corresponding q values $\{q_\ell\}_{\ell=0}^n$. We can then solve a sequence of problems

$$-\nabla \cdot \left(||\nabla u^\ell||_\ell^q \nabla u^\ell \right) = f, \quad \ell = 0, \dots, n,$$

where the initial guess for iterating on u^ℓ is the previously computed solution $u^{\ell-1}$. If a particular Λ_ℓ leads to convergence problems, one may try a smaller increase in Λ : $\Lambda_* = \frac{1}{2}(\Lambda_{\ell-1} + \Lambda_\ell)$, and repeat halving the step in Λ until convergence is reestablished.

10.6 Symbolic nonlinear finite element equations

The integrals in nonlinear finite element equations are computed by numerical integration rules in computer programs, so the formulas for the variational form is directly transferred to numbers. It is of interest to understand the nature of the system of difference equations that arises from the finite element method in nonlinear problems and to compare with corresponding expressions arising from finite difference discretization. We shall dive into this problem here. To see the structure of the difference equations implied by the finite element method, we have to find symbolic expressions for the integrals, and this is extremely difficult since the integrals involve the unknown function in nonlinear problems. However, there are some techniques that allow us to approximate the integrals and work out symbolic formulas that can compared with their finite difference counterparts.

We shall address the 1D model problem (10.50) from the beginning of Section 10.4. The finite difference discretization is shown in Section 10.4.1, while the variational form based on Galerkin's method is developed in Section 10.4.3. We build directly on formulas developed in the latter section.

10.6.1 Finite element basis functions

Introduction of finite element basis functions φ_i means setting

$$\psi_i = \varphi_{\nu(i)}, \quad i \in \mathcal{I}_s,$$

where degree of freedom number $\nu(i)$ in the mesh corresponds to unknown number i (c_i). In the present example, we use all the basis functions except the last at $i = N_n - 1$, i.e., $\mathcal{I}_s = \{0, \dots, N_n - 2\}$, and $\nu(j) = j$. The expansion of u can be taken as

$$u = D + \sum_{j \in \mathcal{I}_s} c_j \varphi_{\nu(j)},$$

but it is more common in a finite element context to use a boundary function $B = \sum_{j \in I_b} U_j \varphi_j$, where U_j are prescribed Dirichlet conditions for degree of freedom number j and U_j is the corresponding value.

$$u = D \varphi_{N_n-1} + \sum_{j \in \mathcal{I}_s} c_j \varphi_{\nu(j)}.$$

In the general case with u prescribed as U_j at some nodes $j \in I_b$, we set

$$u = \sum_{j \in I_b} U_j \varphi_j + \sum_{j \in \mathcal{I}_s} c_j \varphi_{\nu(j)},$$

where $c_j = u(x^{\nu(j)})$. That is, $\nu(j)$ maps unknown number j to the corresponding node number $\nu(j)$ such that $c_j = u(x^{\nu(j)})$.

10.6.2 The group finite element method

Finite element approximation of functions of u . Since we already expand u as $\sum_j \varphi_j u(x_j)$, we may use the same approximation for other functions as well. For example,

$$f(u) \approx \sum_j f(x_j) \varphi_j,$$

where $f(x_j)$ is the value of f at node j . Since f is a function of u , $f(x_j) = f(u(x_j))$. Introducing u_j as a short form for $u(x_j)$, we can write

$$f(u) \approx \sum_j f(u_j) \varphi_j.$$

This approximation is known as the *group finite element method* or the *product approximation* technique. The index j runs over all node numbers in the mesh.

The principal advantages of the group finite element method are two-fold:

1. Complicated nonlinear expressions can be simplified to increase the efficiency of numerical computations.
2. One can derive *symbolic forms* of the difference equations arising from the finite element method in nonlinear problems. The symbolic form is useful for comparing finite element and finite difference equations of nonlinear differential equation problems.

Below, we shall explore point 2 to see exactly how the finite element method creates more complex expressions in the resulting linear system (the difference equations) than the finite difference method does. It turns out that it is very difficult to see what kind of terms in the difference equations that arise from $\int f(u)\varphi_i dx$ without using the group finite element method or numerical integration utilizing the nodes only.

Note, however, that an expression like $\int f(u)\varphi_i dx$ causes no problems in a computer program as the integral is calculated by numerical integration using an existing approximation of u in $f(u)$ such that the integrand can be sampled at any spatial point.

Simplified problem. Our aim now is to derive symbolic expressions for the difference equations arising from the finite element method in nonlinear problems and compare the expressions with those arising in the finite difference method. To this end, let us simplify the model problem and set $a = 0$, $\alpha = 1$, $f(u) = u^2$, and have Neumann conditions at both ends such that we get a very simple nonlinear problem $-u'' = u^2$, $u'(0) = 1$, $u'(L) = 0$. The variational form is then

$$\int_0^L u'v' dx = \int_0^L u^2 v dx - v(0), \quad \forall v \in V.$$

The term with $u'v'$ is well known so the only new feature is the term $\int u^2 v dx$.

To make the distance from finite element equations to finite difference equations as short as possible, we shall substitute c_j in the sum $u = \sum_j c_j \varphi_j$ by $u_j = u(x_j)$ since c_j is the value of u at node j . (In the more general case with Dirichlet conditions as well, we have a sum $\sum_j c_j \varphi_{\nu(j)}$ where c_j is replaced by $u(x_{\nu(j)})$. We can then introduce some other counter k such that it is meaningful to write $u = \sum_k u_k \varphi_k$, where k runs over appropriate node numbers.) The quantity u_j in $\sum_j u_j \varphi_j$ is the same

as u at mesh point number j in the finite difference method, which is commonly denoted u_j .

Integrating nonlinear functions. Consider the term $\int u^2 v \, dx$ in the variational formulation with $v = \varphi_i$ and $u = \sum_k \varphi_k u_k$:

$$\int_0^L \left(\sum_k u_k \varphi_k \right)^2 \varphi_i \, dx.$$

Evaluating this integral for P1 elements (see Problem 10.11) results in the expression

$$\frac{h}{12} (u_{i-1}^2 + 2u_i(u_{i-1} + u_{i+1}) + 6u_i^2 + u_{i+1}^2),$$

to be compared with the simple value u_i^2 that would arise in a finite difference discretization when u^2 is sampled at mesh point x_i . More complicated $f(u)$ functions in the integral $\int_0^L f(u) \varphi_i \, dx$ give rise to much more lengthy expressions, if it is possible to carry out the integral symbolically at all.

Application of the group finite element method. Let us use the group finite element method to derive the terms in the difference equation corresponding to $f(u)$ in the differential equation. We have

$$\int_0^L f(u) \varphi_i \, dx \approx \int_0^L \left(\sum_j \varphi_j f(u_j) \right) \varphi_i \, dx = \sum_j \left(\int_0^L \varphi_i \varphi_j \, dx \right) f(u_j).$$

We recognize this expression as the mass matrix M , arising from $\int \varphi_i \varphi_j \, dx$, times the vector $f = (f(u_0), f(u_1), \dots)$: Mf . The associated terms in the difference equations are, for P1 elements,

$$\frac{h}{6} (f(u_{i-1}) + 4f(u_i) + f(u_{i+1})).$$

Occasionally, we want to interpret this expression in terms of finite differences, and to this end a rewrite of this expression is convenient:

$$\frac{h}{6} (f(u_{i-1}) + 4f(u_i) + f(u_{i+1})) = h [f(u) - \frac{h^2}{6} D_x D_x f(u)]_i.$$

That is, the finite element treatment of $f(u)$ (when using a group finite element method) gives the same term as in a finite difference approach,

$f(u_i)$, minus a diffusion term which is the 2nd-order discretization of $\frac{1}{6}h^2 f''(x_i)$.

We may lump the mass matrix through integration with the Trapezoidal rule so that M becomes diagonal in the finite element method. In that case the $f(u)$ term in the differential equation gives rise to a single term $hf(u_i)$, just as in the finite difference method.

10.6.3 Numerical integration of nonlinear terms by hand

Let us reconsider a term $\int f(u)v \, dx$ as treated in the previous section, but now we want to integrate this term numerically. Such an approach can lead to easy-to-interpret formulas if we apply a numerical integration rule that samples the integrand at the node points x_i only, because at such points, $\varphi_j(x_i) = 0$ if $j \neq i$, which leads to great simplifications.

The term in question takes the form

$$\int_0^L f\left(\sum_k u_k \varphi_k\right) \varphi_i \, dx .$$

Evaluation of the integrand at a node x_ℓ leads to a collapse of the sum $\sum_k u_k \varphi_k$ to one term because

$$\sum_k u_k \varphi_k(x_\ell) = u_\ell .$$

$$f\left(\sum_k u_k \underbrace{\varphi_k(x_\ell)}_{\delta_{k\ell}}\right) \underbrace{\varphi_i(x_\ell)}_{\delta_{i\ell}} = f(u_\ell) \delta_{i\ell},$$

where we have used the Kronecker delta: $\delta_{ij} = 0$ if $i \neq j$ and $\delta_{ij} = 1$ if $i = j$.

Considering the Trapezoidal rule for integration, where the integration points are the nodes, we have

$$\int_0^L f\left(\sum_k u_k \varphi_k(x)\right) \varphi_i(x) \, dx \approx h \sum_{\ell=0}^{N_n} f(u_\ell) \delta_{i\ell} - \mathcal{C} = hf(u_i) .$$

This is the same representation of the f term as in the finite difference method. The term \mathcal{C} contains the evaluations of the integrand at the ends with weight $\frac{1}{2}$, needed to make a true Trapezoidal rule:

$$\mathcal{C} = \frac{h}{2} f(u_0)\varphi_i(0) + \frac{h}{2} f(u_{N_n-1})\varphi_i(L).$$

The answer $hf(u_i)$ must therefore be multiplied by $\frac{1}{2}$ if $i = 0$ or $i = N_n - 1$. Note that $\mathcal{C} = 0$ for $i = 1, \dots, N_n - 2$.

One can alternatively use the Trapezoidal rule on the reference cell and assemble the contributions. It is a bit more labor in this context, but working on the reference cell is safer as that approach is guaranteed to handle discontinuous derivatives of finite element functions correctly (not important in this particular example), while the rule above was derived with the assumption that f is continuous at the integration points.

The conclusion is that it suffices to use the Trapezoidal rule if one wants to derive the difference equations in the finite element method and make them similar to those arising in the finite difference method. The Trapezoidal rule has sufficient accuracy for P1 elements, but for P2 elements one should turn to Simpson's rule.

10.6.4 Discretization of a variable coefficient Laplace term

Turning back to the model problem (10.50), it remains to calculate the contribution of the $(\alpha u')'$ and boundary terms to the difference equations. The integral in the variational form corresponding to $(\alpha u')'$ is

$$\int_0^L \alpha \left(\sum_k c_k \psi_k \right) \psi'_i \psi'_j dx.$$

Numerical integration utilizing a value of $\sum_k c_k \psi_k$ from a previous iteration must in general be used to compute the integral. Now our aim is to integrate symbolically, as much as we can, to obtain some insight into how the finite element method approximates this term. To be able to derive symbolic expressions, we must either turn to the group finite element method or numerical integration in the node points. Finite element basis functions φ_i are now used.

Group finite element method. We set $\alpha(u) \approx \sum_k \alpha(u_k) \varphi_k$, and then we write

$$\int_0^L \alpha \left(\sum_k c_k \varphi_k \right) \varphi'_i \varphi'_j dx \approx \sum_k \underbrace{\left(\int_0^L \varphi_k \varphi'_i \varphi'_j dx \right)}_{L_{i,j,k}} \alpha(u_k) = \sum_k L_{i,j,k} \alpha(u_k).$$

Further calculations are now easiest to carry out in the reference cell. With P1 elements we can compute $L_{i,j,k}$ for the two k values that are relevant on the reference cell. Turning to local indices, one gets

$$L_{r,s,t}^{(e)} = \frac{1}{2h} \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}, \quad t = 0, 1,$$

where $r, s, t = 0, 1$ are indices over local degrees of freedom in the reference cell ($i = q(e, r)$, $j = q(e, s)$, and $k = q(e, t)$). The sum $\sum_k L_{i,j,k} \alpha(u_k)$ at the cell level becomes $\sum_{t=0}^1 L_{r,s,t}^{(e)} \alpha(\tilde{u}_t)$, where \tilde{u}_t is $u(x_{q(e,t)})$, i.e., the value of u at local node number t in cell number e . The element matrix becomes

$$\frac{1}{2} (\alpha(\tilde{u}_0) + \alpha(\tilde{u}^{(1)})) \frac{1}{h} \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}. \quad (10.92)$$

As usual, we employ a left-to-right numbering of cells and nodes. Row number i in the global matrix gets contributions from the first row of the element matrix in cell i and the last row of the element matrix in cell $i - 1$. In cell number $i - 1$ the sum $\alpha(\tilde{u}_0) + \alpha(\tilde{u}^{(1)})$ corresponds to $\alpha(u_{i-1}) + \alpha(u_i)$. The same sum becomes $\alpha(u_i) + \alpha(u_{i+1})$ in cell number i . We can with this insight assemble the contributions to row number i in the global matrix:

$$\frac{1}{2h} (-(\alpha(u_{i-1}) + \alpha(u_i)), \quad \alpha(u_{i-1}) + 2\alpha(u_i) + \alpha(u_{i+1}), \quad \alpha(u_i) + \alpha(u_{i+1})).$$

Multiplying by the vector of unknowns u_i results in a formula that can be arranged to

$$-\frac{1}{h} \left(\frac{1}{2} (\alpha(u_i) + \alpha(u_{i+1})) (u_{i+1} - u_i) - \frac{1}{2} (\alpha(u_{i-1}) + \alpha(u_i)) (u_i - u_{i-1}) \right), \quad (10.93)$$

which is nothing but the standard finite difference discretization of $-(\alpha(u)u')'$ with an arithmetic mean of $\alpha(u)$ (and the usual factor h because of the integration in the finite element method).

Numerical integration at the nodes. Instead of using the group finite element method and exact integration we can turn to the Trapezoidal rule for computing $\int_0^L \alpha(\sum_k u_k \varphi_k) \varphi'_i \varphi'_j dx$, again at the cell level since that is most convenient when we deal with discontinuous functions φ'_i :

$$\begin{aligned}
\int_{-1}^1 \alpha \left(\sum_t \tilde{u}_t \tilde{\varphi}_t \right) \tilde{\varphi}'_r \tilde{\varphi}'_s \frac{h}{2} dX &= \int_{-1}^1 \alpha \left(\sum_{t=0}^1 \tilde{u}_t \tilde{\varphi}_t \right) \frac{2}{h} \frac{d\tilde{\varphi}_r}{dX} \frac{2}{h} \frac{d\tilde{\varphi}_s}{dX} \frac{h}{2} dX \\
&= \frac{1}{2h} (-1)^r (-1)^s \int_{-1}^1 \alpha \left(\sum_{t=0}^1 u_t \tilde{\varphi}_t(X) \right) dX \\
&\approx \frac{1}{2h} (-1)^r (-1)^s \alpha \left(\sum_{t=0}^1 \tilde{\varphi}_t(-1) \tilde{u}_t \right) + \alpha \left(\sum_{t=0}^1 \tilde{\varphi}_t(1) \tilde{u}_t \right) \\
&= \frac{1}{2h} (-1)^r (-1)^s (\alpha(\tilde{u}_0) + \alpha(\tilde{u}^{(1)})). \quad (10.94)
\end{aligned}$$

The element matrix in (10.94) is identical to the one in (10.92), showing that the group finite element method and Trapezoidal integration are equivalent with a standard finite discretization of a nonlinear Laplace term $(\alpha(u)u')'$ using an arithmetic mean for α : $[D_x \bar{x} D_x u]_i$.

Remark about integration in the physical x coordinate

We might comment on integration in the physical coordinate system too. The common Trapezoidal rule in Section 10.6.3 cannot be used to integrate derivatives like φ'_i , because the formula is derived under the assumption of a continuous integrand. One must instead use the more basic version of the Trapezoidal rule where all the trapezoids are summed up. This is straightforward, but I think it is even more straightforward to apply the Trapezoidal rule on the reference cell and assemble the contributions.

The term $\int a u v \, dx$ in the variational form is linear and gives these terms in the algebraic equations:

$$\frac{ah}{6} (u_{i-1} + 4u_i + u_{i+1}) = ah [u - \frac{h^2}{6} D_x D_x u]_i.$$

The final term in the variational form is the Neumann condition at the boundary: $Cv(0) = C\varphi_i(0)$. With a left-to-right numbering only $i = 0$ will give a contribution $Cv(0) = C\delta_{i0}$ (since $\varphi_i(0) \neq 0$ only for $i = 0$).

Summary

For the equation

$$-(\alpha(u)u')' + au = f(u),$$

P1 finite elements results in difference equations where

- the term $-(\alpha(u)u')'$ becomes $-h[D_x \overline{\alpha(u)}^x D_x u]_i$ if the group finite element method or Trapezoidal integration is applied,
- $f(u)$ becomes $hf(u_i)$ with Trapezoidal integration or the “mass matrix” representation $h[f(u) - \frac{h}{6}D_x D_x f(u)]_i$ if computed by a group finite element method,
- au leads to the “mass matrix” form $ah[u - \frac{h}{6}D_x D_x u]_i$.

As we now have explicit expressions for the nonlinear difference equations also in the finite element method, a Picard or Newton method can be defined as shown for the finite difference method. However, our efforts in deriving symbolic forms of the difference equations in the finite element method was motivated by a desire to see how nonlinear terms in differential equations make the finite element and difference method different. For practical calculations in computer programs we apply numerical integration, normally the more accurate Gauss-Legendre quadrature rules, to the integrals directly. This allows us to easily *evaluate* the nonlinear algebraic equations for a given numerical approximation of u (here denoted u^-). To *solve* the nonlinear algebraic equations we need to apply the Picard iteration method or Newton’s method to the variational form directly, as shown next.

10.7 Exercises

Problem 10.1: Determine if equations are nonlinear or not

Classify each term in the following equations as linear or nonlinear. Assume that u , \mathbf{u} , and p are unknown functions and that all other symbols are known quantities.

1. $mu'' + \beta|u'|u' + cu = F(t)$
2. $u_t = \alpha u_{xx}$
3. $u_{tt} = c^2 \nabla^2 u$
4. $u_t = \nabla \cdot (\alpha(u) \nabla u) + f(x, y)$
5. $u_t + f(u)_x = 0$
6. $\mathbf{u}_t + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + r \nabla^2 \mathbf{u}$, $\nabla \cdot \mathbf{u} = 0$ (\mathbf{u} is a vector field)

7. $u' = f(u, t)$
 8. $\nabla^2 u = \lambda e^u$

Solution.

1. mu'' is linear; $\beta|u'|u'$ is nonlinear; cu is linear; $F(t)$ does not contain the unknown u and is hence constant in u , so the term is linear.
2. u_t is linear; αu_{xx} is linear.
3. u_{tt} is linear; $c^2 \nabla^2 u$ is linear.
4. u_t is linear; $\nabla \cdot (\alpha(u) \nabla u)$ is nonlinear; $f(x, y)$ is constant in u and hence linear.
5. u_t is linear; $f(u)_x$ is nonlinear if f is nonlinear in u .
6. \mathbf{u}_t is linear; $\mathbf{u} \cdot \nabla \mathbf{u}$ is nonlinear; $-\nabla p$ is linear (in p); $r \nabla^2 \mathbf{u}$ is linear; $\nabla \cdot \mathbf{u}$ is linear.
7. u' is linear; $f(u, t)$ is nonlinear if f is nonlinear in u .
8. $\nabla^2 u$ is linear; λe^u is nonlinear.

Filename: `nonlinear_vs_linear`.

Exercise 10.2: Derive and investigate a generalized logistic model

The logistic model for population growth is derived by assuming a nonlinear growth rate,

$$u' = a(u)u, \quad u(0) = I, \quad (10.95)$$

and the logistic model arises from the simplest possible choice of $a(u)$: $r(u) = \varrho(1 - u/M)$, where M is the maximum value of u that the environment can sustain, and ϱ is the growth under unlimited access to resources (as in the beginning when u is small). The idea is that $a(u) \sim \varrho$ when u is small and that $a(t) \rightarrow 0$ as $u \rightarrow M$.

An $a(u)$ that generalizes the linear choice is the polynomial form

$$a(u) = \varrho(1 - u/M)^p, \quad (10.96)$$

where $p > 0$ is some real number.

a) Formulate a Forward Euler, Backward Euler, and a Crank-Nicolson scheme for (10.95).

Hint. Use a geometric mean approximation in the Crank-Nicolson scheme: $[a(u)u]^{n+1/2} \approx a(u^n)u^{n+1}$.

Solution. The Forward Euler scheme reads

$$[D_t^+ u = a(u)u]^n,$$

or written out,

$$\frac{u^{n+1} - u^n}{\Delta t} = a(u^n)u^n.$$

The scheme is linear in the unknown u^{n+1} :

$$u^{n+1} = u^n + \Delta t a(u^n)u^n.$$

The Backward Euler scheme,

$$[D_t^- u = a(u)u]^n,$$

becomes

$$\frac{u^n - u^{n-1}}{\Delta t} = a(u^n)u^n,$$

which is a nonlinear equation in the unknown u , here expressed as u^{n+1} :

$$u^{n+1} - \Delta t a(u^{n+1})u^{n+1} = u^n.$$

The standard Crank-Nicolson scheme,

$$D_t u = \overline{a(u)u}^t]^{n+\frac{1}{2}},$$

takes the form

$$\frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{2}a(u^n)u^n + \frac{1}{2}a(u^{n+1})u^{n+1}.$$

This is a nonlinear equation in the unknown u^{n+1} ,

$$u^{n+1} - \frac{1}{2}\Delta t a(u^{n+1})u^{n+1} = u^n + \frac{1}{2}\Delta t a(u^n)u^n.$$

However, with the suggested geometric mean, the $a(u)u$ term is linearized:

$$\frac{u^{n+1} - u^n}{\Delta t} = a(u^n)u^{n+1},$$

leading to a linear equation in u^{n+1} :

$$(1 - \Delta t a(u^n))u^{n+1} = u^n.$$

- b)** Formulate Picard and Newton iteration for the Backward Euler scheme in a).

Solution. A Picard iteration for

$$u^{n+1} - \Delta t a(u^{n+1}) u^{n+1} = u^n .$$

applies old values in for u^{n+1} in $a(u^{n+1})$. If u^- is the most recently computed approximation to u^{n+1} , we can write the Picard linearization as

$$(1 - \Delta t a(u^-)) u^{n+1} = u^n .$$

Alternatively, with an iteration index k ,

$$(1 - \Delta t a(u^{n+1,k})) u^{n+1,k+1} = u^n .$$

Newton's method starts with identifying the nonlinear equation as $F(u) = 0$, and here

$$F(u) = u - \Delta t a(u) u - u^n .$$

The Jacobian is

$$J(u) = \frac{dF(u)}{du} = 1 - \Delta t(a'(u)u + a(u)) .$$

The key equation in Newton's method is then

$$J(u^-)\delta u = -F(u^-), \quad u \leftarrow u - \delta u .$$

- c)** Implement the numerical solution methods from a) and b). Use `logistic.py` to compare the case $p = 1$ and the choice (10.96).

Solution. We specialize the code for $a(u)$ to (10.96) since the code was developed from `logistic.py`. It is convenient to work with a dimensionless form of the problem. Choosing a time scale $t_c = 1/\varrho$ and a scale for u , $u_c = M$, leads to

$$u' = \varrho(1-u)^p u, \quad u(0) = \alpha,$$

where α is a dimensionless number

$$\alpha = \frac{I}{M} .$$

The three schemes can be implemented as follows.

```
import numpy as np

def FE_logistic(p, u0, dt, Nt):
    u = np.zeros(Nt+1)
    u[0] = u0
    for n in range(Nt):
        u[n+1] = u[n] + dt*(1 - u[n])**p*u[n]
    return u

def BE_logistic(p, u0, dt, Nt, choice='Picard',
               eps_r=1E-3, omega=1, max_iter=1000):
    # u[n] = u[n-1] + dt*(1-u[n])**p*u[n]
    # -dt*(1-u[n])**p*u[n] + u[n] = u[n-1]
    if choice == 'Picard1':
        choice = 'Picard'
        max_iter = 1

    u = np.zeros(Nt+1)
    iterations = []
    u[0] = u0
    for n in range(1, Nt+1):
        c = -u[n-1]
        if choice == 'Picard':
            def F(u):
                return -dt*(1-u)**p*u + u + c

            u_ = u[n-1]
            k = 0
            while abs(F(u_)) > eps_r and k < max_iter:
                # u*(1-dt*(1-u_)**p) + c = 0
                u_ = omega*(-c/(1-dt*(1-u_)**p)) + (1-omega)*u_
                k += 1
            u[n] = u_
            iterations.append(k)

        elif choice == 'Newton':
            def F(u):
                return -dt*(1-u)**p*u + u + c

            def dF(u):
                return dt*p*(1-u)**(p-1)*u - dt*(1-u)**p + 1

            u_ = u[n-1]
            k = 0
            while abs(F(u_)) > eps_r and k < max_iter:
                u_ = u_ - F(u_)/dF(u_)
                k += 1
            u[n] = u_
            iterations.append(k)
    return u, iterations

def CN_logistic(p, u0, dt, Nt):
    # u[n+1] = u[n] + dt*(1-u[n])**p*u[n+1]
    # (1 - dt*(1-u[n])**p)*u[n+1] = u[n]
```

```

u = np.zeros(Nt+1)
u[0] = u0
for n in range(0, Nt):
    u[n+1] = u[n]/(1 - dt*(1 - u[n])**p)
return u

```

A first verification is to choose $p = 1$ and compare the results with those from `logistic.py`. The number of iterations and the final numerical answers should be identical.

d) Implement unit tests that check the asymptotic limit of the solutions: $u \rightarrow M$ as $t \rightarrow \infty$.

Hint. You need to experiment to find what “infinite time” is (increases substantially with p) and what the appropriate tolerance is for testing the asymptotic limit.

Solution. The test function may look like

```

def test_asymptotic_value():
    T = 100
    dt = 0.1
    Nt = int(round(T/float(dt)))
    u0 = 0.1
    p = 1.8

    u_CN = CN_logistic(p, u0, dt, Nt)
    u_BE_Picard, iter_Picard = BE_logistic(
        p, u0, dt, Nt, choice='Picard',
        eps_r=1E-5, omega=1, max_iter=1000)
    u_BE_Newton, iter_Newton = BE_logistic(
        p, u0, dt, Nt, choice='Newton',
        eps_r=1E-5, omega=1, max_iter=1000)
    u_FE = FE_logistic(p, u0, dt, Nt)

    for arr in u_CN, u_BE_Picard, u_BE_Newton, u_FE:
        expected = 1
        computed = arr[-1]
        tol = 0.01
        msg = 'expected=%s, computed=%s' % (expected, computed)
        print(msg)
        assert abs(expected - computed) < tol

```

It is important with a sufficiently small `eps_r` tolerance for the asymptotic value to be accurate (using `eps_r=1E-3` leads to a value 0.92 at $t = T$ instead of 0.994 when `eps_r=1E-5`).

e) Perform experiments with Newton and Picard iteration for the model (10.96). See how sensitive the number of iterations is to Δt and p .

Solution. Appropriate code is

```

import numpy as np

def FE_logistic(p, u0, dt, Nt):
    u = np.zeros(Nt+1)
    u[0] = u0
    for n in range(Nt):
        u[n+1] = u[n] + dt*(1 - u[n])**p*u[n]
    return u

def BE_logistic(p, u0, dt, Nt, choice='Picard',
               eps_r=1E-3, omega=1, max_iter=1000):
    # u[n] = u[n-1] + dt*(1-u[n])**p*u[n]
    # -dt*(1-u[n])**p*u[n] + u[n] = u[n-1]
    if choice == 'Picard1':
        choice = 'Picard'
        max_iter = 1

    u = np.zeros(Nt+1)
    iterations = []
    u[0] = u0
    for n in range(1, Nt+1):
        c = -u[n-1]
        if choice == 'Picard':
            def F(u):
                return -dt*(1-u)**p*u + u + c

            u_ = u[n-1]
            k = 0
            while abs(F(u_)) > eps_r and k < max_iter:
                # u*(1-dt*(1-u_)**p) + c = 0
                u_ = omega*(-c/(1-dt*(1-u_)**p)) + (1-omega)*u_
                k += 1
            u[n] = u_
            iterations.append(k)

        elif choice == 'Newton':
            def F(u):
                return -dt*(1-u)**p*u + u + c

            def dF(u):
                return dt*p*(1-u)**(p-1)*u - dt*(1-u)**p + 1

            u_ = u[n-1]
            k = 0
            while abs(F(u_)) > eps_r and k < max_iter:
                u_ = u_ - F(u_)/dF(u_)
                k += 1
            u[n] = u_
            iterations.append(k)
    return u, iterations

def CN_logistic(p, u0, dt, Nt):

```

```

# u[n+1] = u[n] + dt*(1-u[n])**p*u[n+1]
# (1 - dt*(1-u[n])**p)*u[n+1] = u[n]
u = np.zeros(Nt+1)
u[0] = u0
for n in range(0, Nt):
    u[n+1] = u[n]/(1 - dt*(1 - u[n])**p)
return u

def test_asymptotic_value():
    T = 100
    dt = 0.1
    Nt = int(round(T/float(dt)))
    u0 = 0.1
    p = 1.8

    u_CN = CN_logistic(p, u0, dt, Nt)
    u_BE_Picard, iter_Picard = BE_logistic(
        p, u0, dt, Nt, choice='Picard',
        eps_r=1E-5, omega=1, max_iter=1000)
    u_BE_Newton, iter_Newton = BE_logistic(
        p, u0, dt, Nt, choice='Newton',
        eps_r=1E-5, omega=1, max_iter=1000)
    u_FE = FE_logistic(p, u0, dt, Nt)

    for arr in u_CN, u_BE_Picard, u_BE_Newton, u_FE:
        expected = 1
        computed = arr[-1]
        tol = 0.01
        msg = 'expected=%s, computed=%s' % (expected, computed)
        print(msg)
        assert abs(expected - computed) < tol

import numpy as np
import matplotlib.pyplot as plt

def demo():
    T = 12
    p = 1.2
    try:
        dt = float(sys.argv[1])
        eps_r = float(sys.argv[2])
        omega = float(sys.argv[3])
    except:
        dt = 0.8
        eps_r = 1E-3
        omega = 1
    N = int(round(T/float(dt)))

    u_FE = FE_logistic(p, 0.1, dt, N)
    u_BE31, iter_BE31 = BE_logistic(p, 0.1, dt, N,
                                     'Picard1', eps_r, omega)
    u_BE3, iter_BE3 = BE_logistic(p, 0.1, dt, N,
                                  'Picard', eps_r, omega)
    u_BE4, iter_BE4 = BE_logistic(p, 0.1, dt, N,

```

```

'Newton', eps_r, omega)
u_CN = CN_logistic(p, 0.1, dt, N)

print('Picard mean no of iterations (dt=%g):' % dt, \
      int(round(np.mean(iter_BE3))))
print('Newton mean no of iterations (dt=%g):' % dt, \
      int(round(np.mean(iter_BE4)))))

t = np.linspace(0, dt*N, N+1)
plt.plot(t, u_FE, t, u_BE3, t, u_BE31, t, u_BE4, t, u_CN)
plt.legend(['FE', 'BE Picard', 'BE Picard1', 'BE Newton', 'CN gm'])
plt.title('dt=%g, eps=%OE' % (dt, eps_r))
plt.xlabel('t')
plt.ylabel('u')
filestem = 'logistic_N%d_eps%03d' % (N, np.log10(eps_r))
plt.savefig(filestem + '_u.png')
plt.savefig(filestem + '_u.pdf')
plt.figure()
plt.plot(list(range(1, len(iter_BE3)+1)), iter_BE3, 'r-o',
          list(range(1, len(iter_BE4)+1)), iter_BE4, 'b-o')
plt.legend(['Picard', 'Newton'])
plt.title('dt=%g, eps=%OE' % (dt, eps_r))
plt.axis([1, N+1, 0, max(iter_BE3 + iter_BE4)+1])
plt.xlabel('Time level')
plt.ylabel('No of iterations')
plt.savefig(filestem + '_iter.png')
plt.savefig(filestem + '_iter.pdf')

```

Filename: logistic_p.

Problem 10.3: Experience the behavior of Newton's method

The program `Newton_demo.py` illustrates graphically each step in Newton's method and is run like

Terminal

Terminal> python Newton_demo.py f dfdx x0 xmin xmax

Use this program to investigate potential problems with Newton's method when solving $e^{-0.5x^2} \cos(\pi x) = 0$. Try a starting point $x_0 = 0.8$ and $x_0 = 0.85$ and watch the different behavior. Just run

Terminal

Terminal> python Newton_demo.py '0.2 + exp(-0.5*x**2)*cos(pi*x)' \
'-x*exp(-x**2)*cos(pi*x) - pi*exp(-x**2)*sin(pi*x)' \
0.85 -3 3

and repeat with 0.85 replaced by 0.8. Zoom in to see the details. The program reads

```

import sys
import matplotlib.pyplot as plt
from numpy import *

from sys import argv
if not len(argv) == 6:
    print("usage: > Newton_demo.py f dfx x0 xmin xmax ")
    sys.exit(0)

f_str = argv[1]
dfdx_str = argv[2]
x0 = float(argv[3])
xmin = float(argv[4])
xmax = float(argv[5])

i = 0
tol = 1.0e-9
maxit = 100
x = x0
f = eval(f_str, vars())
dfdx = eval(dfdx_str, vars())
xs = []
fs = []
xs.append(x)
fs.append(f)
print("x=% .3e   f=% .3e   dfdx=% .3e " % (x, f, dfdx))
while abs(f) > tol and i <= maxit and x > xmin and x < xmax :
    x = x0 - f/dfdx
    f = eval(f_str, vars())
    dfdx = eval(dfdx_str, vars())
    x0 = x
    xs.append(x0)
    fs.append(f)
    i = i+1
    print("x=% .3e   f=% .3e   dfdx=% .3e " % (x, f, dfdx))

x = arange(xmin, xmax, (xmax-xmin)/100.0)
f = eval(f_str, vars())

plt.plot(x, f, "g")
plt.plot(xs, fs, "bo")
plt.plot(xs, fs, "b")
plt.show()

```

Problem 10.4: Compute the Jacobian of a 2×2 system

Write up the system (10.18)-(10.19) in the form $F(u) = 0$, $F = (F_0, F_1)$, $u = (u_0, u_1)$, and compute the Jacobian $J_{i,j} = \partial F_i / \partial u_j$.

Problem 10.5: Solve nonlinear equations arising from a vibration ODE

Consider a nonlinear vibration problem

$$mu'' + bu'|u'| + s(u) = F(t), \quad (10.97)$$

where $m > 0$ is a constant, $b \geq 0$ is a constant, $s(u)$ a possibly nonlinear function of u , and $F(t)$ is a prescribed function. Such models arise from Newton's second law of motion in mechanical vibration problems where $s(u)$ is a spring or restoring force, mu'' is mass times acceleration, and $bu'|u'|$ models water or air drag.

- a)** Rewrite the equation for u as a system of two first-order ODEs, and discretize this system by a Crank-Nicolson (centered difference) method. With $v = u'$, we get a nonlinear term $v^{n+\frac{1}{2}}|v^{n+\frac{1}{2}}|$. Use a geometric average for $v^{n+\frac{1}{2}}$.
- b)** Formulate a Picard iteration method to solve the system of nonlinear algebraic equations.
- c)** Explain how to apply Newton's method to solve the nonlinear equations at each time level. Derive expressions for the Jacobian and the right-hand side in each Newton iteration.

Filename: `nonlin_vib`.

Exercise 10.6: Find the truncation error of arithmetic mean of products

In Section 10.3.4 we introduce alternative arithmetic means of a product. Say the product is $P(t)Q(t)$ evaluated at $t = t_{n+\frac{1}{2}}$. The exact value is

$$[PQ]^{n+\frac{1}{2}} = P^{n+\frac{1}{2}}Q^{n+\frac{1}{2}}$$

There are two obvious candidates for evaluating $[PQ]^{n+\frac{1}{2}}$ as a mean of values of P and Q at t_n and t_{n+1} . Either we can take the arithmetic mean of each factor P and Q ,

$$[PQ]^{n+\frac{1}{2}} \approx \frac{1}{2}(P^n + P^{n+1})\frac{1}{2}(Q^n + Q^{n+1}), \quad (10.98)$$

or we can take the arithmetic mean of the product PQ :

$$[PQ]^{n+\frac{1}{2}} \approx \frac{1}{2}(P^n Q^n + P^{n+1} Q^{n+1}). \quad (10.99)$$

The arithmetic average of $P(t_{n+\frac{1}{2}})$ is $\mathcal{O}(\Delta t^2)$:

$$P(t_{n+\frac{1}{2}}) = \frac{1}{2}(P^n + P^{n+1}) + \mathcal{O}(\Delta t^2).$$

A fundamental question is whether (10.98) and (10.99) have different orders of accuracy in $\Delta t = t_{n+1} - t_n$. To investigate this question, expand quantities at t_{n+1} and t_n in Taylor series around $t_{n+\frac{1}{2}}$, and subtract the true value $[PQ]^{n+\frac{1}{2}}$ from the approximations (10.98) and (10.99) to see what the order of the error terms are.

Hint. You may explore `sympy` for carrying out the tedious calculations. A general Taylor series expansion of $P(t + \frac{1}{2}\Delta t)$ around t involving just a general function $P(t)$ can be created as follows:

```
>>> from sympy import *
>>> t, dt = symbols('t dt')
>>> P = symbols('P', cls=Function)
>>> P(t).series(t, 0, 4)
P(0) + t*Subs(Derivative(P(_x), _x), (_x,), (0,)) +
t**2*Subs(Derivative(P(_x), _x, _x), (_x,), (0,))/2 +
t**3*Subs(Derivative(P(_x), _x, _x, _x), (_x,), (0,))/6 + O(t**4)
>>> P_p = P(t).series(t, 0, 4).subs(t, dt/2)
>>> P_p
P(0) + dt*Subs(Derivative(P(_x), _x), (_x,), (0,))/2 +
dt**2*Subs(Derivative(P(_x), _x, _x), (_x,), (0,))/8 +
dt**3*Subs(Derivative(P(_x), _x, _x, _x), (_x,), (0,))/48 + O(dt**4)
```

The error of the arithmetic mean, $\frac{1}{2}(P(-\frac{1}{2}\Delta t) + P(-\frac{1}{2}\Delta t))$ for $t = 0$ is then

```
>>> P_m = P(t).series(t, 0, 4).subs(t, -dt/2)
>>> mean = Rational(1,2)*(P_m + P_p)
>>> error = simplify(expand(mean) - P(0))
>>> error
dt**2*Subs(Derivative(P(_x), _x, _x), (_x,), (0,))/8 + O(dt**4)
```

Use these examples to investigate the error of (10.98) and (10.99) for $n = 0$. (Choosing $n = 0$ is necessary for not making the expressions too complicated for `sympy`, but there is of course no lack of generality by using $n = 0$ rather than an arbitrary n - the main point is the product and addition of Taylor series.)

Filename: `product_arith_mean`.

Problem 10.7: Newton's method for linear problems

Suppose we have a linear system $F(u) = Au - b = 0$. Apply Newton's method to this system, and show that the method converges in one iteration. Filename: `Newton_linear`.

Exercise 10.8: Discretize a 1D problem with a nonlinear coefficient

We consider the problem

$$((1 + u^2)u')' = 1, \quad x \in (0, 1), \quad u(0) = u(1) = 0. \quad (10.100)$$

- a) Discretize (10.100) by a centered finite difference method on a uniform mesh.
 - b) Discretize (10.100) by a finite element method with P1 elements of equal length. Use the Trapezoidal method to compute all integrals. Set up the resulting matrix system in symbolic form such that the equations can be compared with those in a).
- Filename: `nonlin_1D_coeff_discretize`.

Exercise 10.9: Linearize a 1D problem with a nonlinear coefficient

We have a two-point boundary value problem

$$((1 + u^2)u')' = 1, \quad x \in (0, 1), \quad u(0) = u(1) = 0. \quad (10.101)$$

- a) Construct a Picard iteration method for (10.101) without discretizing in space.
 - b) Apply Newton's method to (10.101) without discretizing in space.
 - c) Discretize (10.101) by a centered finite difference scheme. Construct a Picard method for the resulting system of nonlinear algebraic equations.
 - d) Discretize (10.101) by a centered finite difference scheme. Define the system of nonlinear algebraic equations, calculate the Jacobian, and set up Newton's method for solving the system.
- Filename: `nonlin_1D_coeff_linearize`.

Problem 10.10: Finite differences for the 1D Bratu problem

We address the so-called Bratu problem

$$u'' + \lambda e^u = 0, \quad x \in (0, 1), \quad u(0) = u(1) = 0, \quad (10.102)$$

where λ is a given parameter and u is a function of x . This is a widely used model problem for studying numerical methods for nonlinear differential equations. The problem (10.102) has an exact solution

$$u_e(x) = -2 \ln \left(\frac{\cosh((x - \frac{1}{2})\theta/2)}{\cosh(\theta/4)} \right),$$

where θ solves

$$\theta = \sqrt{2\lambda} \cosh(\theta/4).$$

There are two solutions of (10.102) for $0 < \lambda < \lambda_c$ and no solution for $\lambda > \lambda_c$. For $\lambda = \lambda_c$ there is one unique solution. The critical value λ_c solves

$$1 = \sqrt{2\lambda_c} \frac{1}{4} \sinh(\theta(\lambda_c)/4).$$

A numerical value is $\lambda_c = 3.513830719$.

- a)** Discretize (10.102) by a centered finite difference method.
- b)** Set up the nonlinear equations $F_i(u_0, u_1, \dots, u_{N_x}) = 0$ from a). Calculate the associated Jacobian.
- c)** Implement a solver that can compute $u(x)$ using Newton's method. Plot the error as a function of x in each iteration.
- d)** Investigate whether Newton's method gives second-order convergence by computing $\|u_e - u\|/\|u_e - u^-\|^2$ in each iteration, where u is solution in the current iteration and u^- is the solution in the previous iteration. Filename: `nonlin_1D_Bratu_fd`.

Problem 10.11: Integrate functions of finite element expansions

We shall investigate integrals on the form

$$\int_0^L f\left(\sum_k u_k \varphi_k(x)\right) \varphi_i(x) dx, \quad (10.103)$$

where $\varphi_i(x)$ are P1 finite element basis functions and u_k are unknown coefficients, more precisely the values of the unknown function u at nodes x_k . We introduce a node numbering that goes from left to right and also that all cells have the same length h . Given i , the integral only gets contributions from $[x_{i-1}, x_{i+1}]$. On this interval $\varphi_k(x) = 0$ for $k < i - 1$ and $k > i + 1$, so only three basis functions will contribute:

$$\sum_k u_k \varphi_k(x) = u_{i-1} \varphi_{i-1}(x) + u_i \varphi_i(x) + u_{i+1} \varphi_{i+1}(x).$$

The integral (10.103) now takes the simplified form

$$\int_{x_{i-1}}^{x_{i+1}} f(u_{i-1} \varphi_{i-1}(x) + u_i \varphi_i(x) + u_{i+1} \varphi_{i+1}(x)) \varphi_i(x) dx.$$

Split this integral in two integrals over cell L (left), $[x_{i-1}, x_i]$, and cell R (right), $[x_i, x_{i+1}]$. Over cell L, u simplifies to $u_{i-1} \varphi_{i-1} + u_i \varphi_i$ (since $\varphi_{i+1} = 0$ on this cell), and over cell R, u simplifies to $u_i \varphi_i + u_{i+1} \varphi_{i+1}$. Make a `sympy` program that can compute the integral and write it out as a difference equation. Give the $f(u)$ formula on the command line. Try out $f(u) = u^2, \sin u, \exp u$.

Hint. Introduce symbols `u_i`, `u_im1`, and `u_ip1` for u_i , u_{i-1} , and u_{i+1} , respectively, and similar symbols for x_i , x_{i-1} , and x_{i+1} . Find formulas for the basis functions on each of the two cells, make expressions for u on the two cells, integrate over each cell, expand the answer and simplify. You can ask `sympy` for `LATEX` code and render it either by creating a `LATEX` document and compiling it to a PDF document or by using <http://latex.codecogs.com> to display `LATEX` formulas in a web page. Here are some appropriate Python statements for the latter purpose:

```
from sympy import *
...
# expr_i holds the integral as a sympy expression
latex_code = latex(expr_i, mode='plain')
# Replace u_im1 sympy symbol name by latex symbol u_{i-1}
latex_code = latex_code.replace('im1', '{i-1}')
# Replace u_ip1 sympy symbol name by latex symbol u_{i+1}
latex_code = latex_code.replace('ip1', '{i+1}')
# Escape (quote) latex_code so it can be sent as HTML text
import cgi
html_code = cgi.escape(latex_code)
# Make a file with HTML code for displaying the LaTeX formula
f = open('tmp.html', 'w')
# Include an image that can be clicked on to yield a new
# page with an interactive editor and display area where the
# formula can be further edited
```

```

text = """
<a href="http://www.codecogs.com/eqnedit.php?latex=%(html_code)s"
    target="_blank">

</a>
"""
% vars()
f.write(text)
f.close()

```

The formula is displayed by loading `tmp.html` into a web browser.
 Filename: `fu_fem_int`.

Problem 10.12: Finite elements for the 1D Bratu problem

We address the same 1D Bratu problem as described in Problem 10.10.

- a) Discretize (10.12) by a finite element method using a uniform mesh with P1 elements. Use a group finite element method for the e^u term.
 - b) Set up the nonlinear equations $F_i(u_0, u_1, \dots, u_{N_x}) = 0$ from a). Calculate the associated Jacobian.
- Filename: `nonlin_1D_Bratu_fe`.

Exercise 10.13: Discretize a nonlinear 1D heat conduction PDE by finite differences

We address the 1D heat conduction PDE

$$\varrho c(T)T_t = (k(T)T_x)_x,$$

for $x \in [0, L]$, where ϱ is the density of the solid material, $c(T)$ is the heat capacity, T is the temperature, and $k(T)$ is the heat conduction coefficient. $T(x, 0) = I(x)$, and ends are subject to a cooling law:

$$k(T)T_x|_{x=0} = h(T)(T - T_s), \quad -k(T)T_x|_{x=L} = h(T)(T - T_s),$$

where $h(T)$ is a heat transfer coefficient and T_s is the given surrounding temperature.

- a) Discretize this PDE in time using either a Backward Euler or Crank-Nicolson scheme.

- b)** Formulate a Picard iteration method for the time-discrete problem (i.e., an iteration method before discretizing in space).
- c)** Formulate a Newton method for the time-discrete problem in b).
- d)** Discretize the PDE by a finite difference method in space. Derive the matrix and right-hand side of a Picard iteration method applied to the space-time discretized PDE.
- e)** Derive the matrix and right-hand side of a Newton method applied to the discretized PDE in d).

Filename: `nonlin_1D_heat_FD`.

Exercise 10.14: Use different symbols for different approximations of the solution

The symbol u has several meanings, depending on the context, as briefly mentioned in Section 10.5.1. Go through the derivation of the Picard iteration method in that section and use different symbols for all the different approximations of u :

- $u_e(\mathbf{x}, t)$ for the exact solution of the PDE problem
- $u_e(\mathbf{x})^n$ for the exact solution after time discretization
- $u^n(\mathbf{x})$ for the spatially discrete solution $\sum_j c_j \psi_j$
- $u^{n,k}$ for approximation in Picard/Newton iteration no k to $u^n(\mathbf{x})$

Filename: `nonlin_heat_FE_usymbols`.

Exercise 10.15: Derive Picard and Newton systems from a variational form

We study the multi-dimensional heat conduction PDE

$$\varrho c(T) T_t = \nabla \cdot (k(T) \nabla T)$$

in a spatial domain Ω , with a nonlinear Robin boundary condition

$$-k(T) \frac{\partial T}{\partial n} = h(T)(T - T_s(t)),$$

at the boundary $\partial\Omega$. The primary unknown is the temperature T , ϱ is the density of the solid material, $c(T)$ is the heat capacity, $k(T)$ is the heat conduction, $h(T)$ is a heat transfer coefficient, and $T_s(T)$ is a possibly time-dependent temperature of the surroundings.

- a)** Use a Backward Euler or Crank-Nicolson time discretization and derive the variational form for the spatial problem to be solved at each time level.
- b)** Define a Picard iteration method from the variational form at a time level.
- c)** Derive expressions for the matrix and the right-hand side of the equation system that arises from applying Newton's method to the variational form at a time level.
- d)** Apply the Backward Euler or Crank-Nicolson scheme in time first. Derive a Newton method at the PDE level. Make a variational form of the resulting PDE at a time level.

Filename: `nonlin_heat_FE`.

Exercise 10.16: Derive algebraic equations for nonlinear 1D heat conduction

We consider the same problem as in Exercise 10.15, but restricted to one space dimension: $\Omega = [0, L]$. Simplify the boundary condition to $T_x = 0$ (i.e., $h(T) = 0$). Use a uniform finite element mesh of P1 elements, the group finite element method, and the Trapezoidal rule for integration at the nodes to derive symbolic expressions for the algebraic equations arising from this diffusion problem. Filename: `nonlin_1D_heat_FE`.

Exercise 10.17: Differentiate a highly nonlinear term

The operator $\nabla \cdot (\alpha(u) \nabla u)$ with $\alpha(u) = |\nabla u|^q$ appears in several physical problems, especially flow of Non-Newtonian fluids. The expression $|\nabla u|$ is defined as the Euclidean norm of a vector: $|\nabla u|^2 = \nabla u \cdot \nabla u$. In a Newton method one has to carry out the differentiation $\partial \alpha(u)/\partial c_j$, for $u = \sum_k c_k \psi_k$. Show that

$$\frac{\partial}{\partial u_j} |\nabla u|^q = q |\nabla u|^{q-2} \nabla u \cdot \nabla \psi_j .$$

Solution.

$$\begin{aligned}
\frac{\partial}{\partial c_j} |\nabla u|^q &= \frac{\partial}{\partial c_j} (\nabla u \cdot \nabla u)^{\frac{q}{2}} = \frac{q}{2} (\nabla u \cdot \nabla u)^{\frac{q}{2}-1} \frac{\partial}{\partial c_j} (\nabla u \cdot \nabla u) \\
&= \frac{q}{2} |\nabla u|^{q-2} \left(\frac{\partial}{\partial c_j} (\nabla u) \cdot \nabla u + \nabla u \cdot \frac{\partial}{\partial c_j} (\nabla u) \right) \\
&= q |\nabla u|^{q-2} (\nabla u \cdot \nabla \frac{\partial u}{\partial c_j}) = q |\nabla u|^{q-2} (\nabla u \cdot \nabla \psi_j)
\end{aligned}$$

Filename: `nonlin_differentiate.`

Exercise 10.18: Crank-Nicolson for a nonlinear 3D diffusion equation

Redo Section 10.5.2 when a Crank-Nicolson scheme is used to discretize the equations in time and the problem is formulated for three spatial dimensions.

Hint. Express the Jacobian as $J_{i,j,k,r,s,t} = \partial F_{i,j,k} / \partial u_{r,s,t}$ and observe, as in the 2D case, that $J_{i,j,k,r,s,t}$ is very sparse: $J_{i,j,k,r,s,t} \neq 0$ only for $r = i \pm 1$, $s = j \pm 1$, and $t = k \pm 1$ as well as $r = i$, $s = j$, and $t = k$.

Filename: `nonlin_heat_FD_CN_2D.`

Exercise 10.19: Find the sparsity of the Jacobian

Consider a typical nonlinear Laplace term like $\nabla \cdot \alpha(u) \nabla u$ discretized by centered finite differences. Explain why the Jacobian corresponding to this term has the same sparsity pattern as the matrix associated with the corresponding linear term $\alpha \nabla^2 u$.

Hint. Set up the unknowns that enter the difference equation at a point (i, j) in 2D or (i, j, k) in 3D, and identify the nonzero entries of the Jacobian that can arise from such a type of difference equation.

Filename: `nonlin_sparsity_Jacobian.`

Problem 10.20: Investigate a 1D problem with a continuation method

Flow of a pseudo-plastic power-law fluid between two flat plates can be modeled by

$$\frac{d}{dx} \left(\mu_0 \left| \frac{du}{dx} \right|^{n-1} \frac{du}{dx} \right) = -\beta, \quad u'(0) = 0, \quad u(H) = 0,$$

where $\beta > 0$ and $\mu_0 > 0$ are constants. A target value of n may be $n = 0.2$.

- a)** Formulate a Picard iteration method directly for the differential equation problem.
- b)** Perform a finite difference discretization of the problem in each Picard iteration. Implement a solver that can compute u on a mesh. Verify that the solver gives an exact solution for $n = 1$ on a uniform mesh regardless of the cell size.
- c)** Given a sequence of decreasing n values, solve the problem for each n using the solution for the previous n as initial guess for the Picard iteration. This is called a continuation method. Experiment with $n = (1, 0.6, 0.2)$ and $n = (1, 0.9, 0.8, \dots, 0.2)$ and make a table of the number of Picard iterations versus n .
- d)** Derive a Newton method at the differential equation level and discretize the resulting linear equations in each Newton iteration with the finite difference method.
- e)** Investigate if Newton's method has better convergence properties than Picard iteration, both in combination with a continuation method.

A successful family of methods, usually referred to as Conjugate Gradient-like algorithms, or Krylov subspace methods, can be viewed as Galerkin or least-squares methods applied to a linear system $Ax = b$. This view is different from the standard approaches to deriving the classical Conjugate Gradient method in the literature. Nevertheless, the fundamental ideas of least squares and Galerkin approximations from Section 3 can be used to derive the most popular and successful methods for linear systems, and this is the topic of the present chapter. Such a view may increase the general understanding of variational methods and their applicability.

Our exposition focuses on the basic reasoning behind the methods, and a natural continuation of the material here is provided by several review texts. Bruaset [8] gives an accessible theoretical overview of a wide range of Conjugate Gradient-like methods. Barrett et al. [4] present a collection of computational algorithms and give valuable information about the practical use of the methods. Saad [27] and Axelsson [3] have evolved as modern, classical text books on iterative methods in general for linear systems.

Given a linear system

$$Ax = b, \quad x, b \in \mathbb{R}^n, \quad A \in \mathbb{R}^{n,n} \tag{11.1}$$

and a start vector x^0 , we want to construct an iterative solution method that produces approximations x^1, x^2, \dots , which hopefully converge to the exact solution x . In iteration no. k we seek an approximation

$$x^{k+1} = x^k + u, \quad u = \sum_{j=0}^k c_j q_j \quad (11.2)$$

where $q_j \in \mathbb{R}^n$ are known vectors and c_j are constants to be determined. To be specific, let q_0, \dots, q_k be basis vectors for V_{k+1} :

$$V_{k+1} = \text{span}\{q_0, \dots, q_k\}.$$

The associated inner product (\cdot, \cdot) is here the standard Euclidean inner product on \mathbb{R}^n .

The corresponding error in the equation $Ax = b$, the residual, becomes

$$r^{k+1} = b - Ax^{k+1} = r^k - \sum_{j=0}^k c_j Aq_j.$$

11.1 Conjugate gradient-like iterative methods

11.1.1 The Galerkin method

Galerkin's method states that the error in the equation, the residual, is orthogonal to the space V_{k+1} where we seek the approximation to the problem.

The Galerkin (or projection) method aims at finding $u = \sum_j c_j q_j \in V_{k+1}$ such that

$$(r^{k+1}, v) = 0, \quad \forall v \in V_{k+1}.$$

This statement is equivalent to the residual being orthogonal to each basis vector:

$$(r^{k+1}, q_i) = 0, \quad i = 0, \dots, k. \quad (11.3)$$

Inserting the expression for r^{k+1} in (11.3) gives a linear system for c_j :

$$\sum_{j=0}^k (Aq_i, q_j) c_j = (r^k, q_i), \quad i = 0, \dots, k. \quad (11.4)$$

11.1.2 The least squares method

The idea of the least-squares method is to minimize the square of the norm of the residual with respect to the free parameters c_0, \dots, c_k . That is, we minimize (r^{k+1}, r^{k+1}) :

$$\frac{\partial}{\partial c_i} (r^{k+1}, r^{k+1}) = 2 \left(\frac{\partial r^{k+1}}{\partial c_i}, r^{k+1} \right) = 0, \quad i = 0, \dots, k.$$

Since $\partial r^{k+1} / \partial c_i = -Aq_i$, this approach leads to the following linear system:

$$\sum_{j=0}^k (Aq_i, Aq_j) c_j = (r^{k+1}, Aq_i), \quad i = 0, \dots, k. \quad (11.5)$$

11.1.3 Krylov subspaces

To obtain a complete algorithm, we need to establish a rule to update the basis $\mathcal{B} = \{q_0, \dots, q_k\}$ for the next iteration. That is, we need to compute a new basis vector $q_{k+1} \in V_{k+2}$ such that

$$\mathcal{B} = \{q_0, \dots, q_{k+1}\} \quad (11.6)$$

is a basis for the space V_{k+2} that is used in the next iteration. The present family of methods applies the *Krylov space*, where V_k is defined as

$$V_k = \text{span} \left\{ r^0, Ar^0, A^2r^0, \dots, A^{k-1}r^0 \right\}. \quad (11.7)$$

Some frequent names of the associated iterative methods are therefore Krylov subspace iterations, Krylov projection methods, or simply Krylov methods. It is a fact that $V_k \subset V_{k+1}$ and that $r^0, Ar^0, \dots, A^{k-1}r^0$ are linearly independent vectors.

11.1.4 Computation of the basis vectors

A potential formula for updating q_{k+1} , such that $q_{k+1} \in V_{k+2}$, is

$$q_{k+1} = r^{k+1} + \sum_{j=0}^k \beta_j q_j. \quad (11.8)$$

(Since r^{k+1} involves Aq_k , and $q_k \in V_{k+1}$, multiplying by A raises the dimension of the Krylov space by 1, so $Aq_k \in V_{k+2}$.) The free parameters β_j can be used to enforce desirable orthogonality properties of q_0, \dots, q_{k+1} . For example, it is convenient to require that the coefficient matrices in the linear systems for c_0, \dots, c_k are diagonal. Otherwise, we must solve a $(k+1) \times (k+1)$ linear system in each iteration. If k should approach n , the systems for the coefficients c_i are of the same size as our original system $Ax = b$! A diagonal matrix, however, ensures an efficient closed form solution for c_0, \dots, c_k .

To obtain a diagonal coefficient matrix, we require in Galerkin's method that

$$(Aq_i, q_j) = 0 \quad \text{when } i \neq j,$$

whereas we in the least-squares method require

$$(Aq_i, Aq_j) = 0 \quad \text{when } i \neq j.$$

We can define the inner product

$$\langle u, v \rangle \equiv (Au, v) = u^T Av, \quad (11.9)$$

provided A is symmetric and positive definite. Another useful inner product is

$$[u, v] \equiv (Au, Av) = u^T A^T Av. \quad (11.10)$$

These inner products will be referred to as the A product, with the associated A norm, and the $A^T A$ product, with the associated $A^T A$ norm.

The orthogonality condition on the q_i vectors are then $\langle q_{k+1}, q_i \rangle = 0$ in the Galerkin method and $[q_{k+1}, q_i] = 0$ in the least-squares method, where i runs from 0 to k . A standard Gram-Schmidt process can be used for constructing q_{k+1} orthogonal to q_0, \dots, q_k . This leads to the determination of the β_0, \dots, β_k constants as

$$\beta_i = \frac{\langle r^{k+1}, q_i \rangle}{\langle q_i, q_i \rangle} \quad (\text{Galerkin}) \quad (11.11)$$

$$\beta_i = \frac{[r^{k+1}, q_i]}{[q_i, q_i]} \quad (\text{least squares}) \quad (11.12)$$

for $i = 0, \dots, k$.

11.1.5 Computation of a new solution vector

The orthogonality condition on the basis vectors q_i leads to the following solution for c_0, \dots, c_k :

$$c_i = \frac{(r^k, q_i)}{\langle q_i, q_i \rangle} \quad (\text{Galerkin}) \quad (11.13)$$

$$c_i = \frac{(r^k, Aq_i)}{[q_i, q_i]} \quad (\text{least squares}) \quad (11.14)$$

In iteration k , $(r^k, q_i) = 0$ and $(r^k, Aq_i) = 0$, for $i = 0, \dots, k - 1$, in the Galerkin and least squares case, respectively. Hence, $c_i = 0$, for $i = 0, \dots, k - 1$. In other words,

$$x^{k+1} = x^k + c_k q_k.$$

When A is symmetric and positive definite, one can show that also $\beta_i = 0$, for $i = 1, \dots, k - 1$, in both the Galerkin and least squares methods [8]. This means that x^k and q_{k+1} can be updated using only q_k and not the previous q_0, \dots, q_{k-1} vectors. This property has of course dramatic effects on the storage requirements of the algorithms as the number of iterations increases.

For the suggested algorithms to work, we must require that the denominators in (11.13) and (11.14) do not vanish. This is always fulfilled for the least-squares method, while a (positive or negative) definite matrix A avoids break-down of the Galerkin-based iteration (provided $q_i \neq 0$).

The Galerkin solution method for linear systems was originally devised as a *direct method* in the 1950s. After n iterations the exact solution is found in exact arithmetic, but at a higher cost than when using Gaussian elimination. Naturally, the method did not receive significant popularity before researchers discovered (in the beginning of the 1970s) that the method could produce a good approximation to x for $k \ll n$ iterations. The method, called the Conjugate Gradient method, has from then on caught considerable interest as an iterative scheme for solving linear systems arising from PDEs discretized such that the coefficient matrix becomes sparse.

Finally, we mention how to terminate the iteration. The simplest criterion is $\|r^{k+1}\| \leq \epsilon_r$, where ϵ_r is a small prescribed tolerance. Sometimes it is more appropriate to use a relative residual, $\|r^{k+1}\|/\|r^0\| \leq \epsilon_r$.

Termination criteria for Conjugate Gradient-like methods is a subject on its own [8].

11.1.6 Summary of the least squares method

In the algorithm below, we have summarized the computational steps in the least-squares method. Notice that we update the residual recursively instead of using $r^k = b - Ax^k$ in each iteration since we then avoid a possibly expensive matrix-vector product.

1. given a start vector x^0 , compute $r^0 = b - Ax^0$ and set $q_0 = r^0$.
2. for $k = 0, 1, 2, \dots$ until termination criteria are fulfilled:
 - a. $c_k = (r^k, Aq_k)/[q_k, q_k]$
 - b. $x^{k+1} = x^k + c_k q_k$
 - c. $r^{k+1} = r^k - c_k Aq_k$
 - d. if A is symmetric then
 - i. $\beta_k = [r^{k+1}, q_k]/[q_k, q_k]$
 - A. $q_{k+1} = r^{k+1} - \beta_k q_k$
 - e. else
 - i. $\beta_j = [r^{k+1}, q_j]/[q_j, q_j], \quad j = 0, \dots, k$
 - ii. $q_{k+1} = r^{k+1} - \sum_{j=0}^k \beta_j q_j$

Remark. The algorithm above is just a summary of the steps in the derivation of the least-squares method and should not be directly used for practical computations without further developments.

11.1.7 Truncation and restart

When A is nonsymmetric, the storage requirements of q_0, \dots, q_k may be prohibitively large. It has become a standard trick to either *truncate* or *restart* the algorithm. In the latter case one restarts the algorithm every $K+1$ -th step, i.e., one aborts the iteration and starts the algorithm again with $x^0 = x^K$. The other alternative is to truncate the sum $\sum_{j=0}^k \beta_j q_j$ and use only the last K vectors:

$$x^{k+1} = x^k + \sum_{j=k-K}^k \beta_j q_j .$$

Both the restarted and truncated version of the algorithm require storage of only $K + 1$ basis vectors q_{k-K}, \dots, q_k . The basis vectors are also often called *search direction vectors*. The truncated version of the least-squares algorithm above is widely known as *Generalized Minimum Residuals*, shortened as GMRES, or GMRES(K) to explicitly indicate the number of search direction vectors. In the literature one encounters the name *Generalized Conjugate Residual method*, abbreviated CGR, for the restarted version of Orthomin. When A is symmetric, the method is known under the name *Conjugate Residuals*.

11.1.8 Summary of the Galerkin method

In case of Galerkin's method, we assume that A is symmetric and positive definite. The resulting computational procedure is the famous Conjugate Gradient method. Since A must be symmetric, the recursive update of q_{k+1} needs only one previous search direction vector q_k , that is, $\beta_j = 0$ for $j < k$.

1. Given a start vector x^0 , compute $r^0 = b - Ax^0$ and set $q_0 = r^0$.
2. for $k = 1, 2, \dots$ until termination criteria are fulfilled:
 - a. $c_k = (r^k, q_k) / \langle q_k, q_k \rangle$
 - b. $x^k = x^{k-1} + c_k q_k$
 - c. $r^k = r^{k-1} - c_k A q_k$
 - d. $\beta_k = \langle r^{k+1}, q_k \rangle / \langle q_k, q_k \rangle$
 - e. $q_{k+1} = r^{k+1} - \beta_k q_k$

The previous remark that the listed algorithm is just a summary of the steps in the solution procedure, and not an efficient algorithm that should be implemented in its present form, must be repeated here. In general, we recommend to rely on some high-quality linear algebra library that offers an implementation of the Conjugate gradient method.

The computational nature of Conjugate gradient-like methods

Looking at the Galerkin and least squares algorithms above, one notice that the matrix A is only used in matrix-vector products. This means that it is sufficient to store only the nonzero entries of A . The rest of the algorithms consists of vector operations of the

type $y \leftarrow ax + y$, the slightly more general variant $q \leftarrow ax + y$, as well as inner products.

11.1.9 A framework based on the error

Let us define the error $e^k = x - x^k$. Multiplying this equation by A leads to the well-known relation between the error and the residual for linear systems:

$$Ae^k = r^k. \quad (11.15)$$

Using $r^k = Ae^k$ we can reformulate the Galerkin and least-squares methods in terms of the error. The Galerkin method can then be written

$$(r^k, q_i) = (Ae^k, q_i) = \langle e^k, q_i \rangle = 0, \quad i = 0, \dots, k. \quad (11.16)$$

For the least-squares method we obtain

$$(r^k, Aq_i) = [e^k, q_i] = 0, \quad i = 0, \dots, k. \quad (11.17)$$

This means that

$$\langle e^k, v \rangle = 0 \quad \forall v \in V_{k+1} \text{ (Galerkin)}$$

$$[e^k, v] = 0 \quad \forall v \in V_{k+1} \text{ (least-squares)}$$

In other words, the error is A -orthogonal to the space V_{k+1} in the Galerkin method, whereas the error is $A^T A$ -orthogonal to V_{k+1} in the least-squares method. When the error is orthogonal to a space, we find the best approximation in the associated norm to the solution in that space. Specifically here, it means that for a symmetric and positive definite A , the Conjugate gradient method finds the optimal adjustment in V_{k+1} of the vector x^k (in the A -norm) in the update for x^{k+1} . Similarly, the least squares formulation finds the optimal adjustment in V_{k+1} measured in the $A^T A$ -norm.

A lot of Conjugate gradient-like methods were developed in the 1980s and 1990s, some of the most popular methods do not fit directly into the framework presented here. The theoretical similarities between the methods are covered in [8], whereas we refer to [4] for algorithms and practical comments related to widespread methods, such as the SYMMLQ

method (for symmetric indefinite systems), the Generalized Minimal Residual (GMRES) method, the BiConjugate Gradient (BiCG) method, the Quasi-Minimal Residual (QMR) method, and the BiConjugate Gradient Stabilized (BiCGStab) method. When A is symmetric and positive definite, the Conjugate gradient method is the optimal choice with respect to computational efficiency, but when A is nonsymmetric, the performance of the methods is strongly problem dependent, and one needs to experiment.

11.2 Preconditioning

11.2.1 Motivation and Basic Principles

The Conjugate Gradient method has been subject to extensive analysis, and its convergence properties are well understood. To reduce the initial error $e^0 = x - x^0$ with a factor $0 < \epsilon \ll 1$ after k iterations, or more precisely, $\|e^k\|_A \leq \epsilon \|e^0\|_A$, it can be shown that k is bounded by

$$\frac{1}{2} \ln \frac{2}{\epsilon} \sqrt{\kappa},$$

where κ is the ratio of the largest and smallest eigenvalue of A . The quantity κ is commonly referred to as the spectral *condition number*. Common finite element and finite difference discretizations of Poisson-like PDEs lead to $\kappa \sim h^{-2}$, where h denotes the mesh size. This implies that the Conjugate Gradient method converges slowly in PDE problems with fine meshes, as the number of iterations is proportional to h^{-1} .

To speed up the Conjugate Gradient method, we should manipulate the eigenvalue distribution. For instance, we could reduce the condition number κ . This can be achieved by so-called *preconditioning*. Instead of applying the iterative method to the system $Ax = b$, we multiply by a matrix M^{-1} and apply the iterative method to the mathematically equivalent system

$$M^{-1}Ax = M^{-1}b. \quad (11.18)$$

The aim now is to construct a nonsingular *preconditioning matrix or algorithm* such that $M^{-1}A$ has a more favorable condition number than A . We remark that we use the notation M^{-1} here to indicate that it should resemble the inverse of the matrix A .

For increased flexibility we can write $M^{-1} = C_L C_R$ and transform the system according to

$$C_L A C_R y = C_L b, \quad y = C_R^{-1} x, \quad (11.19)$$

where C_L is the *left* and C_R is the *right* preconditioner. If the original coefficient matrix A is symmetric and positive definite, $C_L = C_R^T$ leads to preservation of these properties in the transformed system. This is important when applying the Conjugate Gradient method to the preconditioned linear system. Even if A and M are symmetric and positive definite, $M^{-1}A$ does not necessarily inherit these properties. It appears that for practical purposes one can express the iterative algorithms such that it is sufficient to work with a single preconditioning matrix M only [4, 8]. We shall therefore speak of preconditioning in terms of the left preconditioner M in the following.

11.2.2 Use of the preconditioning matrix in the iterative methods

Optimal convergence for the Conjugate Gradient method is achieved when the coefficient matrix $M^{-1}A$ equals the identity matrix I and only one iteration is required. In the algorithm we need to perform matrix-vector products $M^{-1}Au$ for an arbitrary $u \in \mathbb{R}^n$. This means that we have to solve a linear system with M as coefficient matrix in each iteration since we implement the product $y = M^{-1}Au$ in a two step fashion: First we compute $v = Au$ and then we solve the linear system $My = v$ for y . The optimal choice $M = A$ therefore involves the solution of $Ay = v$ in each iteration, which is a problem of the same complexity as our original system $Ax = b$. The strategy must hence be to compute an $M^{-1} \approx A^{-1}$ such that the algorithmic operations involved in the inversion of M are cheap.

The preceding discussion motivates the following demands on the preconditioning matrix M :

- M^{-1} should be a good approximation to A ,
- M^{-1} should be inexpensive to compute,
- M^{-1} should be sparse in order to minimize storage requirements,
- linear systems with M as coefficient matrix must be efficiently solved.

Regarding the last property, such systems must be solved in $\mathcal{O}(n)$ operations, that is, a complexity of the same order as the vector updates

in the Conjugate Gradient-like algorithms. These four properties are contradictory and some sort of compromise must be sought.

11.2.3 Classical iterative methods as preconditioners

The simplest possible iterative method for solving $Ax = b$ is

$$x^{k+1} = x^k + r^k.$$

Applying this method to the preconditioned system $M^{-1}Ax = M^{-1}b$ results in the scheme

$$x^{k+1} = x^k + M^{-1}r^k,$$

which is nothing but the formula for classical iterative methods such as the Jacobi method, the Gauss-Seidel method, SOR (Successive over-relaxation), and SSOR (Symmetric successive over-relaxation). This motivates for choosing M as one iteration of these classical methods. In particular, these methods provide simple formulas for M :

- Jacobi preconditioning: $M = D$.
- Gauss-Seidel preconditioning: $M = D + L$.
- SOR preconditioning: $M = \omega^{-1}D + L$.
- SSOR preconditioning: $M = (2-\omega)^{-1} (\omega^{-1}D + L) (\omega^{-1}D)^{-1} (\omega^{-1}D + U)$

Turning our attention to the four requirements of the preconditioning matrix, we realize that the suggested M matrices do not demand additional storage, linear systems with M as coefficient matrix are solved effectively in $\mathcal{O}(n)$ operations, and M needs no initial computation. The only questionable property is how well M approximates A , and that is the weak point of using classical iterative methods as preconditioners.

The Conjugate Gradient method can only utilize the Jacobi and SSOR preconditioners among the classical iterative methods, because the M matrix in that case is on the form $M^{-1} = C_L C_L^T$, which is necessary to ensure that the coefficient matrix of the preconditioned system is symmetric and positive definite. For certain PDEs, like the Poisson equation, it can be shown that the SSOR preconditioner reduces the condition number with an order of magnitude, i.e., from $\mathcal{O}(h^{-2})$ to $\mathcal{O}(h^{-1})$, provided we use the optimal choice of the relaxation parameter ω . According to experiments, however, the performance of the SSOR preconditioned Conjugate Gradient method is not very sensitive to the

choice of ω . We refer to [4, 8] for more information about classical iterative methods as preconditioners.

11.2.4 Incomplete factorization preconditioners

Imagine that we choose $M = A$ and solve systems $My = v$ by a direct method. Such methods typically first compute the LU factorization $M = \bar{L}\bar{U}$ and thereafter perform two triangular solves. The lower and upper triangular factors \bar{L} and \bar{U} are computed from a Gaussian elimination procedure. Unfortunately, \bar{L} and \bar{U} contain nonzero values, so-called *fill-in*, in many locations where the original matrix A contains zeroes. This decreased sparsity of \bar{L} and \bar{U} increases both the storage requirements and the computational efforts related to solving systems $My = v$. An idea to improve the situation is to compute *sparse* versions of the factors \bar{L} and \bar{U} . This is achieved by performing Gaussian elimination, but neglecting the fill-in (!). In this way, we can compute approximate factors \hat{L} and \hat{U} that become as sparse as A . The storage requirements are hence only doubled by introducing a preconditioner, and the triangular solves become an $\mathcal{O}(n)$ operation since the number of nonzeros in the \hat{L} and \hat{U} matrices (and A) is $\mathcal{O}(n)$ when the underlying PDE is discretized by finite difference or finite element methods. We call $M = \hat{L}\hat{U}$ an *incomplete LU factorization* preconditioner, often just referred to as the ILU preconditioner.

Instead of throwing away all fill-in entries, we can add them to the main diagonal. This yields the *modified incomplete LU factorization* (MILU). One can also allow a certain amount of fill-in to improve the quality of the preconditioner, at the expense of more storage for the factors. Libraries offering ILU preconditioners will then often have a tolerance for how small a fill-in element in the Gaussian elimination must be to be dropped, and a parameter that governs the maximum number of nonzeros per row in the factors. A popular ILU implementation is the open source C code [SuperLU](#). This preconditioner is available to Python programmers through the `scipy.sparse.linalg.spilu` function.

The general algorithm for MILU preconditioning follows the steps of traditional exact Gaussian elimination, except that we restrict the computations to the nonzero entries in A . The factors \hat{L} and \hat{U} can be stored directly in the sparsity structure of A , that is, the algorithm overwrites a copy M of A with its MILU factorization. The steps in the MILU factorizations are listed below.

1. Given a sparsity pattern as an index set \mathcal{I} , copy $M_{i,j} \leftarrow A_{i,j}$, $i, j = 1, \dots, n$
2. for $k = 1, 2, \dots, n$
 - a. for $i = k + 1, \dots, n$
 - if $(i, k) \in \mathcal{I}$ then
 - i. $M_{i,k} \leftarrow M_{i,k}/M_{k,k}$
 - else
 - i. $M_{i,k} = 0$
 - $r = M_{i,k}$
 - for $j = k + 1, \dots, n$
 - if $j = i$ then
 - $M_{i,j} \leftarrow M_{i,j} - rM_{k,j} + \sum_{p=k+1}^n (M_{i,p} - rM_{k,p})$
 - else
 - if $(i, j) \in \mathcal{I}$ then
 - $M_{i,j} \leftarrow M_{i,j} - rM_{k,j}$
 - else
 - $M_{i,j} = 0$

We also remark here that the algorithm above needs careful refinement before it should be implemented in a code. For example, one will not run through a series of (i, j) indices and test for each of them if $(i, j) \in \mathcal{I}$. Instead one should run more directly through the sparsity structure of A . More comprehensive material on how to solve sparse linear systems can be found in e.g. [27] and [12].

11.2.5 Preconditioners developed for solving PDE problems

The above mentioned preconditioners are general purpose preconditioners that may be applied to solve any linear system and may speed up the solution process significantly. For linear systems arising from PDE problems there are however often more efficient preconditioners that exploit the fact that the matrices come from discretized PDEs. These preconditioners often enable the solution of linear systems involving millions of unknowns in a matter of seconds or less on a regular desktop

computer. This means that solving these huge linear systems often takes less time than printing them file. Moreover, these preconditioners may be used on parallel computers in a scalable fashion such that simulations involving billions of degrees of freedom are available e.g. by cloud services or super-computing centers.

The most efficient preconditioners are the so-called multilevel preconditioners (e.g. multigrid and domain decomposition) which in particular for Poisson type problems yields error reduction of a factor 10 per iteration and hence a typical iteration count of the iterative method of 5-10. These preconditioners utilize the fact that different parts of the solution can be localized or represented on a coarser resolution during the computations before being glued together to a close match to the true solution. While a proper description of these methods are beyond the scope here, we mention that black-box preconditioners are implemented in open source frameworks such as PETSc, Hypre, and Trilinos and these frameworks are used by most available finite element software (such as FEniCS). Many books have been devoted to the topic, c.f. e.g. [28, 30].

While these multilevel preconditioners are available and efficient for PDEs similar to the Poisson problem there is currently no black-box solver that handles general systems of PDEs. We mention, however, a promising approach for tackling some systems of PDEs, namely the so-called operator preconditioning framework [24]. This framework utilizes tools from functional analysis to deduct appropriate block preconditioners typically involving blocks composed of for instance multilevel preconditioners in a systematic construction. The framework has extended the use of Poisson type multilevel preconditioners to many systems of PDEs with success.

A.1 Finite difference operator notation

All the finite differences here, and their corresponding operator notation, take place on a time mesh for a function of time. The same formulas apply in space too (just replace t by a spatial coordinate and add spatial coordinates in u).

$$u'(t_n) \approx [D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} \quad (\text{A.1})$$

$$u'(t_n) \approx [D_{2t} u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t} \quad (\text{A.2})$$

$$u'(t_n) \approx [D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \quad (\text{A.3})$$

$$u'(t_n) \approx [D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t} \quad (\text{A.4})$$

$$u'(t_{n+\theta}) \approx [\bar{D}_t u]^{n+\theta} = \frac{u^{n+1} - u^n}{\Delta t} \quad (\text{A.5})$$

$$u'(t_n) \approx [D_t^{2-} u]^n = \frac{3u^n - 4u^{n-1} + u^{n-2}}{2\Delta t} \quad (\text{A.6})$$

$$u''(t_n) \approx [D_t D_t u]^n = \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} \quad (\text{A.7})$$

$$u(t_{n+\frac{1}{2}}) \approx [\bar{u}^t]^{n+\frac{1}{2}} = \frac{1}{2}(u^{n+1} + u^n) \quad (\text{A.8})$$

$$u(t_{n+\frac{1}{2}})^2 \approx [\bar{u}^{2,t,g}]^{n+\frac{1}{2}} = u^{n+1} u^n \quad (\text{A.9})$$

$$u(t_{n+\frac{1}{2}}) \approx [\bar{u}^{t,h}]^{n+\frac{1}{2}} = \frac{2}{\frac{1}{u^{n+1}} + \frac{1}{u^n}} \quad (\text{A.10})$$

$$u(t_{n+\theta}) \approx [\bar{u}^{t,\theta}]^{n+\theta} = \theta u^{n+1} + (1 - \theta) u^n, \quad (\text{A.11})$$

$$t_{n+\theta} = \theta t_{n+1} + (1 - \theta) t_{n-1} \quad (\text{A.12})$$

A.2 Truncation errors of finite difference approximations

$$\begin{aligned} u'_e(t_n) &= [D_t u_e]^n + R^n = \frac{u_e^{n+\frac{1}{2}} - u_e^{n-\frac{1}{2}}}{\Delta t} + R^n, \\ R^n &= -\frac{1}{24} u'''_e(t_n) \Delta t^2 + \mathcal{O}(\Delta t^4) \end{aligned} \quad (\text{A.13})$$

$$\begin{aligned} u'_e(t_n) &= [D_{2t} u_e]^n + R^n = \frac{u_e^{n+1} - u_e^{n-1}}{2\Delta t} + R^n, \\ R^n &= -\frac{1}{6} u'''_e(t_n) \Delta t^2 + \mathcal{O}(\Delta t^4) \end{aligned} \quad (\text{A.14})$$

$$\begin{aligned} u'_e(t_n) &= [D_t^- u_e]^n + R^n = \frac{u_e^n - u_e^{n-1}}{\Delta t} + R^n, \\ R^n &= -\frac{1}{2} u''_e(t_n) \Delta t + \mathcal{O}(\Delta t^2) \end{aligned} \quad (\text{A.15})$$

$$\begin{aligned} u'_e(t_n) &= [D_t^+ u_e]^n + R^n = \frac{u_e^{n+1} - u_e^n}{\Delta t} + R^n, \\ R^n &= -\frac{1}{2} u''_e(t_n) \Delta t + \mathcal{O}(\Delta t^2) \end{aligned} \quad (\text{A.16})$$

$$\begin{aligned} u'_e(t_{n+\theta}) &= [\bar{D}_t u_e]^{n+\theta} + R^{n+\theta} = \frac{u_e^{n+1} - u_e^n}{\Delta t} + R^{n+\theta}, \\ R^{n+\theta} &= -\frac{1}{2}(1-2\theta) u''_e(t_{n+\theta}) \Delta t + \frac{1}{6}((1-\theta)^3 - \theta^3) u'''_e(t_{n+\theta}) \Delta t^2 + \\ &\quad \mathcal{O}(\Delta t^3) \end{aligned} \quad (\text{A.17})$$

$$\begin{aligned} u'_e(t_n) &= [D_t^{2-} u_e]^n + R^n = \frac{3u_e^n - 4u_e^{n-1} + u_e^{n-2}}{2\Delta t} + R^n, \\ R^n &= \frac{1}{3} u'''_e(t_n) \Delta t^2 + \mathcal{O}(\Delta t^3) \end{aligned} \quad (\text{A.18})$$

$$\begin{aligned} u''_e(t_n) &= [D_t D_t u_e]^n + R^n = \frac{u_e^{n+1} - 2u_e^n + u_e^{n-1}}{\Delta t^2} + R^n, \\ R^n &= -\frac{1}{12} u''''_e(t_n) \Delta t^2 + \mathcal{O}(\Delta t^4) \end{aligned} \quad (\text{A.19})$$

$$\begin{aligned} u_e(t_{n+\theta}) &= [\bar{u}_e^{t,\theta}]^{n+\theta} + R^{n+\theta} = \theta u_e^{n+1} + (1-\theta) u_e^n + R^{n+\theta}, \\ R^{n+\theta} &= -\frac{1}{2} u''_e(t_{n+\theta}) \Delta t^2 \theta(1-\theta) + \mathcal{O}(\Delta t^3). \end{aligned} \quad (\text{A.20})$$

A.3 Finite differences of exponential functions

Complex exponentials. Let $u^n = \exp(i\omega n \Delta t) = e^{i\omega t_n}$.

$$[D_t D_t u]^n = u^n \frac{2}{\Delta t} (\cos \omega \Delta t - 1) = -\frac{4}{\Delta t} \sin^2 \left(\frac{\omega \Delta t}{2} \right), \quad (\text{A.21})$$

$$[D_t^+ u]^n = u^n \frac{1}{\Delta t} (\exp(i\omega \Delta t) - 1), \quad (\text{A.22})$$

$$[D_t^- u]^n = u^n \frac{1}{\Delta t} (1 - \exp(-i\omega \Delta t)), \quad (\text{A.23})$$

$$[D_t u]^n = u^n \frac{2}{\Delta t} i \sin \left(\frac{\omega \Delta t}{2} \right), \quad (\text{A.24})$$

$$[D_{2t} u]^n = u^n \frac{1}{\Delta t} i \sin(\omega \Delta t). \quad (\text{A.25})$$

Real exponentials. Let $u^n = \exp(\omega n \Delta t) = e^{\omega t_n}$.

$$[D_t D_t u]^n = u^n \frac{2}{\Delta t} (\cos \omega \Delta t - 1) = -\frac{4}{\Delta t} \sin^2 \left(\frac{\omega \Delta t}{2} \right), \quad (\text{A.26})$$

$$[D_t^+ u]^n = u^n \frac{1}{\Delta t} (\exp(i\omega \Delta t) - 1), \quad (\text{A.27})$$

$$[D_t^- u]^n = u^n \frac{1}{\Delta t} (1 - \exp(-i\omega \Delta t)), \quad (\text{A.28})$$

$$[D_t u]^n = u^n \frac{2}{\Delta t} i \sin \left(\frac{\omega \Delta t}{2} \right), \quad (\text{A.29})$$

$$[D_{2t} u]^n = u^n \frac{1}{\Delta t} i \sin(\omega \Delta t). \quad (\text{A.30})$$

A.4 Finite differences of t^n

The following results are useful when checking if a polynomial term in a solution fulfills the discrete equation for the numerical method.

$$[D_t^+ t]^n = 1, \quad (\text{A.31})$$

$$[D_t^- t]^n = 1, \quad (\text{A.32})$$

$$[D_t t]^n = 1, \quad (\text{A.33})$$

$$[D_{2t} t]^n = 1, \quad (\text{A.34})$$

$$[D_t D_t t]^n = 0. \quad (\text{A.35})$$

The next formulas concern the action of difference operators on a t^2 term.

$$[D_t^+ t^2]^n = (2n + 1)\Delta t, \quad (\text{A.36})$$

$$[D_t^- t^2]^n = (2n - 1)\Delta t, \quad (\text{A.37})$$

$$[D_t t^2]^n = 2n\Delta t, \quad (\text{A.38})$$

$$[D_{2t} t^2]^n = 2n\Delta t, \quad (\text{A.39})$$

$$[D_t D_t t^2]^n = 2, \quad (\text{A.40})$$

Finally, we present formulas for a t^3 term: **These must be controlled against lib.py.** Use t_n instead of $n\Delta t$??

$$[D_t^+ t^3]^n = 3(n\Delta t)^2 + 3n\Delta t^2 + \Delta t^2, \quad (\text{A.41})$$

$$[D_t^- t^3]^n = 3(n\Delta t)^2 - 3n\Delta t^2 + \Delta t^2, \quad (\text{A.42})$$

$$[D_t t^3]^n = 3(n\Delta t)^2 + \frac{1}{4}\Delta t^2, \quad (\text{A.43})$$

$$[D_{2t} t^3]^n = 3(n\Delta t)^2 + \Delta t^2, \quad (\text{A.44})$$

$$[D_t D_t t^3]^n = 6n\Delta t, \quad (\text{A.45})$$

A.4.1 Software

Application of finite difference operators to polynomials and exponential functions, resulting in the formulas above, can easily be computed by some `sympy` code:

```
from sympy import *
t, dt, n, w = symbols('t dt n w', real=True)

# Finite difference operators

def D_t_forward(u):
    return (u(t + dt) - u(t))/dt

def D_t_backward(u):
    return (u(t) - u(t-dt))/dt

def D_t_centered(u):
    return (u(t + dt/2) - u(t-dt/2))/dt

def D_2t_centered(u):
    return (u(t + dt) - u(t-dt))/(2*dt)

def D_tD_t(u):
```

```

        return (u(t + dt) - 2*u(t) + u(t-dt))/(dt**2)

op_list = [D_t_forward, D_t_backward,
           D_t_centered, D_2t_centered, D_t_D_t]

def ft1(t):
    return t

def ft2(t):
    return t**2

def ft3(t):
    return t**3

def f_expiwt(t):
    return exp(I*w*t)

def f_expwt(t):
    return exp(w*t)

func_list = [ft1, ft2, ft3, f_expiwt, f_expwt]

```

To see the results, one can now make a simple loop like

```

for func in func_list:
    for op in op_list:
        f = func
        e = op(f)
        e = simplify(expand(e))
        print(e)
        if func in [f_expiwt, f_expwt]:
            e = e/f(t)
            e = e.subs(t, n*dt)
        print(expand(e))
        print(factor(simplify(expand(e))))

```

References

- [1] M. S. Alnæs and K.-A. Mardal. On the efficiency of symbolic computations combined with code generation for finite element methods. *ACM Transactions on Mathematical Software (TOMS)*, 37(1):6, 2010.
- [2] D. N. Arnold and A. Logg. Periodic table of the finite elements. *SIAM News*, 47(9):212, 2014.
- [3] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, 1996.
- [4] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, second edition, 1994. http://www.netlib.org/linalg/html_templates/Templates.html.
- [5] D. Braess. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*. Cambridge University Press, third edition, 2007.
- [6] S. Brenner and R. Scott. *The Mathematical Theory of Finite Element Methods*. Springer, third edition, 2007.
- [7] F. Brezzi and M. Fortin. *Mixed and hybrid finite element methods*, volume 15. Springer Science & Business Media, 2012.
- [8] A. M. Bruaset. *A Survey of Preconditioned Iterative Methods*. Chapman and Hall, 1995.
- [9] H. Elman, D. Silvester, and A. Wathen. *Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics*. Oxford University Press, second edition, 2015.
- [10] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational*

- Differential Equations*. Cambridge University Press, second edition, 1996.
- [11] P. M. Gresho and R. L. Sani. *Incompressible flow and the finite element method*. John Wiley and Sons, Inc., New York, NY (United States), 1998.
 - [12] W. Hackbusch. *Iterative solution of large sparse systems of equations*, volume 95. Springer, 1994.
 - [13] C. Johnson. *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Dover, 2009.
 - [14] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. SIAM, 1995.
 - [15] R. C. Kirby. Fast simplicial finite element algorithms using bernstein polynomials. *Numerische Mathematik*, 117(4):631–652, 2011.
 - [16] R. C. Kirby, A. Logg, M. E. Rognes, , and A. R. Terrel. Common and unusual finite elements. In A. Logg, K.-A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, pages 95–119. Springer, 2012.
 - [17] R. C. Kirby and K.-A. Mardal. Constructing general reference finite elements. In A. Logg, K.-A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, pages 121–132. Springer, 2012.
 - [18] H. P. Langtangen. *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming*. Texts in Computational Science and Engineering. Springer, second edition, 2003.
 - [19] H. P. Langtangen and A. Logg. *Solving PDEs in Hours – The FEniCS Tutorial Volume II*. 2016. <http://hplgit.github.io/fenics-tutorial/doc/web/>.
 - [20] H. P. Langtangen and A. Logg. *Solving PDEs in Minutes – The FEniCS Tutorial Volume I*. 2016. <http://hplgit.github.io/fenics-tutorial/doc/web/>.
 - [21] H. P. Langtangen, K.-A. Mardal, and R. Winther. Numerical methods for incompressible viscous flow. *Advances in water Resources*, 25(8-12):1125–1146, 2002.
 - [22] M. G. Larson and F. Bengzon. *The Finite Element Method: Theory, Implementation, and Applications*. Texts in Computational Science and Engineering. Springer, 2013.
 - [23] A. Logg, K.-A. Mardal, and G. N. Wells. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.
 - [24] K.-A. Mardal and R. Winther. Preconditioning discretizations of systems of partial differential equations. *Numerical Linear Algebra*

- with Applications*, 18(1):1–40, 2011.
- [25] M. Mortensen, H. P. Langtangen, and G. N. Wells. A FEniCS-based programming framework for modeling turbulent flow by the Reynolds-averaged Navier-Stokes equations. *Advances in Water Resources*, 34(9), 2011.
 - [26] A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*. Springer, 1994.
 - [27] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, second edition, 2003. http://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf.
 - [28] B. Smith, P. Bjorstad, and W. D. Gropp. *Domain decomposition: parallel multilevel methods for elliptic partial differential equations*. Cambridge university press, 2004.
 - [29] V. Thomée. *Galerkin finite element methods for parabolic problems*. 1984.
 - [30] U. Trottenberg, C.W. Oosterlee, and A. Schuller. *Multigrid*. Elsevier, 2000.
 - [31] S. Turek. *Efficient Solvers for Incompressible Flow Problems: An Algorithmic and Computational Approach*, volume 6. Springer Science & Business Media, 1999.
 - [32] A. Tveito and R. Winther. *Introduction to partial differential equations: a computational approach*. Springer Science & Business Media;, 2004.
 - [33] O. C. Zienkiewicz and R. L. Taylor. *The finite element method*. McGraw-hill London, 1977.

Index

- $A^T A = A^T b$ (normal equations), 27
affine mapping, 116, 151
approximation
 by sines, 34
 collocation, 39
 interpolation, 39
 of functions, 18
 of general vectors, 14
 of vectors in the plane, 11
assembly, 113
basis vector, 11
Bernstein(interpolating) polynomial, 49
cell, 137
`cells` list, 139
chapeau function, 108
Chebyshev nodes, 45
collocation method (approximation), 39
continuation method, 456, 483
convection-diffusion, 224, 299
degree of freedom, 137
dof map, 137
`dof_map` list, 139
edges, 150
element matrix, 113
essential boundary condition, 217
`Expression`, 155
faces, 150
FEniCS, 153
finite element basis function, 108
finite element expansion
 reference element, 138
finite element mesh, 101
finite element, definition, 137
fixed-point iteration, 411
`FunctionSpace`, 155
Galerkin method
 functions, 20
 vectors, 13, 17
Gauss-Legendre quadrature, 145

- group finite element method, 460
- hat function, 108
- Hermite polynomials, 142
- ILU, 496
- incomplete factorization, 496
- integration by parts, 204
- internal node, 103
- interpolation method (approximation), 39
- isoparametric mapping, 151
- Kronecker delta, 42, 104
- Krylov space, 487
- Lagrange (interpolating) polynomial, 42
- `latex.codecogs.com` web site, 478
- least squares method
- vectors, 12
- linear elements, 108
- linear solvers
- conjugate gradients, 491
 - GCR, 490
 - generalized conjugate residuals, 490
 - GMRES, 490
 - minimum residuals, 490
 - preconditioning, 493
- linear systems
- preconditioned, 493
- linearization, 411
- explicit time integration, 408
 - fixed-point iteration, 411
 - Picard iteration, 411
 - successive substitutions, 411
- lumped mass matrix, 136, 341
- mapping of reference cells
- affine mapping, 116
- isoparametric mapping, 151
- mass lumping, 136, 341
- mass matrix, 136, 337, 341
- mesh
- finite elements, 101
- method of weighted residuals, 196
- Midpoint rule, 144
- MILU, 496
- mixed finite elements, 369
- natural boundary condition, 217
- Newton-Cotes rules, 144
- norm, 11
- normal equations, 27
- numerical integration
- Midpoint rule, 144
 - Newton-Cotes formulas, 144
 - Simpson's rule, 144
 - Trapezoidal rule, 144
- online rendering of L^AT_EX formulas, 478
- P1 element, 108
- P2 element, 108
- Petrov-Galerkin methods, 299
- Picard iteration, 411
- preconditioning, 493
- classical iterations, 495
- product approximation technique, 460
- `project` (FEniCS function), 155
- projection
- functions, 20
 - vectors, 13, 17
- quadratic elements, 108
- reference cell, 137
- relaxation (nonlinear equations), 416

residual, 194
Runge's phenomenon, 45

search (direction) vectors, 490
shared node, 103
simplex elements, 150
simplices, 150
Simpson's rule, 144
single Picard iteration technique,
 412
`solve` (FEniCS function), 155
sparse matrices, 128
stiffness matrix, 337
stopping criteria (nonlinear prob-
 lems), 412, 428
strong form, 205
successive substitutions, 411

tensor product, 59
test function, 3, 198
test space, 198
`TestFunction`, 155
Trapezoidal rule, 144
trial function, 3, 198
trial space, 198
`TrialFunction`, 155

variational formulation, 196
vertex, 137
`vertices` list, 139

weak form, 205
weak formulation, 196
weighted residuals, 196