

Logg
Mardal
Wells
(Eds.)

Automated Scientific Computing



www.fenics.org

Automated Scientific Computing
Logg, Mardal, Wells (editors)

This is a preliminary draft, June 9, 2009.
Send comments and suggestions to fenics-book-dev@fenics.org.

Automated Scientific Computing

The FEniCS Book

Preliminary draft, June 9, 2009

And now that we may give final praise to the machine we may say that it will be desirable to all who are engaged in computations which, it is well known, are the managers of financial affairs, the administrators of others' estates, merchants, surveyors, geographers, navigators, astronomers... For it is unworthy of excellent men to lose hours like slaves in the labor of calculations which could safely be relegated to anyone else if the machine were used.

Gottfried Wilhelm Leibniz (1646–1716)

1	Introduction	
	By Anders Logg, Garth N. Wells and Kent-Andre Mardal	17
2	A FEniCS Tutorial	
	By Hans Petter Langtangen	19
I	Methodology	21
3	The Finite Element Method	
	By Robert C. Kirby and Anders Logg	23
3.1	A Simple Model Problem	23
3.2	Finite Element Discretization	25
3.2.1	Discretizing Poisson's equation	25
3.2.2	Discretizing the first order system	27
3.3	Finite Element Abstract Formalism	28
3.3.1	Linear problems	28
3.3.2	Nonlinear problems	29
3.4	Finite Element Function Spaces	30
3.4.1	The mesh	30
3.4.2	The finite element definition	32
3.4.3	The nodal basis	32
3.4.4	The local-to-global mapping	33
3.4.5	The global function space	34
3.4.6	The mapping from the reference element	36
3.5	Finite Element Solvers	38
3.6	Finite Element Error Estimation and Adaptivity	39
3.6.1	A priori error analysis	40

3.6.2	A posteriori error analysis	41
3.6.3	Adaptivity	43
3.7	Automating the Finite Element Method	44
3.8	Outlook	45
3.9	Historical Notes	45
4	Common and Unusual Finite Elements	
	By Robert C. Kirby, Anders Logg and Andy R. Terrel	47
4.1	Ciarlet's Finite Element Definition	47
4.2	Notation	48
4.3	The Argyris Element	50
4.3.1	Definition	50
4.3.2	Historical notes	50
4.4	The Brezzi–Douglas–Marini element	51
4.4.1	Definition	51
4.4.2	Historical notes	51
4.5	The Crouzeix–Raviart element	53
4.5.1	Definition	53
4.5.2	Historical notes	53
4.6	The Hermite Element	54
4.6.1	Definition	54
4.6.2	Historical notes	54
4.7	The Lagrange Element	55
4.7.1	Definition	55
4.7.2	Historical notes	56
4.8	The Morley Element	57
4.8.1	Definition	57
4.8.2	Historical notes	57
4.9	The Nédélec Element	58
4.9.1	Definition	58
4.9.2	Historical notes	59
4.10	The PEERS Element	61
4.10.1	Definition	61
4.10.2	Historical notes	61
4.11	The Raviart–Thomas Element	63
4.11.1	Definition	63
4.11.2	Historical notes	63
4.12	Summary	65
5	Constructing General Reference Finite Elements	
	By Robert C. Kirby and Kent-Andre Mardal	69
5.1	Introduction	69
5.2	Preliminaries	71

5.3	Mathematical Framework	73
5.3.1	Change of basis	73
5.3.2	Polynomial spaces	74
5.4	Examples of Elements	76
5.4.1	Bases for other polynomial spaces	78
5.5	Operations on the Polynomial spaces	80
5.5.1	Evaluation	80
5.5.2	Differentiation	80
5.5.3	Integration	81
5.5.4	Linear functionals	82
6	Finite Element Variational Forms	
	By Robert C. Kirby and Anders Logg	83
7	Finite Element Assembly	
	By Anders Logg	85
8	Quadrature Representation of Finite Element Variational Forms	
	By Kristian B. Ølgaard and Garth N. Wells	87
9	Tensor Representation of Finite Element Variational Forms	
	By Anders Logg and possibly others	89
10	Discrete Optimization of Finite Element Matrix Evaluation	
	By Robert C. Kirby, Matthew G. Knepley, Anders Logg, L. Ridgway Scott and Andy R. Terrel	91
11	Parallel Adaptive Mesh Refinement	
	By Johan Hoffman, Johan Jansson and Niclas Jansson	93
11.1	A brief overview of parallel computing	93
11.2	Local mesh refinement	94
11.2.1	The challenge of parallel mesh refinement	94
11.2.2	A modified longest edge bisection algorithm	95
11.3	The need of dynamic load balancing	97
11.3.1	Workload modelling	98
11.3.2	Remapping strategies	99
11.4	The implementation on a massively parallel system	99
11.4.1	The refinement method	100
11.4.2	The remapping scheme	101
11.4.3	Theoretical and experimental analysis	102
11.5	Summary and outlook	105

II	Implementation	107
12	DOLFIN: A C++/Python Finite Element Library	
	By Anders Logg and Garth N. Wells	109
13	FFC: A Finite Element Form Compiler	
	By Anders Logg and possibly others	111
14	FErari: An Optimizing Compiler for Variational Forms	
	By Robert C. Kirby and Anders Logg	113
15	FIAT: Numerical Construction of Finite Element Basis Functions	
	By Robert C. Kirby	115
	15.1 Introduction	115
	15.2 Prime basis: Collapsed-coordinate polynomials	116
	15.3 Representing polynomials and functionals	117
	15.4 Other polynomial spaces	120
	15.4.1 Supplemented polynomial spaces	121
	15.4.2 Constrained polynomial spaces	121
	15.5 Conveying topological information to clients	122
	15.6 Functional evaluation	123
	15.7 Overview of fundamental class structure	124
16	Instant: Just-in-Time Compilation of C/C++ Code in Python	
	By Ilmar M. Wilbers, Kent-Andre Mardal and Martin S. Alnæs	127
	16.1 Introduction	127
	16.2 Examples	128
	16.2.1 Installing Instant	128
	16.2.2 Hello World	129
	16.2.3 NumPy Arrays	130
	16.2.4 Ordinary Differential Equations	130
	16.2.5 Numpy Arrays and OpenMP	132
	16.3 Instant Explained	134
	16.3.1 Arrays and Typemaps	136
	16.3.2 Module name, signature, and cache	140
	16.3.3 Locking	141
	16.4 Instant API	141
	16.4.1 build_module	141
	16.4.2 inline	146
	16.4.3 inline_module	146
	16.4.4 inline_with_numpy	146
	16.4.5 inline_module_with_numpy	146
	16.4.6 import_module	146
	16.4.7 header_and_libs_from_pkgconfig	147

16.4.8	get_status_output	147
16.4.9	get_swig_version	148
16.4.10	check_swig_version	148
17	SyFi: Symbolic Construction of Finite Element Basis Functions	
	By Martin S. Alnæs and Kent-Andre Mardal	149
18	UFC: A Finite Element Code Generation Interface	
	By Martin S. Alnæs, Anders Logg and Kent-Andre Mardal	151
19	UFL: A Finite Element Form Language	
	By Martin Sandve Alnæs	153
19.1	Overview	154
19.1.1	Design goals	154
19.1.2	Motivational example	155
19.2	Defining finite element spaces	156
19.3	Defining forms	159
19.4	Defining expressions	160
19.4.1	Form arguments	161
19.4.2	Index notation	162
19.4.3	Algebraic operators and functions	164
19.4.4	Differential operators	165
19.4.5	Other operators	167
19.5	Form operators	168
19.5.1	Differentiating forms	168
19.5.2	Adjoint	170
19.5.3	Replacing functions	170
19.5.4	Action	171
19.5.5	Splitting a system	171
19.5.6	Computing the sensitivity of a function	171
19.6	Expression representation	172
19.6.1	The structure of an expression	172
19.6.2	Tree representation	173
19.6.3	Expression node properties	174
19.6.4	Linearized graph representation	175
19.6.5	Partitioning	176
19.7	Computing derivatives	176
19.7.1	Relations to form compiler approaches	177
19.7.2	Approaches to computing derivatives	178
19.7.3	Forward mode Automatic Differentiation	178
19.7.4	Extensions to tensors and indexed expressions	179
19.7.5	Higher order derivatives	180
19.7.6	Basic differentiation rules	181
19.8	Algorithms	183

19.8.1	Effective tree traversal in Python	183
19.8.2	Type based function dispatch in Python	183
19.8.3	Implementing expression transformations	185
19.8.4	Important transformations	186
19.8.5	Evaluating expressions	187
19.8.6	Viewing expressions	188
19.9	Implementation issues	188
19.9.1	Python as a basis for a domain specific language	188
19.9.2	Ensuring unique form signatures	189
19.9.3	Efficiency considerations	190
19.10	Future directions	190
19.11	Acknowledgements	191
20	Unicorn: A Unified Continuum Mechanics Solver	
	By Johan Hoffman, Johan Jansson, Niclas Jansson and Murtazo Nazarov	193
20.1	Unified Continuum modeling	194
20.1.1	Automated computational modeling and software design	195
20.2	Space-time General Galerkin discretization	195
20.2.1	Standard Galerkin	196
20.2.2	Local ALE	196
20.2.3	Streamline diffusion stabilization	197
20.2.4	Duality-based adaptive error control	197
20.2.5	Unicorn/FEniCS software implementation	197
20.3	Unicorn classes: data types and algorithms	198
20.3.1	Unicorn software design	198
20.3.2	TimeDependentPDE	199
20.3.3	ErrorEstimate	200
20.3.4	SlipBC	202
20.4	Mesh adaptivity	203
20.4.1	Local mesh operations: Madlib	203
20.4.2	Elastic mesh smoothing: cell quality optimization	203
20.4.3	Recursive Rivara bisection	203
20.5	Parallel computation	203
20.5.1	Tensor assembly	203
20.5.2	Mesh refinement	203
20.6	Application examples	203
20.6.1	Incompressible flow	203
20.6.2	Compressible flow	203
20.6.3	Fluid-structure interaction	203
21	Viper: A Minimalistic Scientific Plotter	
	By Ola Skavhaug	207

22 Lessons Learnt in Mixed Language Programming	209
By Kent-Andre Mardal, Anders Logg, and Ola Skavhaug	
III Applications	211
23 Finite Elements for Incompressible Fluids	
By Andy R. Terrel, L. Ridgway Scott, Matthew G. Knepley, Robert C. Kirby and Garth N. Wells	213
24 Benchmarking Finite Element Methods for Navier–Stokes	
By Kristian Valen-Sendstad, Anders Logg and Kent-Andre Mardal	215
25 Image-Based Computational Hemodynamics	
By Luca Antiga	217
26 Simulating the Hemodynamics of the Circle of Willis	
By Kristian Valen-Sendstad, Kent-Andre Mardal and Anders Logg	219
27 Cerebrospinal Fluid Flow	
By Susanne Hentschel, Svein Linge, Emil Alf Løvgren and Kent-Andre Mardal	221
27.1 Medical Background	221
27.2 Mathematical Description	223
27.3 Numerical Experiments	223
27.3.1 Implementation	223
27.3.2 Example 1. Simulation of a Pulse in the SAS.	230
27.3.3 Example 2. Simplified Boundary Conditions.	234
27.3.4 Example 3. Cord Shape and Position.	235
27.3.5 Example 4. Cord with Syrxinx.	236
28 Turbulent Flow and Fluid–Structure Interaction with Unicorn	
By Johan Hoffman, Johan Jansson, Niclas Jansson, Claes Johnson and Murtazo Nazarov	239
28.1 Introduction	239
28.2 Continuum models	240
28.3 Mathematical framework	241
28.4 Computational method	241
28.5 Boundary conditions	241
28.6 Geometry modeling	242
28.7 Fluid-structure interaction	242
28.8 Applications	242
28.8.1 Turbulent flow separation	242
28.8.2 Flight aerodynamics	242
28.8.3 Vehicle aerodynamics	242
28.8.4 Biomedical flow	242

28.8.5 Aeroacoustics	242
28.8.6 Gas flow	243
28.9 References	243
29 Fluid–Structure Interaction using Nitsche’s Method	
By Kristoffer Selim and Anders Logg	245
30 Improved Boussinesq Equations for Surface Water Waves	
By N. Lopes, P. Pereira and L. Trabucho	247
30.1 Introduction	247
30.2 Model derivation	249
30.2.1 Standard models	251
30.2.2 Second-order model	253
30.3 Linear dispersion relation	253
30.4 Wave generation	255
30.4.1 Initial condition	255
30.4.2 Incident wave	256
30.4.3 Source function	256
30.5 Reflective walls and sponge layers	257
30.6 Numerical Methods	257
30.7 Numerical Applications	259
30.8 Conclusions and future work	260
30.9 Acknowledgments	262
31 Multiphase Flow Through Porous Media	
By Xuming Shan and Garth N. Wells	263
32 Computing the Mechanics of the Heart	
By Martin S. Alnæs, Kent-Andre Mardal and Joakim Sundnes	265
33 Simulation of Ca²⁺ Dynamics in the Dyadic Cleft	
By Johan Hake	267
33.1 Introduction	267
33.2 Biological background	268
33.3 Mathematical models	268
33.3.1 Geometry	268
33.3.2 Ca ²⁺ Diffusion	269
33.3.3 Stochastic models of single channels	271
33.4 Numerical methods for the continuous system	272
33.4.1 Discretization	274
33.4.2 Stabilization	276
33.5 <code>diffsim</code> an event driven simulator	280
33.5.1 Stochastic system	280
33.5.2 Time stepping algorithm	281

33.5.3	diffsim an example	282
34	Electromagnetic Waveguide Analysis	
	By Evan Lezar and David B. Davidson	289
34.1	Formulation	290
34.1.1	Waveguide Cutoff Analysis	291
34.1.2	Waveguide Dispersion Analysis	293
34.2	Implementation	294
34.2.1	Formulation	294
34.2.2	Post-Processing	296
34.3	Examples	297
34.3.1	Hollow Rectangular Waveguide	298
34.3.2	Half-Loaded Rectangular Waveguide	301
34.3.3	Shielded Microstrip	303
34.4	Analysis of Waveguide Discontinuities	305
34.5	Conclusion	307
35	Applications in Solid Mechanics	
	By Kristian B. Ølgaard and Garth N. Wells	309
36	Modelling Evolving Discontinuities	
	By Mehdi Nikbakht and Garth N. Wells	311
37	Optimal Control Problems	
	By Kent-Andre Mardal, Oddrun Christine Myklebust and Bjørn Fredrik Nielsen	313
38	Automatic Calibration of Depositional Models	
	By Hans Joachim Schroll	315
38.1	Issues in dual lithology sedimentation	315
38.2	A multidimensional sedimentation model	316
38.3	An inverse approach	316
38.4	The Landweber algorithm	317
38.5	Evaluation of gradients by duality arguments	318
38.6	Aspects of the implementation	320
38.7	Numerical experiments	321
38.8	Results and conclusion	323
39	Computational Thermodynamics	
	By Johan Hoffman, Claes Johnson and Murtazo Nazarov	331
39.1	FEniCS as Computational Science	331
39.2	The 1st and 2nd Laws of Thermodynamics	332
39.3	The Enigma	333
39.4	Computational Foundation	335
39.5	Viscosity Solutions	337
39.6	Joule's 1845 Experiment	338

39.7 The Euler Equations	339
39.8 Energy Estimates for Viscosity Solutions	340
39.9 Compression and Expansion	342
39.10A 2nd Law without Entropy	342
39.11 Comparison with Classical Thermodynamics	343
39.12 EG2	344
39.13 The 2nd Law for EG2	345
39.14 The Stabilization in EG2	345
39.15 Output Uniqueness and Stability	345
40 Saddle Point Stability	
By Marie E. Rognes	347
A Notation	357

CHAPTER 1

Introduction

By Anders Logg, Garth N. Wells and Kent-Andre Mardal

Chapter ref: **[intro]**

CHAPTER 2

A FEniCS Tutorial

By Hans Petter Langtangen

Chapter ref: [**langtangen**]

A series of complete, worked out examples, starting with simple mathematical PDE problems, progressing with physics/mechanics problems, and ending up with advanced computational mechanics problems. The essence is to show that the programming tasks scale with the mathematical formulation of the problems.

Part I
Methodology

The Finite Element Method

By Robert C. Kirby and Anders Logg

Chapter ref: **[kirby-7]**

The finite element method has grown out of Galerkin’s method, emerging as a universal method for the solution of differential equations. Much of the success of the finite element method can be contributed to its generality and simplicity, allowing a wide range of differential equations from all areas of science to be analyzed and solved within a common framework. Another contributing factor to the success of the finite element method is the flexibility of formulation, allowing the properties of the discretization to be controlled by the choice of finite element approximating spaces.

In this chapter, we review the finite element method and introduce some basic concepts and notation. In the coming chapters, we discuss these concepts in more detail, with a particular focus on the implementation and automation of the finite element method as part of the FEniCS project.

3.1 A Simple Model Problem

In 1813, Siméon Denis Poisson (Figure 3.1) published in *Bulletin de la société philomatique* his famous equation as a correction of an equation published earlier by Pierre-Simon Laplace. Poisson’s equation is a second-order partial differential equation stating that the negative Laplacian $-\Delta u$ of some unknown field $u = u(x)$ is equal to a given function $f = f(x)$ on a domain $\Omega \subset \mathbb{R}^d$, possibly amended by a set of boundary conditions for the solution u on the boundary $\partial\Omega$



Figure 3.1: Siméon Denis Poisson (1781–1840), inventor of Poisson’s equation.

of Ω :

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega, \\ u &= u_0 && \text{on } \Gamma_D \subset \partial\Omega, \\ -\partial_n u &= g && \text{on } \Gamma_N \subset \partial\Omega. \end{aligned} \tag{3.1}$$

The Dirichlet boundary condition $u = u_0$ signifies a prescribed value for the unknown u on a subset Γ_D of the boundary and the Neumann boundary condition $-\partial_n u = g$ signifies a prescribed value for the (negative) normal derivative of u on the remaining boundary $\Gamma_N = \partial\Omega \setminus \Gamma_D$. Poisson’s equation is a simple model for gravity, electromagnetism, heat transfer, fluid flow, and many other physical processes. It also appears as the basic building block in a large number of more complex physical models, including the Navier–Stokes equations that we return to below in Chapters ??.

To derive Poisson’s equation (3.1), we may consider a model for the temperature u in a body occupying a domain Ω subject to a heat source f . Letting $\sigma = \sigma(x)$ denote heat flux, it follows by conservation of energy that the outflow of energy over the boundary $\partial\omega$ of any test volume $\omega \subset \Omega$ must be balanced by the energy transferred from the heat source f ,

$$\int_{\partial\omega} \sigma \cdot n \, ds = \int_{\omega} f \, dx.$$

Integrating by parts, it follows that

$$\int_{\omega} \nabla \cdot \sigma \, dx = \int_{\omega} f \, dx$$

for all test volumes ω and thus that $\nabla \cdot \sigma = f$ (by suitable regularity assumptions on σ and f). If we now make the assumption that the heat flux σ is proportional

Letting the test function v vanish on the Dirichlet boundary Γ_D where the solution u is known, we arrive at the following classical variational problem: Find $u \in V$ such that

$$\int_{\Omega} \nabla v \cdot \nabla u \, dx = \int_{\Omega} v f \, dx - \int_{\Gamma_N} v g \, ds \quad \forall v \in \hat{V}. \quad (3.3)$$

The test space \hat{V} is defined by

$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_D\},$$

and the trial space V contains members of \hat{V} shifted by the Dirichlet condition,

$$V = \{v \in H^1(\Omega) : v = u_0 \text{ on } \Gamma_D\}.$$

We may now discretize Poisson's equation by restricting the variational problem (3.3) to a pair of discrete spaces: Find $u_h \in V_h \subset V$ such that

$$\int_{\Omega} \nabla v \cdot \nabla u_h \, dx = \int_{\Omega} v f \, dx - \int_{\Gamma_N} v g \, ds \quad \forall v \in \hat{V}_h \subset \hat{V}. \quad (3.4)$$

We note here that the Dirichlet condition $u = u_0$ on Γ_D enters into the definition of the trial space V_h (it is an *essential* boundary condition), whereas the Neumann condition $-\partial_n u = g$ on Γ_N enters into the variational problem (it is a *natural* boundary condition).

To solve the discrete variational problem (3.4), we must construct a suitable pair of discrete test and trial spaces \hat{V}_h and V_h . We return to this issue below, but assume for now that we have a basis $\{\hat{\phi}_i\}_{i=1}^N$ for \hat{V}_h and a basis $\{\phi_j\}_{j=1}^N$ for V_h . We may then make an ansatz for u_h in terms of the basis functions of the trial space,

$$u_h = \sum_{j=1}^N U_j \phi_j,$$

where $U \in \mathbb{R}^N$ is the vector of degrees of freedom to be computed. Inserting this into (3.4) and varying the test function v over the basis functions of the discrete test space \hat{V}_h , we obtain

$$\sum_{j=1}^N U_j \int_{\Omega} \nabla \hat{\phi}_i \cdot \nabla \phi_j \, dx = \int_{\Omega} \hat{\phi}_i f \, dx - \int_{\Gamma_N} \hat{\phi}_i g \, ds, \quad i = 1, 2, \dots, N.$$

We may thus compute the finite element solution $u_h = \sum_{j=1}^N U_j \phi_j$ by solving the linear system

$$AU = b,$$

where

$$A_{ij} = \int_{\Omega} \nabla \hat{\phi}_i \cdot \nabla \phi_j \, dx,$$

$$b_i = \int_{\Omega} \hat{\phi}_i f \, dx - \int_{\Gamma_N} \hat{\phi}_i g \, ds.$$

3.2.2 Discretizing the first order system

We may similarly discretize the first order system (3.2) by multiplying the first equation by a test function v and the second by a test function τ . Summing up and integrating by parts, we find

$$\int_{\Omega} v \nabla \cdot \sigma + \tau \cdot \sigma - \nabla \cdot \tau u \, dx + \int_{\partial\Omega} \tau \cdot n u \, ds = \int_{\Omega} v f \, dx \quad \forall v \in \hat{V}.$$

The normal flux $\sigma \cdot n = g$ is known on the Neumann boundary Γ_N so we may take $\tau \cdot n = 0$ on Γ_N . Inserting the value for u on the Dirichlet boundary Γ_D , we thus arrive at the following variational problem: Find $(u, \sigma) \in V$ such that

$$\int_{\Omega} v \nabla \cdot \sigma + \tau \cdot \sigma - \nabla \cdot \tau u \, dx = \int_{\Omega} v f \, dx - \int_{\Gamma_D} \tau \cdot n u_0 \, ds \quad \forall (v, \tau) \in \hat{V}. \quad (3.5)$$

Now, \hat{V} and V are a pair of suitable test and trial spaces, here

$$\begin{aligned} \hat{V} &= \{(v, \tau) : v \in L^2(\Omega), \tau \in H(\operatorname{div}; \Omega), \tau \cdot n = 0 \text{ on } \Gamma_N\}, \\ V &= \{(v, \tau) : v \in L^2(\Omega), \tau \in H(\operatorname{div}; \Omega), \tau \cdot n = g \text{ on } \Gamma_N\}. \end{aligned}$$

As above, we restrict this variational problem to a pair of discrete test and trial spaces $\hat{V}_h \subset \hat{V}$ and $V_h \subset V$ and make an ansatz for the finite element solution of the form

$$(u_h, \sigma_h) = \sum_{j=1}^N U_j(\phi_j, \psi_j),$$

where $\{(\phi_j, \psi_j)\}_{j=1}^N$ is a basis for the trial space V_h . Typically, either ϕ_j or ψ_j will vanish, so that the basis is really the tensor product of a basis for an L^2 space with an $H(\operatorname{div})$ space. We thus obtain a linear system for the degrees of freedom $U \in \mathbb{R}^N$ by solving a linear system $AU = b$, where now

$$\begin{aligned} A_{ij} &= \int_{\Omega} \hat{\phi}_i \nabla \cdot \psi_j + \hat{\psi}_i \cdot \psi_j - \nabla \cdot \hat{\psi}_i \phi_j \, dx, \\ b_i &= \int_{\Omega} \hat{\phi}_i f \, dx - \int_{\Gamma_D} \hat{\psi}_i \cdot n u_0 \, ds. \end{aligned}$$

We note that the variational problem (3.5) differs from the variational problem (3.3) in that the Dirichlet condition $u = u_0$ on Γ_D enters into the variational formulation (it is now a natural boundary condition), whereas the Neumann condition $\sigma = g$ on Γ_N enters into the definition of the trial space V (it is now an essential boundary condition).

Such mixed methods require some care in selecting spaces that discretize L^2 and $H(\operatorname{div})$ in a compatible way. Stable discretizations must satisfy the so-called *inf-sup* or *Ladysenskaja–Babuška–Brezzi* (LBB) condition. This theory explains why many of the elements for mixed methods seem complicated compared to those for standard Galerkin methods.

3.3 Finite Element Abstract Formalism

3.3.1 Linear problems

We saw above that the finite element solution of Poisson's equation (3.1) or (3.2) can be obtained by restricting an infinite dimensional variational problem to a finite dimensional variational problem and solving a linear system.

To formalize this, we consider a general linear variational problem written in the following canonical form: Find $u \in V$ such that

$$a(v, u) = L(v) \quad \forall v \in \hat{V}, \quad (3.6)$$

where \hat{V} is the test space and V is the trial space. We may thus express the variational problem in terms of a *bilinear form* a and *linear form* (functional) L ,

$$\begin{aligned} a &: \hat{V} \times V \rightarrow \mathbb{R}, \\ L &: \hat{V} \rightarrow \mathbb{R}. \end{aligned}$$

As above, we discretize the variational problem (3.6) by restricting to a pair of discrete test and trial spaces: Find $u_h \in V_h \subset V$ such that

$$a(v, u_h) = L(v) \quad \forall v \in \hat{V}_h \subset \hat{V}. \quad (3.7)$$

To solve the discrete variational problem (3.7), we make an ansatz of the form

$$u_h = \sum_{j=1}^N U_j \phi_j, \quad (3.8)$$

and take $v = \hat{\phi}_i$, $i = 1, 2, \dots, N$, where $\{\hat{\phi}_i\}_{i=1}^N$ is a basis for the discrete test space \hat{V}_h and $\{\phi_j\}_{j=1}^N$ is a basis for the discrete trial space V_h . It follows that

$$\sum_{j=1}^N U_j a(\hat{\phi}_i, \phi_j) = L(\hat{\phi}_i), \quad i = 1, 2, \dots, N.$$

We thus obtain the degrees of freedom U of the finite element solution u_h by solving a linear system $AU = b$, where

$$\begin{aligned} A_{ij} &= a(\hat{\phi}_i, \phi_j), \quad i, j = 1, 2, \dots, N, \\ b_i &= L(\hat{\phi}_i). \end{aligned} \quad (3.9)$$

3.3.2 Nonlinear problems

We also consider nonlinear variational problems written in the following canonical form: Find $u \in V$ such that

$$F(u; v) = 0 \quad \forall v \in \hat{V}, \quad (3.10)$$

where now $F : V \times \hat{V} \rightarrow \mathbb{R}$ is a *semilinear* form, linear in the argument(s) subsequent to the semicolon. As above, we discretize the variational problem (3.10) by restricting to a pair of discrete test and trial spaces: Find $u_h \in V_h \subset V$ such that

$$F(u_h; v) = 0 \quad \forall v \in \hat{V}_h \subset \hat{V}.$$

The finite element solution $u_h = \sum_{j=1}^N U_j \phi_j$ may then be computed by solving a nonlinear system of equations,

$$b(U) = 0, \quad (3.11)$$

where $b : \mathbb{R}^N \rightarrow \mathbb{R}^N$ and

$$b_i(U) = F(u_h; \hat{\phi}_i), \quad i = 1, 2, \dots, N. \quad (3.12)$$

To solve the nonlinear system (3.11) by Newton's method or some variant of Newton's method, we compute the Jacobian $A = b'$. We note that if the semilinear form F is differentiable in u , then the entries of the Jacobian A are given by

$$A_{ij}(u_h) = \frac{\partial b_i(U)}{\partial U_j} = \frac{\partial}{\partial U_j} F(u_h; \hat{\phi}_i) = F'(u_h; \hat{\phi}_i) \frac{\partial u_h}{\partial U_j} = F'(u_h; \hat{\phi}_i) \phi_j \equiv F'(u_h; \hat{\phi}_i, \phi_j). \quad (3.13)$$

In each Newton iteration, we must then evaluate (assemble) the matrix A and the vector b , and update the solution vector U by

$$U^{k+1} = U^k - \delta U^k,$$

where δU^k solves the linear system

$$A(u_h^k) \delta U^k = b(u_h^k). \quad (3.14)$$

We note that for each fixed u_h , $a = F'(u_h; \cdot, \cdot)$ is a bilinear form and $L = F(u_h; \cdot)$ is a linear form. In each Newton iteration, we thus solve a linear variational problem of the canonical form (3.6): Find $\delta u \in V_0$ such that

$$F'(u_h; v, \delta u) = F(u_h; v) \quad \forall v \in \hat{V}, \quad (3.15)$$

where $V_0 = \{v - w : v, w \in V\}$. Discretizing (3.15) as in Section 3.3.1, we recover the linear system (3.14).

Example 3.1 (Nonlinear Poisson equation). *As an example, consider the following nonlinear Poisson equation:*

$$\begin{aligned} -\nabla \cdot ((1+u)\nabla u) &= f && \text{in } \Omega, \\ u &= 0 && \text{on } \partial\Omega. \end{aligned} \tag{3.16}$$

Multiplying (3.16) with a test function v and integrating by parts, we obtain

$$\int_{\Omega} \nabla v \cdot ((1+u)\nabla u) \, dx = \int_{\Omega} v f \, dx,$$

which is a nonlinear variational problem of the form (3.10), with

$$F(u; v) = \int_{\Omega} \nabla v \cdot ((1+u)\nabla u) \, dx - \int_{\Omega} v f \, dx.$$

Linearizing the semilinear form F around $u = u_h$, we obtain

$$F'(u_h; v, \delta u) = \int_{\Omega} \nabla v \cdot (\delta u \nabla u_h) \, dx + \int_{\Omega} \nabla v \cdot ((1+u_h)\nabla \delta u) \, dx.$$

We may thus compute the entries of the Jacobian matrix $A(u_h)$ by

$$A_{ij}(u_h) = F'(u_h; \hat{\phi}_i, \phi_j) = \int_{\Omega} \nabla \hat{\phi}_i \cdot (\phi_j \nabla u_h) \, dx + \int_{\Omega} \nabla \hat{\phi}_i \cdot ((1+u_h)\nabla \phi_j) \, dx. \tag{3.17}$$

3.4 Finite Element Function Spaces

In the above discussion, we assumed that we could construct discrete subspaces $V_h \subset V$ of infinite dimensional function spaces. A central aspect of the finite element method is the construction of such subspaces by patching together local function spaces defined on a set of *finite elements*. We here give a general overview of the construction of finite element function spaces and return below in Chapters 4 and 5 to the construction of specific function spaces such as subsets of $H^1(\Omega)$, $H(\text{curl})$, $H(\text{div})$ and $L^2(\Omega)$.

3.4.1 The mesh

To define V_h , we first partition the domain Ω into a finite set of disjoint cells $\mathcal{T} = \{K\}$ such that

$$\cup_{K \in \mathcal{T}} K = \Omega.$$

Together, these cells form a *mesh* of the domain Ω . The cells are typically simple polygonal shapes like intervals, triangles, quadrilaterals, tetrahedra or hexahedra as shown in Figure 3.3. But other shapes are possible, in particular curved cells to correctly capture the boundary of a non-polygonal domain as shown in Figure 3.4.

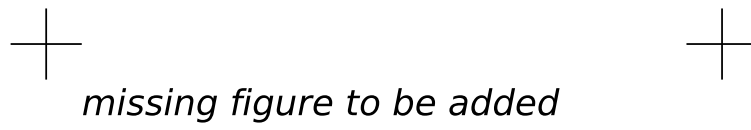


Figure 3.3: Finite element cells in one, two and three space dimensions.

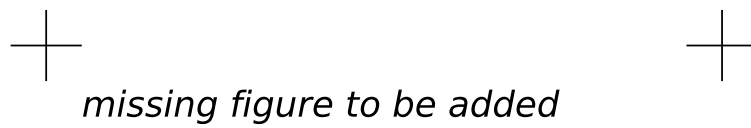


Figure 3.4: A straight triangular cell (left) and curved triangular cell (right).

3.4.2 The finite element definition

Once a domain Ω has been partitioned into cells, one may define a local function space \mathcal{P}_K on each cell K and use these local function spaces to build the global function space V_h . A cell K together with a local function space \mathcal{P}_K and a set of rules for describing functions in \mathcal{P}_K is called a *finite element*. This definition of finite element was first formalized by Ciarlet in [Cia78, Cia02], and it remains the standard formulation today. [BS94, BS08]. The formal definition reads as follows: A finite element is a triple $(K, \mathcal{P}_K, \mathcal{L}_K)$, where

- $K \subset \mathbb{R}^d$ is a bounded closed subset of \mathbb{R}^d with nonempty interior and piecewise smooth boundary;
- \mathcal{P}_K is a function space on K of dimension $n_K < \infty$;
- $\mathcal{L}_K = \{\ell_1^K, \ell_2^K, \dots, \ell_{n_K}^K\}$ is a basis for \mathcal{P}'_K (the bounded linear functionals on \mathcal{P}_K).

As an example, consider the standard linear Lagrange finite element on the triangle in Figure 3.5. The cell K is given by the triangle and the space \mathcal{P}_K is given by the space of first degree polynomials on K . As a basis for \mathcal{P}'_K , we may take point evaluation at the three vertices of K , that is,

$$\begin{aligned} \ell_i^K &: \mathcal{P}_K \rightarrow \mathbb{R}, \\ \ell_i^K(v) &= v(x^i), \end{aligned}$$

for $i = 1, 2, 3$ where x^i is the coordinate of the i th vertex. To check that this is indeed a finite element, we need to verify that \mathcal{L}_K is a basis for \mathcal{P}'_K . This is equivalent to the unisolvence of \mathcal{L}_K , that is, if $v \in \mathcal{P}_K$ and $\ell_i^K(v) = 0$ for all ℓ_i^K , then $v = 0$. [BS08] For the linear Lagrange triangle, we note that if v is zero at each vertex, then v must be zero everywhere, since a plane is uniquely determined by its value at three non-collinear points. Thus, the linear Lagrange triangle is indeed a finite element. In general, determining the unisolvence of \mathcal{L}_K may be non-trivial.

3.4.3 The nodal basis

Expressing finite element solutions in V_h in terms of basis functions for the local function spaces \mathcal{P}_K may be greatly simplified by introducing a *nodal basis* for \mathcal{P}_K . A nodal basis $\{\phi_i^K\}_{i=1}^{n_K}$ for \mathcal{P}_K is a basis for \mathcal{P}_K that satisfies

$$\ell_i^K(\phi_j^K) = \delta_{ij}, \quad i, j = 1, 2, \dots, n_K. \quad (3.18)$$

It follows that any $v \in \mathcal{P}_K$ may be expressed by

$$v = \sum_{i=1}^{n_K} \ell_i^K(v) \phi_i^K. \quad (3.19)$$

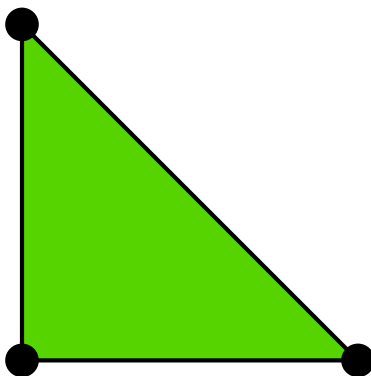


Figure 3.5: The linear Lagrange (Courant) triangle).

In particular, any function v in \mathcal{P}_K for the linear Lagrange triangle is given by $v = \sum_{i=1}^3 v(x^i) \phi_i^K$. In other words, the degrees of freedom of any function v may be obtained by evaluating the linear functionals \mathcal{L}_K . We shall therefore sometimes refer to \mathcal{L}_K as degrees of freedom.

► **Author note:** Give explicit formulas for some nodal basis functions. Use example environment.

For any finite element $(K, \mathcal{P}_K, \mathcal{L}_K)$, the nodal basis may be computed by solving a linear system of size $n_K \times n_K$. To see this, let $\{\psi_i^K\}_{i=1}^{n_K}$ be any basis (the *prime* basis) for \mathcal{P}_K . Such a basis is easy to construct if \mathcal{P}_K is a full polynomial space or may otherwise be computed by a singular-value decomposition or a Gram-Schmidt procedure, see [Kir04]. We may then make an ansatz for the nodal basis in terms of the prime basis:

$$\phi_j = \sum_{k=1}^{n_K} \alpha_{jk} \psi_k^K, \quad j = 1, 2, \dots, n_K.$$

Inserting this into (3.18), we find that

$$\sum_{k=1}^{n_K} \alpha_{jk} \ell_i^K(\psi_k^K) = \delta_{ij}, \quad j = 1, 2, \dots, n_K.$$

In other words, the expansion coefficients α for the nodal basis may be computed by solving the linear system

$$B\alpha^\top = I,$$

where $B_{ij} = \ell_i^K(\psi_j^K)$.

3.4.4 The local-to-global mapping

Now, to define a global function space $V_h = \text{span}\{\phi_i\}_{i=1}^N$ on Ω from a given set $\{(K, \mathcal{P}_K, \mathcal{L}_K)\}_{K \in \mathcal{T}}$ of finite elements, we also need to specify how the local function

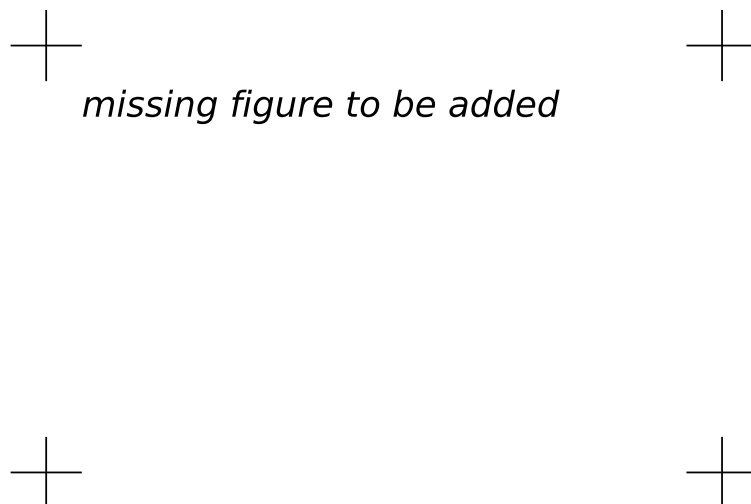


Figure 3.6: Local-to-global mapping for a simple mesh consisting of two triangles.

spaces are patched together. We do this by specifying for each cell $K \in \mathcal{T}$ a *local-to-global mapping*,

$$\iota_K : [1, n_K] \rightarrow N. \quad (3.20)$$

This mapping specifies how the local degrees of freedom $\mathcal{L}_K = \{\ell_i^K\}_{i=1}^{n_K}$ are mapped to global degrees of freedom $\mathcal{L} = \{\ell_i\}_{i=1}^N$. More precisely, the global degrees of freedom are given by

$$\ell_{\iota_K(i)}(v) = \ell_i^K(v|_K), \quad i = 1, 2, \dots, n_K, \quad (3.21)$$

for any $v \in V_h$. Thus, each local degree of freedom $\ell_i^K \in \mathcal{L}_K$ corresponds to a global degree of freedom $\ell_{\iota_K(i)} \in \mathcal{L}$ determined by the local-to-global mapping ι_K . As we shall see, the local-to-global mapping together with the choice of degrees of freedom determine the continuity of the global function space V_h .

For standard piecewise linears, one may define the local-to-global mapping by simply mapping each local vertex number i for $i = 1, 2, 3$ to the corresponding global vertex number $\iota_K(i)$. This is illustrated in Figure 3.6 for a simple mesh consisting of two triangles.

3.4.5 The global function space

One may now define the global function space V_h as the set of functions on Ω satisfying the following pair of conditions. We first require that

$$v|_K \in \mathcal{P}_K \quad \forall K \in \mathcal{T}, \quad (3.22)$$

that is, the restriction of v to each cell K lies in the local function space \mathcal{P}_K . Second, we require that for any pair of cells $(K, K') \in \mathcal{T} \times \mathcal{T}$ and any pair $(i, i') \in$

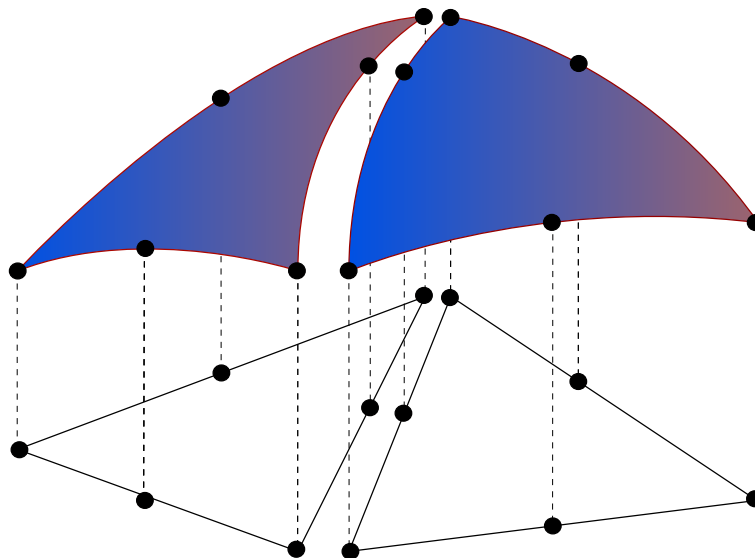


Figure 3.7: Patching together local function spaces on a pair of cells (K, K') to form a global function space on $\Omega = K \cup K'$.

$[1, n_K] \times [1, n_{K'}]$ satisfying

$$\iota_K(i) = \iota_{K'}(i'), \quad (3.23)$$

it holds that

$$\ell_i^K(v|_K) = \ell_{i'}^{K'}(v|_{K'}). \quad (3.24)$$

In other words, if two local degrees of freedom ℓ_i^K and $\ell_{i'}^{K'}$ are mapped to the same global degree of freedom, then they must agree for each function $v \in V_h$. Here, $v|_K$ denotes the continuous extension to K of the restriction of v to the interior of K . This is illustrated in Figure 3.7 for the space of continuous piecewise quadratics obtained by patching together two quadratic Lagrange triangles.

Note that by this construction, the functions of V_h are undefined on cell boundaries, unless the constraints (3.24) force the (restrictions of) functions of V_h to be continuous on cell boundaries. However, this is usually not a problem, since we can perform all operations on the restrictions of functions to the local cells.

The local-to-global mapping together with the choice of degrees of freedom determine the continuity of the global function space V_h . For the Lagrange triangle, the choice of degrees of freedom as point evaluation at vertices ensures that the restrictions $v|_K$ and $v|_{K'}$ of a function $v \in V_h$ to a pair of adjacent triangles K agree at the two common vertices, since ι_K and $\iota_{K'}$ map corresponding degrees of freedom to the same global degree of freedom and this global degree of freedom is single-valued. It follows that the functions of V_h are continuous not only at vertices but along each shared edge since a first-degree polynomial on a line is uniquely determined by its values at two distinct points. Thus, the global function space of piecewise linears generated by the Lagrange triangle is continuous



Figure 3.9: The (affine) mapping F_K from a reference cell \hat{K} to some cell $K \in \mathcal{T}$.

For function spaces discretizing H^1 as in (3.3), the mapping F_K is typically *affine*, that is, F_K can be written in the form $F_K(\hat{x}) = A_K \hat{x} + b_K$ for some matrix $A_K \in \mathbb{R}^{d \times d}$ and some vector $b_K \in \mathbb{R}^d$, or *isoparametric*, in which case the components of F_K are functions in $\hat{\mathcal{P}}$. For function spaces discretizing $H(\text{div})$ like in (3.5) or $H(\text{curl})$, the appropriate mappings are the contravariant and covariant Piola mappings which preserve normal and tangential components respectively, see [RKL08]. For simplicity, we restrict the following discussion to the case when F_K is affine or isoparametric.

For each cell $K \in \mathcal{T}$, the mapping F_K generates a function space on K given by

$$\mathcal{P}_K = \{v : v = \hat{v} \circ F_K^{-1}, \hat{v} \in \hat{\mathcal{P}}\}, \quad (3.26)$$

that is, each function $v = v(x)$ may be expressed as $v(x) = \hat{v}(F_K^{-1}(x)) = \hat{v} \circ F_K^{-1}(x)$ for some $\hat{v} \in \hat{\mathcal{P}}$.

The mapping F_K also generates a set of degrees of freedom \mathcal{L}_K on \mathcal{P}_K given by

$$\mathcal{L}_K = \{\ell_i^K : \ell_i^K(v) = \hat{\ell}_i(v \circ F_K), \quad i = 1, 2, \dots, \hat{n}\}. \quad (3.27)$$

The mappings $\{F_K\}_{K \in \mathcal{T}}$ thus generate from the reference finite element $(\hat{K}, \hat{\mathcal{P}}, \hat{\mathcal{L}})$ a set of finite elements $\{(K, \mathcal{P}_K, \mathcal{L}_K)\}_{K \in \mathcal{T}}$ given by

$$\begin{aligned} K &= F_K(\hat{K}), \\ \mathcal{P}_K &= \{v : v = \hat{v} \circ F_K^{-1} : \hat{v} \in \hat{\mathcal{P}}\}, \\ \mathcal{L}_K &= \{\ell_i^K : \ell_i^K(v) = \hat{\ell}_i(v \circ F_K), \quad i = 1, 2, \dots, \hat{n} = n_K\}. \end{aligned} \quad (3.28)$$

By this construction, we also obtain the nodal basis functions $\{\phi_i^K\}_{i=1}^{n_K}$ on K from a set of nodal basis functions $\{\hat{\phi}_i\}_{i=1}^{\hat{n}}$ on the reference element satisfying $\hat{\ell}_i(\hat{\phi}_j) = \delta_{ij}$.

Letting $\phi_i^K = \hat{\phi}_i \circ F_K^{-1}$ for $i = 1, 2, \dots, n_K$, we find that

$$\ell_i^K(\phi_j^K) = \hat{\ell}_i(\phi_j^K \circ F_K) = \hat{\ell}_i(\hat{\phi}_j) = \delta_{ij}, \quad (3.29)$$

so $\{\phi_i^K\}_{i=1}^{n_K}$ is a nodal basis for \mathcal{P}_K .

We may thus define the function space V_h by specifying a mesh \mathcal{T} , a reference finite element $(\hat{K}, \hat{\mathcal{P}}, \hat{\mathcal{L}})$, a set of local-to-global mappings $\{\iota_K\}_{K \in \mathcal{T}}$ and a set of mappings $\{F_K\}_{K \in \mathcal{T}}$ from the reference cell \hat{K} . Note that in general, the mappings need not be of the same type for all cells K and not all finite elements need to be generated from the same reference finite element. In particular, one could employ a different (higher-degree) isoparametric mapping for cells on a curved boundary.

The above construction is valid for so-called affine-equivalent elements [BS08] like the family H^1 -conforming Lagrange finite elements. A similar construction is possible for $H(\text{div})$ - and $H(\text{curl})$ conforming elements, like the Raviart–Thomas, Brezzi–Douglas–Marini and Nédélec elements, where an appropriate Piola mapping must be used to map the basis functions. However, not all finite elements may be generated from a reference finite element using this simple construction. For example, this construction fails for the family of Hermite finite elements. [Cia02, BS08].

3.5 Finite Element Solvers

Finite elements provide a powerful methodology for discretizing differential equations, but solving the resulting algebraic systems also presents quite a challenge, even for linear systems. Good solvers must handle the sparsity and ill-conditioning of the algebraic system, but also scale well on parallel computers. The linear solve is a fundamental operation not only in linear problems, but also within each iteration of a nonlinear solve via Newton’s method, an eigenvalue solve, or time-stepping.

A classical approach that has been revived recently is direct solution, based on Gaussian elimination. Thanks to techniques enabling parallel scalability and recognizing block structure, packages such as UMFPACK [Dav04] and SuperLU [Li05] have made direct methods competitive for quite large problems.

The 1970s and 1980s saw the advent of modern iterative methods. These grew out of classical iterative methods such as relaxation methods [?] and the conjugate gradient iteration of Hestenes and Stieffel [HS52]. These techniques can use much less memory than direct methods and are easier to parallelize.

► Author note: *Missing reference for relaxation methods*

Multigrid methods [Bra77, Wes92] use relaxation techniques on a hierarchy of meshes to solve elliptic equations, typically for symmetric problems, in nearly linear time. However, they require a hierarchy of meshes that may not always be

available. This motivated the introduction of *algebraic* multigrid methods (AMG) that mimic mesh coarsening, working only on the matrix entries. Successful AMG distributions include the Hypra package [FY02] and the ML package inside Trilinos [HBH⁺05].

Krylov methods such as conjugate gradients and GMRES [SS86] generate a sequence of approximations converging to the solution of the linear system. These methods are based only on the matrix–vector product. The performance of these methods is significantly improved by use of *preconditioners*, which transform the linear system

$$AU = b$$

into

$$P^{-1}AU = P^{-1}b,$$

which is known as left preconditioning. The preconditioner P^{-1} may also be applied from the right by recognizing that $AU = AP^{-1}(PU)$. To ensure good convergence, the preconditioner P^{-1} should be a good approximation of A^{-1} . Some preconditioners are strictly algebraic, meaning they only use information available from the entries of A . Classical relaxation methods such as Gauss–Seidel may be used as preconditioners, as can so-called incomplete factorizations [?]. If multigrid or AMG is available, it also can serve as a powerful preconditioner. Other kinds of preconditioners require special knowledge about the differential equations being solved and may require new matrices modeling related physical processes. Such methods are sometimes called *physics-based* preconditioners [?]. An automated system, such as FEniCS, provides an interesting opportunity to assist with the development and implementation of these powerful but less widely used methods.

► Author note: *Missing reference for incomplete LU factorization and physics-based preconditioners*

Fortunately, many of the methods discussed here are included in modern libraries such as PETSc [BBE⁺04] and Trilinos [HBH⁺05]. FEniCS typically interacts with the solvers discussed here through these packages and so mainly need to be aware of the various methods at a high level, such as when the various methods are appropriate and how to access them.

3.6 Finite Element Error Estimation and Adaptivity

The error $e = u_h - u$ in a computed finite element solution u_h approximating the exact solution u of (3.6) may be estimated either *a priori* or *a posteriori*. Both types of estimates are based on relating the size of the error to the size of the (weak) residual $r : V \rightarrow \mathbb{R}$ defined by

$$r(v) = a(v, u_h) - L(v). \tag{3.30}$$

We note that the weak residual is formally related to the strong residual $R \in V'$ by $r(v) = (v, R)$.

A priori error estimates express the error in terms of the regularity of the exact (unknown) solution and may give useful information about the order of convergence of a finite element method. A posteriori error estimates express the error in terms of computable quantities like the residual and (possibly) the solution of an auxiliary dual problem.

3.6.1 A priori error analysis

We consider the linear variational problem (3.6). We first assume that the bilinear form a and the linear form L are continuous (bounded), that is, there exists a constant $C > 0$ such that

$$a(v, w) \leq C \|v\|_V \|w\|_V, \quad (3.31)$$

$$L(v) \leq C \|v\|_V, \quad (3.32)$$

for all $v, w \in V$. For simplicity, we assume in this section that $\hat{V} = V$ is a Hilbert space. For (3.1), this corresponds to the case of homogeneous Dirichlet boundary conditions and $V = H_0^1(\Omega)$. Extensions to the general case $\hat{V} \neq V$ are possible, see for example [OD96]. We further assume that the bilinear form a is coercive (V -elliptic), that is, there exists a constant $\alpha > 0$ such that

$$a(v, v) \geq \alpha \|v\|_V^2, \quad (3.33)$$

for all $v \in V$. It then follows by the Lax–Milgram theorem [LM54] that there exists a unique solution $u \in V$ to the variational problem (3.6).

To derive an a priori error estimate for the approximate solution u_h defined by the discrete variational problem (3.7), we first note that

$$a(v, u_h - u) = a(v, u_h) - a(v, u) = L(v) - L(v) = 0$$

for all $v \in V_h \subset V$. By the coercivity and continuity of the bilinear form a , we find that

$$\begin{aligned} \alpha \|u_h - u\|_V^2 &\leq a(u_h - u, u_h - u) = a(u_h - v, u_h - u) + a(v - u, u_h - u) \\ &= a(v - u, u_h - u) \leq C \|v - u\|_V \|u_h - u\|_V. \end{aligned}$$

for all $v \in V_h$. It follows that

$$\|u_h - u\|_V \leq \frac{C}{\alpha} \|v - u\|_V \quad \forall v \in V_h. \quad (3.34)$$

The estimate (3.34) is referred to as Cea's lemma. We note that when the bilinear form a is symmetric, it is also an inner product. We may then take $\|v\|_V =$



Figure 3.10: The finite element solution $u_h \in V_h \subset V$ is the a -projection of $u \in V$ onto the subspace V_h and is consequently the best possible approximation of u in the subspace V_h .

$\sqrt{a(v, v)}$ and $C = \alpha = 1$. In this case, u_h is the a -projection onto V_h and Cea's lemma states that

$$\|u_h - u\|_V \leq \|v - u\|_V \quad \forall v \in V_h, \quad (3.35)$$

that is, u_h is the best possible solution of the variational problem (3.6) in the subspace V_h . This is illustrated in Figure 3.10.

Cea's lemma together with a suitable interpolation estimate now yields the a priori error estimate for u_h . By choosing $v = \pi_h u$, where $\pi_h : V \rightarrow V_h$ is an interpolation operator into V_h , we find that

$$\|u_h - u\|_V \leq \frac{C}{\alpha} \|\pi_h u - u\|_V \leq \frac{CC_i}{\alpha} \|h^p D^q u\|, \quad (3.36)$$

where C_i is an interpolation constant and the values of p and q depend on the accuracy of interpolation and the definition of $\|\cdot\|_V$. For the solution of Poisson's equation in H_0^1 , we have $C = \alpha = 1$ and $p = q = 1$.

3.6.2 A posteriori error analysis

Energy norm error estimates

The continuity and coercivity of the bilinear form a also allows a simple derivation of an a posteriori error estimate. In fact, it follows that the V -norm of the error $e = u_h - u$ is equivalent to the V' -norm of the residual r . To see this, we note that by the continuity of the bilinear form a , we have

$$r(v) = a(v, u_h) - L(v) = a(v, u_h) - a(v, u) = a(v, u_h - u) \leq C \|u_h - u\|_V \|v\|_V.$$

Furthermore, by coercivity, we find that

$$\alpha \|u_h - u\|^2 \leq a(u_h - u, u_h - u) = a(u_h - u, u_h) - L(u_h - u) = r(u_h - u).$$

It follows that

$$\alpha \|u_h - u\|_V \leq \|r\|_{V'} \leq C \|u_h - u\|_V, \quad (3.37)$$

where $\|r\|_{V'} = \sup_{v \in V, v \neq 0} r(v) / \|v\|_V$.

The estimates (3.36) and (3.37) are sometimes referred to as *energy norm* error estimates. This is the case when the bilinear form a is symmetric and thus defines an inner product. One may then take $\|v\|_V = \sqrt{a(v, v)}$ and $C = \alpha = 1$. In this case, it follows that

$$\|e\|_V = \|r\|_{V'}. \quad (3.38)$$

The term energy norm refers to $a(v, v)$ corresponding to physical energy in many applications.

Duality-based error control

The classical a priori and a posteriori error estimates (3.36) and (3.37) relate the V -norm of the error $e = u_h - u$ to the regularity of the exact solution u and the residual $r = a(v, u_h) - L(v)$ of the finite element solution u_h respectively. However, in applications it is often necessary to control the error in a certain *output functional* $\mathcal{M} : V \rightarrow \mathbb{R}$ of the computed solution to within some given tolerance $\text{TOL} > 0$. In these situations, one would thus ideally like to choose the finite element space $V_h \subset V$ such that the finite element solution u_h satisfies

$$|\mathcal{M}(u_h) - \mathcal{M}(u)| \leq \text{TOL} \quad (3.39)$$

with minimal computational work. We assume here that both the output functional and the variational problem are linear, but the analysis may be easily extended to the full nonlinear case, see [EEHJ95, BR01].

To estimate the error in the output functional \mathcal{M} , we introduce an auxiliary *dual* problem: Find $z \in V^*$ such that

$$a^*(v, z) = \mathcal{M}(v) \quad \forall v \in \hat{V}^*. \quad (3.40)$$

We note here that the functional \mathcal{M} enters as data in the dual problem. The dual (adjoint) bilinear form $a^* : \hat{V}^* \times V^*$ is defined by

$$a^*(v, w) = a(w, v).$$

The dual trial and test spaces are given by

$$\begin{aligned} V^* &= \hat{V}, \\ \hat{V}^* &= V_0 = \{v - w : v, w \in V\}, \end{aligned}$$

that is, the dual trial space is the primal test space and the dual test space is the primal trial space modulo boundary conditions. In particular, if $V = u_0 + \hat{V}$ and $V_h = u_0 + \hat{V}_h$ then $\hat{V}^* = \hat{V}$, and thus both the dual test and trial functions vanish at Dirichlet boundaries. The definition of the dual problem leads us to the following representation of the error:

$$\begin{aligned} \mathcal{M}(u_h) - \mathcal{M}(u) &= \mathcal{M}(u_h - u) \\ &= a^*(u_h - u, z) \\ &= a(z, u_h - u) \\ &= a(z, u_h) - L(z) \\ &= r(z). \end{aligned}$$

We thus find that the error is exactly represented by the residual applied to the dual solution,

$$\mathcal{M}(u_h) - \mathcal{M}(u) = r(z). \quad (3.41)$$

3.6.3 *Adaptivity*

As seen above, one may thus estimate the error in a computed finite element solution u_h , either the error in the V -norm or the error in an output functional, by estimating the size of the residual r . This may be done in several different ways. The estimate typically involves integration by parts to recover the strong element-wise residual of the original PDE, possibly in combination with the solution of local problems over cells or patches of cells. In the case of the standard piecewise linear finite element approximation of Poisson's equation (3.1), one may obtain the following estimate:

$$\|u_h - u\|_V = \|\nabla e\| \leq C \left(\sum_{K \in \mathcal{T}} h_K^2 \|R\|_K^2 + h_K \|[\partial_n u_h]\|_{\partial K}^2 \right)^{1/2},$$

where $R|_K = -\Delta u_h|_K - f|_K$ is the strong residual, h_K denotes the mesh size (diameter of smallest circumscribed sphere) and $[\partial_n u_h]$ denotes the jump of the normal derivative across mesh facets. Letting $\eta_K^2 = h_K^2 \|R\|_K^2 + h_K \|[\partial_n u_h]\|_{\partial K}^2$, one thus obtains the estimate

$$\|u_h - u\|_V \leq E \equiv \left(C \sum_K \eta_K^2 \right)^{1/2}.$$

An adaptive algorithm seeks to determine a mesh size $h = h(x)$ such that $E \leq \text{TOL}$. Starting from an initial coarse mesh, the mesh is successively refined in those cells where the error indicator η_K is large. Several strategies are available, such as refining the top fraction of all cells where η_K is large, say the first 20%

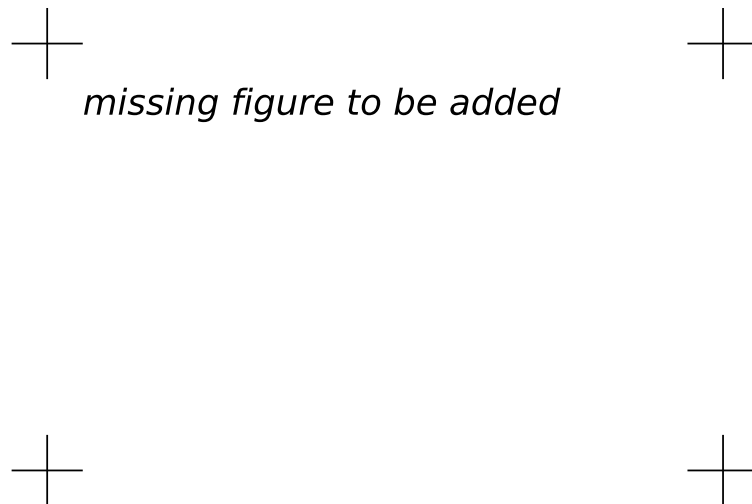


Figure 3.11: An adaptively refined mesh obtained by successive refinement of an original coarse mesh.

of all cells ordered by η_K . Other strategies include refining all cells where η_K is above a certain fraction of $\max_{K \in \mathcal{T}} \eta_K$, or refining a top fraction of all cells such that the sum of their error indicators account for a significant fraction of E .

► Author note: *Find good reference for adaptive strategies.*

Once the mesh has been refined, a new solution and new error indicators can be computed. The process is then repeated until either $E \leq \text{TOL}$ (the stopping criterion) or the available resources (CPU time and memory) have been exhausted. The adaptive algorithm thus yields a sequence of successively refined meshes as illustrated in Figure 3.11. For time-dependent problems, an adaptive algorithm needs to distribute both the mesh size and the size of the time step in both space and time. Ideally, the error estimate E is close to the actual error, as measured by the *efficiency index* $E/\|u_h - u\|_V$ which should be close to one by bounded below by one.

3.7 Automating the Finite Element Method

The FEniCS project seeks to automate Scientific Computing as explained in Chapter [intro]. This is a formidable task, but it may be solved in part by automating the finite element method. In particular, this automation relies on the following key steps:

- (i) automation of discretization,
- (ii) automation of discrete solution,

(iii) automation of error control.

Since its inception in 2003, the FEniCS project has been concerned mainly with the automation of discretization, resulting in the development of the form compilers FFC and SyFi/SFC, the code generation interface UFC and the form language UFL. As a result, the first step towards a complete automation is now close to complete; variational problems for a large class of partial differential equations may now be automatically discretized by the finite element method using FEniCS. For the automation of discrete solution, that is, the solution of linear and nonlinear systems arising from the automated discretization of variational problems, interfaces to state-of-the-art libraries for linear algebra have been implemented as part of DOLFIN. Ongoing work is now seeking to automate error control by automated error estimation and adaptivity as part of FEniCS.

3.8 Outlook

In the following chapters, we return to specific aspects of the automation of the finite element method. In the next chapter, we review a number of common and unusual finite elements, including the standard Lagrange elements but also some more exotic elements. In Chapter 5, we then discuss the automated generation of finite element nodal basis functions from a given finite element definition $(K, \mathcal{P}_K, \mathcal{L}_K)$. In Chapter 6, we consider general finite element variational forms arising from the discretization of PDEs and discuss the automated assembly of the corresponding discrete operators in Chapter 7. We then discuss specific optimization strategies for form evaluation in Chapters ??–??.

► Author note: *This section needs to be reworked when the following chapters have materialized.*

3.9 Historical Notes

In 1915, Boris Grigoryevich Galerkin formulated a general method for solving differential equations. [Gal15] A similar approach was presented sometime earlier by Bubnov. Galerkin’s method, or the Bubnov–Galerkin method, was originally formulated with global polynomials and goes back to the variational principles of Leibniz, Euler, Lagrange, Dirichlet, Hamilton, Castigliano [Cas79], Rayleigh [Ray70] and Ritz [Rit08]. Galerkin’s method with piecewise polynomial spaces (\hat{V}_h, V_h) is known as the *finite element method*. The finite element method was introduced by engineers for structural analysis in the 1950s and was independently proposed by Courant in 1943 [Cou43]. The exploitation of the finite element method among engineers and mathematicians exploded in the 1960s. Since then, the machinery of the finite element method has been expanded and refined into a comprehensive framework for design and analysis of



Figure 3.12: Boris Galerkin (1871–1945), inventor of Galerkin’s method.

numerical methods for differential equations, see [ZTZ67, SF73, Cia76, Cia78, BCO81, Hug87, BS94] Recently, the quest for compatible (stable) discretizations of mixed variational problems has led to the introduction of finite element exterior calculus. [AFW06a]

Work on a posteriori error analysis of finite element methods dates back to the pioneering work of Babuška and Rheinboldt. [BR78]. Important references include the works [BW85, ZZ87, EJ91, EJ95a, EJ, EJ95b, EJ95c, EJJ98, AO93] and the reviews papers [EEHJ95, Ver94, Ver99, AO00, BR01].

► Author note: *Need to check for missing/inaccurate references here.*

► Editor note: *Might use a special box/layout for historical notes if they appear in many places.*

Common and Unusual Finite Elements

By Robert C. Kirby, Anders Logg and Andy R. Terrel

Chapter ref: **[kirby-6]**

This chapter provides a glimpse of the considerable range of finite elements in the literature and the challenges that may be involved with automating “all” the elements. Many of the elements presented here are included in the FEniCS project already; some are future work.

4.1 Ciarlet’s Finite Element Definition

As discussed in Chapter 3, a finite element is defined by a triple $(K, \mathcal{P}_K, \mathcal{L}_K)$, where

- $K \subset \mathbb{R}^d$ is a bounded closed subset of \mathbb{R}^d with nonempty interior and piecewise smooth boundary;
- \mathcal{P}_K is a function space on K of dimension $n_K < \infty$;
- $\mathcal{L}_K = \{\ell_1^K, \ell_2^K, \dots, \ell_{n_K}^K\}$ is a basis for \mathcal{P}'_K (the bounded linear functionals on \mathcal{P}_K).

This definition was first introduced by Ciarlet in a set of lecture notes [Cia75] and became popular after his 1978 book [Cia78, Cia02]. It remains the standard definition today, see for example [BS08]. Similar ideas were introduced earlier in [CR72] which discusses unisolvence of a set of interpolation points $\Sigma = \{a_i\}_i$. This is closely related to the unisolvence of \mathcal{L}_K . In fact, the set of functionals \mathcal{L}_K is given by $\ell_i^K(v) = v(a_i)$. It is also interesting to note that the Ciarlet triple was

originally written as (K, P, Σ) with Σ denoting \mathcal{L}_K . Conditions for uniquely determining a polynomial based on interpolation of function values and derivatives at a set of points was also discussed in [?], although the term unisolvence was not used.

4.2 Notation

It is common to refer to the space of linear functionals \mathcal{L}_K as the *degrees of freedom* of the element $(K, \mathcal{P}_K, \mathcal{L}_K)$. The degrees of freedom are typically given by point evaluation or moments of function values or derivatives. Other commonly used degrees of freedom are point evaluation or moments of certain components of function values, such as normal or tangential components, but also directional derivatives. We summarize the notation used to indicate degrees of freedom graphically in Figure 4.1. A filled circle at a point \bar{x} denotes point evaluation at that point,

$$\ell(v) = v(\bar{x}).$$

We note that for a vector valued function v with d components, a filled circle denotes evaluation of all components and thus corresponds to d degrees of freedom,

$$\begin{aligned}\ell_1(v) &= v_1(\bar{x}), \\ \ell_2(v) &= v_2(\bar{x}), \\ \ell_3(v) &= v_3(\bar{x}).\end{aligned}$$

An arrow denotes evaluation of a component of a function value in a given direction, such as a normal component $\ell(v) = v(\bar{x}) \cdot n$ or tangential component $\ell(v) = v(\bar{x}) \cdot t$. A plain circle denotes evaluation of all first derivatives, a line denotes evaluation of a directional first derivative such as a normal derivative $\ell(v) = \nabla v(\bar{x}) \cdot n$. A dotted circle denotes evaluation of all second derivatives. Finally, a circle with a number indicates a number of interior moments (integration against functions over the domain K).

●	<i>point evaluation</i>
→	<i>point evaluation of directional component</i>
○	<i>point evaluation of all first derivatives</i>
/	<i>point evaluation of directional derivative</i>
⊙	<i>point evaluation of all second derivatives</i>
③	<i>interior moments</i>

Figure 4.1: Notation

4.3 The Argyris Element

4.3.1 Definition

The Argyris triangle [AFS68, Cia02] is based on the space $\mathcal{P}_K = P_5(K)$ of quintic polynomials over some triangle K . It can be pieced together with full C^1 continuity between elements with C^2 continuity at the vertices of a triangulation. Quintic polynomials in \mathbb{R}^2 are a 21-dimensional space, and the dual basis \mathcal{L}_K consists of six degrees of freedom per vertex and one per each edge. The vertex degrees of freedom are the function value, two first derivatives to specify the gradient, and three second derivatives to specify the unique components of the (symmetric) Hessian matrix.

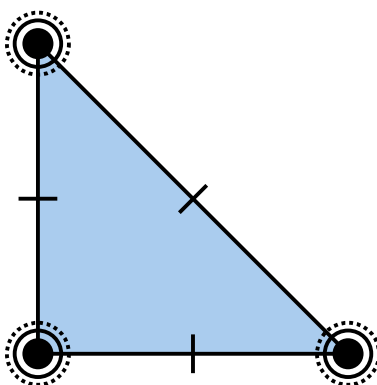


Figure 4.2: The quintic Argyris triangle.

4.3.2 Historical notes

The Argyris element [AFS68] was first called the TUBA element and was applied to fourth-order plate-bending problems. In fact, as Ciarlet points out [Cia02], the element also appeared in an earlier work by Felippa [Fel66].

The normal derivatives in the dual basis for the Argyris element prevent it from being affine-interpolation equivalent. This prevents the nodal basis from being constructed on a reference cell and affinely mapped. Recent work by Dominguez and Sayas [?] has developed a transformation that corrects this issue and requires less computational effort than directly forming the basis on each cell in a mesh.

The Argyris element can be generalized to polynomial degrees higher than quintic, still giving C^1 continuity with C^2 continuity at the vertices [?]. The Argyris element also makes an appearance in exact sequences of finite elements, where differential complexes are used to explain the stability of many kinds of finite elements and derive new ones [AFW06a].

4.4 The Brezzi–Douglas–Marini element

4.4.1 Definition

The Brezzi–Douglas–Marini element [BDM85b] discretizes $H(\text{div})$. That is, it provides a vector field that may be assembled with continuous normal components so that global divergences are well-defined. The BDM space on a simplex in d dimensions ($d = 2, 3$) consists of vectors of length d whose components are polynomials of degree q for $q \geq 1$.

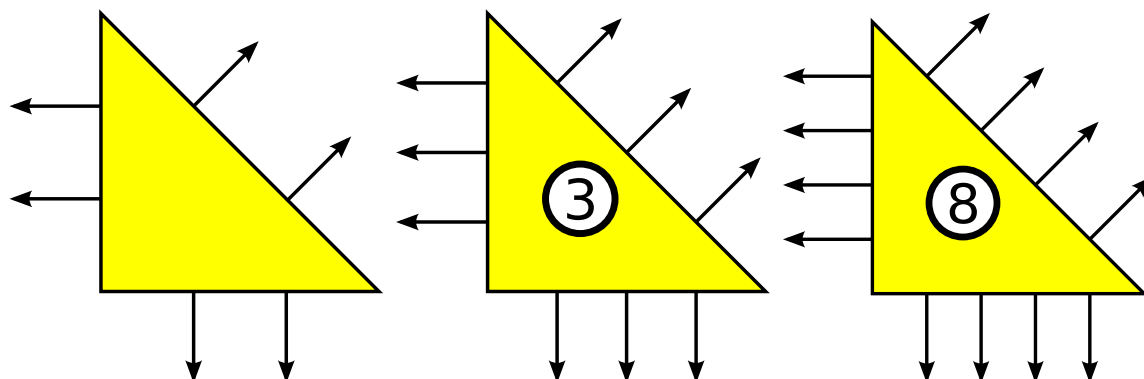


Figure 4.3: The linear, quadratic and cubic Brezzi–Douglas–Marini triangles.

The degrees of freedom for the BDM triangle include the normal component on each edge, specified either by integral moments against P_q or the value of the normal component at $q + 1$ points per edge. For $q > 1$, the degrees of freedom also include integration against gradients of $P_q(K)$ over K . For $q > 2$, the degrees of freedom also include integration against curls of $b_K P_{q-2}(K)$ over K , where b_K is the cubic bubble function associated with K .

► Author note: *What about tets? Will also make up for the empty space on the next page.*

The BDM element is also defined on rectangles and boxes, although it has quite a different flavor. Unusually for rectangular domains, it is not defined using tensor products of one-dimensional polynomials, but instead by supplementing polynomials of complete degree $[P_q(K)]^d$ with extra functions to make the divergence onto $P_q(K)$. The boundary degrees of freedom are similar to the simplicial case, but the internal degrees of freedom are integral moments against $[P_q(K)]^d$.

4.4.2 Historical notes

The BDM element was originally derived in two dimensions [BDM85b] as an alternative to the Raviart–Thomas element using a complete polynomial space.

Extensions to tetrahedra came via the “second-kind” elements of Nédélec [?] as well as in Brezzi and Fortin [BF91]. While Nédélec uses quite different internal degrees of freedom (integral moments against the Raviart–Thomas spaces), the degrees of freedom in Brezzi and Fortin are quite similar to [BDM85b].

A slight modification of the BDM element constrains the normal components on the boundary to be of degree $q - 1$ rather than q . This is called the Brezzi–Douglas–Fortin–Marini or BDFM element [BF91]. In similar spirit, elements with differing orders on the boundary suitable for varying the polynomial degree between triangles were derived in [BDM85a]. Besides mixed formulations of second-order scalar elliptic equations, the BDM element also appears in elasticity [AFW07], where it is seen that each row of the stress tensor may be approximated in a BDM space with the symmetry of the stress tensor imposed weakly. ► Author note: *Fill up the blank space here. Adding a discussion and possibly a figure for tets should help.*

4.5 The Crouzeix–Raviart element

4.5.1 Definition

The Crouzeix–Raviart element [CR73] most commonly refers to a linear non-conforming element. It uses piecewise linear polynomials, but unlike the Lagrange element, the degrees of freedom are located at edge midpoints rather than at vertices. This gives rise to a weaker form of continuity, but it is still a suitable C^0 -nonconforming element. The extension to tetrahedra in \mathbb{R}^3 replaces the degrees of freedom on edge midpoints by degrees of freedom on face midpoints.

► Author note: *What other element does it refer to? Sounds like there may be several, but I just know about this one.*

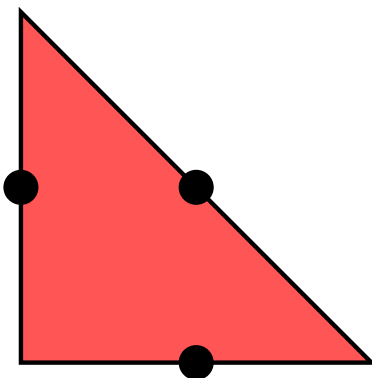


Figure 4.4: The linear Crouzeix–Raviart triangle.

4.5.2 Historical notes

Crouzeix and Raviart developed two simple Stokes elements, both using pointwise evaluation for degrees of freedom. The second element used extra bubble functions to enrich the typical Lagrange element, but the work of Crouzeix and Falk [CF89] later showed that the bubble functions were in fact not necessary for quadratic and higher orders.

► Author note: *The discussion in the previous paragraph should be expanded so it states more explicitly what this has to do with the CR element.*

The element is usually associated with solving the Stokes problem but has been used for linear elasticity [HL03] and Reissner–Mindlin plates [AF89] as a remedy for locking. There is an odd order extension of the element from Arnold and Falk.

► Author note: *Missing reference here to odd order extension.*

4.6 The Hermite Element

4.6.1 Definition

The Hermite element [Cia02] generalizes the classic cubic Hermite interpolating polynomials on the line segment. On the triangle, the space of cubic polynomials is ten-dimensional, and the ten degrees of freedom are point evaluation at the triangle vertices and barycenter, together with the components of the gradient evaluated at the vertices. The generalization to tetrahedra is analagous.

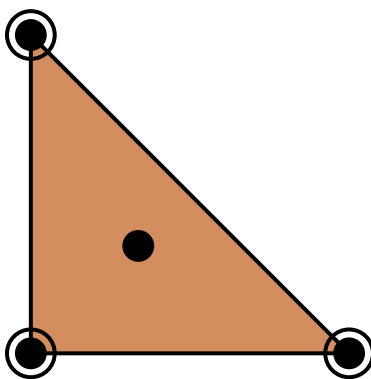


Figure 4.5: The cubic Hermite triangle.

Unlike the cubic Hermite functions on a line segment, the cubic Hermite triangle and tetrahedron cannot be patched together in a fully C^1 fashion.

4.6.2 Historical notes

Hermite-type elements appear in the finite element literature almost from the beginning, appearing at least as early as the classic paper of Ciarlet and Raviart [CR72]. They have long been known as useful C^1 -nonconforming elements [Bra07, Cia02]. Under affine mapping, the Hermite elements form *affine-interpolation equivalent* families. [BS08].

4.7 The Lagrange Element

4.7.1 Definition

The best-known and most widely used finite element is the Lagrange P_1 element. In general, the Lagrange element uses $\mathcal{P}_K = P_q(K)$, polynomials of degree q on K , and the degrees of freedom are simply pointwise evaluation at an array of points. While numerical conditioning and interpolation properties can be dramatically improved by choosing these points in a clever way [?], for the purposes of this chapter the points may be assumed to lie on an equispaced lattice.

► Author note: *Missing reference for statement about node placement.*

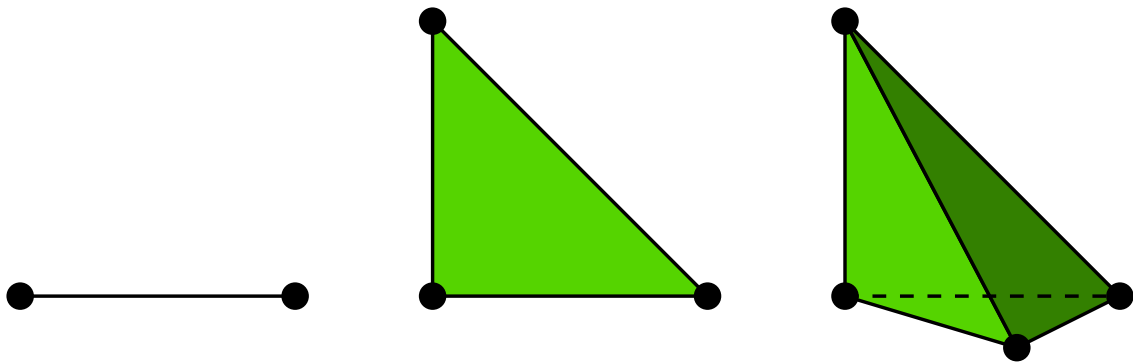


Figure 4.6: The linear Lagrange interval, triangle and tetrahedron.

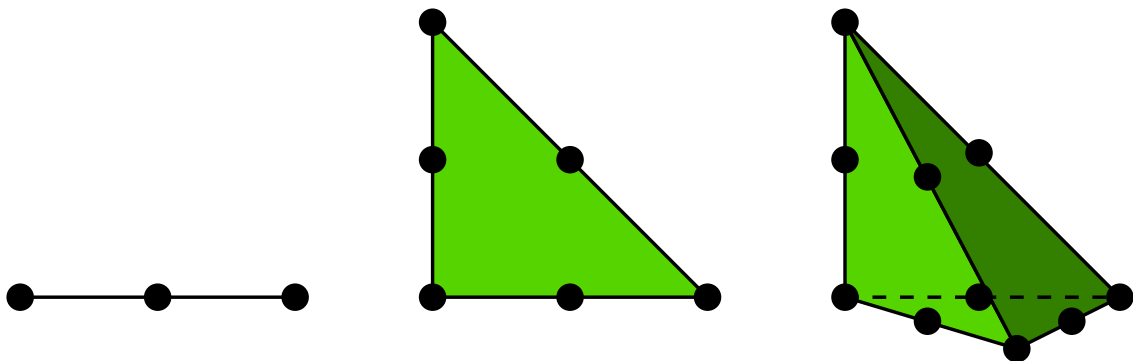


Figure 4.7: The quadratic Lagrange interval, triangle and tetrahedron.

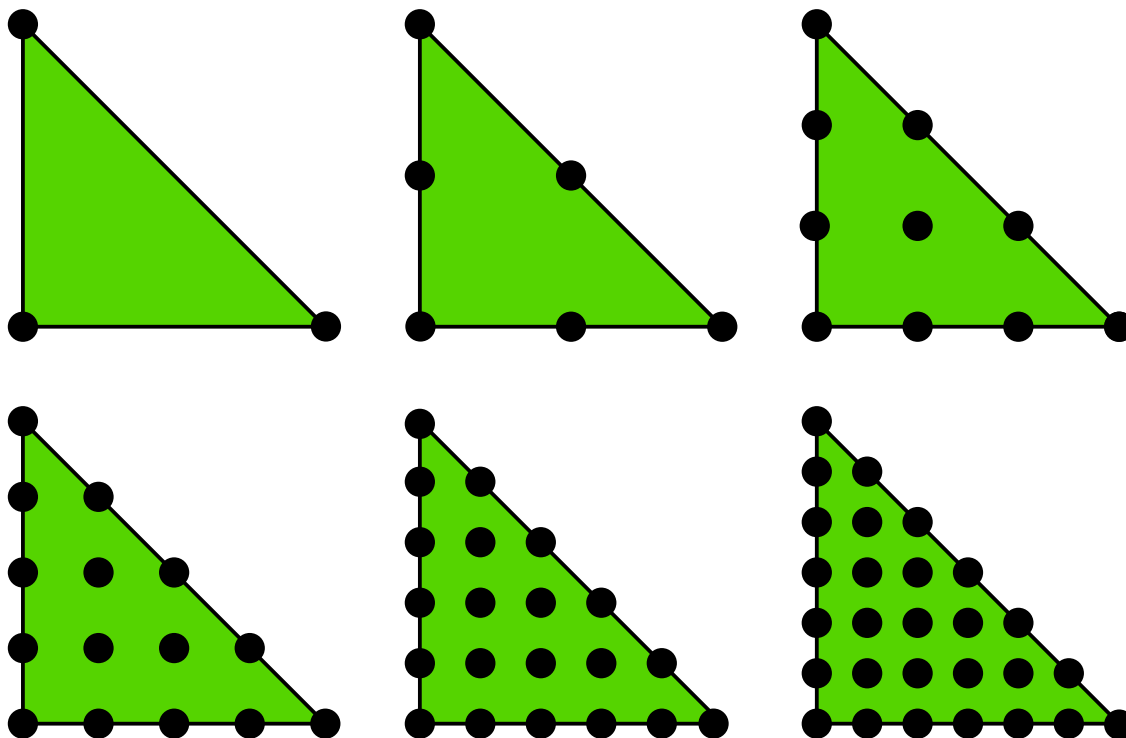


Figure 4.8: The Lagrange P_q triangle for $q = 1, 2, 3, 4, 5, 6$.

4.7.2 Historical notes

Reams could be filled with all the uses of the Lagrange elements. The Lagrange element predates the modern study of finite elements. The lowest-order triangle is sometimes called the *Courant* triangle, after the seminal paper [Cou43] in which variational techniques are considered and the P_1 triangle is used to derive a finite difference method. The rest is history.

► Author note: *Expand the historical notes for the Lagrange element. As far as I can see, Bramble and Zlamal don't seem to be aware of the higher order Lagrange elements (only the Courant triangle). Their paper from 1970 focuses only on Hermite interpolation.*

4.8 The Morley Element

4.8.1 Definition

The Morley triangle [Mor68] is a simple H^2 -nonconforming quadratic element that is used in fourth-order problems. The function space is simply $\mathcal{P}_K = P_2(K)$, the six-dimensional space of quadratics. The degrees of freedom consist of point-wise evaluation at each vertex and the normal derivative at each edge midpoint. It is interesting that the Morley triangle is neither C^1 nor even C^0 , yet it is suitable for fourth-order problems, and is the simplest known element for this purpose.

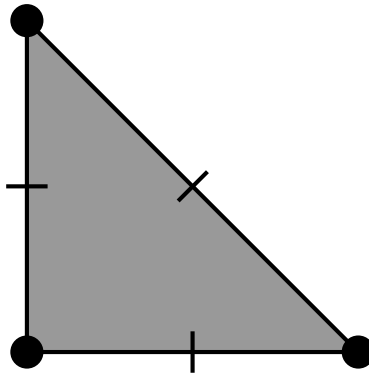


Figure 4.9: The quadratic Morley triangle.

4.8.2 Historical notes

The Morley element was first introduced to the engineering literature by Morley in 1968 [Mor68]. In the mathematical literature, Lascaux and Lesaint [LL75] considered it in the context of the patch test in a study of plate-bending elements.

► Author note: *Fill up page.*

4.9 The Nédélec Element

4.9.1 Definition

The widely celebrated $H(\text{curl})$ -conforming elements of Nédélec [?, ?] are much used in electromagnetic calculations and stand as a premier example of the power of “nonstandard” (meaning not lowest-order Lagrange) finite elements.

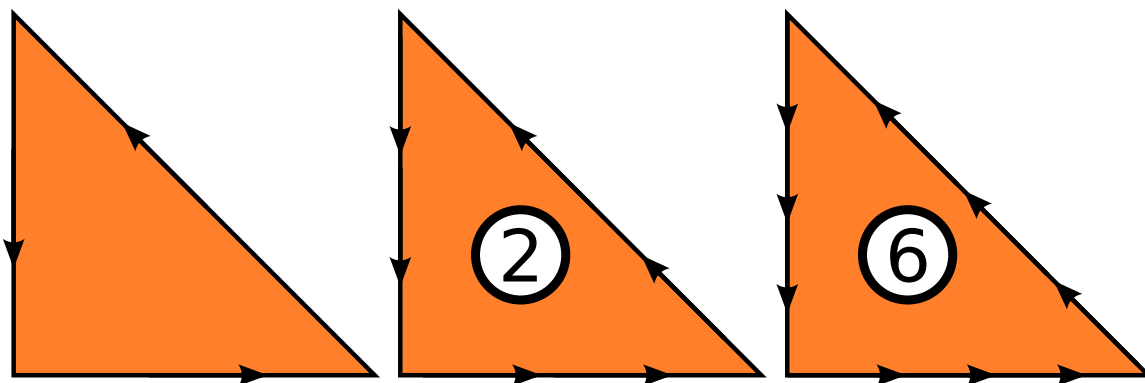


Figure 4.10: The linear, quadratic and cubic Nédélec triangles.

On triangles, the function space \mathcal{P}_K may be obtained by a simple rotation of the Raviart–Thomas basis functions, but the construction of the tetrahedral element is substantially different. In the lowest order case $q = 1$, the space \mathcal{P}_K may be written as functions of the form

$$v(x) = \alpha + \beta \times x,$$

where α and β are vectors in \mathbb{R}^3 . Hence, \mathcal{P}_K contains all vector-valued constant functions and some but not all linears. In the higher order case, the function space may be written as the direct sum

$$\mathcal{P}_K = [P_{q-1}(K)]^3 \oplus S_q,$$

where

$$S_q = \{v \in [\tilde{P}_q(K)]^3 : v \cdot x = 0\}.$$

Here, $\tilde{P}_q(K)$ is the space of homogeneous polynomials of degree q on K . An alternate characterization of \mathcal{P}_K is that it is the space of polynomials of degree $q + 1$ on which the q th power of the elastic stress tensor vanishes. The dimension of \mathcal{P}_q is exactly

$$n_K = \frac{q(q+2)(q+3)}{2}.$$

► Author note: *What is the q th power of the elastic stress tensor?*

Simplex	$H(\text{div})$		$H(\text{curl})$	
$K \subset \mathbb{R}^2$	RT_{q-1}	$\mathcal{P}_q^- \Lambda^1(K)$	$\text{NED}_{q-1}(\text{curl})$	—
	BDM_q	$\mathcal{P}_q \Lambda^1(K)$		
$K \subset \mathbb{R}^3$	$\text{RT}_{q-1} = \text{NED}_{q-1}^1(\text{div})$	$\mathcal{P}_q^- \Lambda^2(K)$	$\text{NED}_{q-1}^1(\text{curl})$	$\mathcal{P}_q^- \Lambda^1(K)$
	$\text{BDM}_q = \text{NED}_q^2(\text{div})$	$\mathcal{P}_q \Lambda^2(K)$	$\text{NED}_q^2(\text{curl})$	$\mathcal{P}_q \Lambda^1(K)$

Table 4.1: Nedelec elements of the first and second kind and their relation to the Raviart–Thomas and Brezzi–Douglas–Marini elements as well as to the notation of finite element exterior calculus.

► Author note: *What is the dimension on triangles?*

The degrees of freedom are chosen to ensure tangential continuity between elements and thus a well-defined global curl. In the lowest order case, the six degrees of freedom are the average value of the tangential component along each edge of the tetrahedron, hence the term “edge elements”. In the more general case, the degrees of freedom are the $q - 1$ tangential moments along each edge, moments of the tangential components against $(P_{q-2})^2$ on each face, and moments against $(P_{q-3})^3$ in the interior of the tetrahedron.

For tetrahedra, there also exists another family of elements known as Nedelec elements of the second kind, appearing in [?]. These have a simpler function space at the expense of more complicated degrees of freedom. The second kind space of order q is simply vectors of polynomials of degree q . The degrees of freedom are integral moments of degree q along each edge together with integral moments against lower-order first-kind bases on the faces and interior.

► Author note: *Note different numbering compared to RT, starting at 1, not zero.*

4.9.2 Historical notes

Nédélec’s original paper [?] provided rectangular and simplicial elements for $H(\text{div})$ and $H(\text{curl})$ based on incomplete function spaces. This built on earlier two-dimensional work for Maxwell’s equations [AGSNR80] and extended the work of Raviart and Thomas for $H(\text{div})$ to three dimensions. The second kind elements, appearing in [?], extend the Brezzi–Douglas–Marini triangle [BDM85b] to three dimensions and curl-conforming spaces. We summarize the relation between the Nedelec elements of first and second kind with the Raviart–Thomas and Brezzi–Douglas–Marini elements in Table 4.1.

In many ways, Nédélec’s work anticipates the recently introduced finite element exterior calculus [AFW06a], where the first kind spaces appear as $\mathcal{P}_q^- \Lambda^k$ spaces and the second kind as $\mathcal{P}_q \Lambda^k$. Moreover, the use of a differential operator

(the elastic strain) in [?] to characterize the function space foreshadows the use of differential complexes [AFW06b].

► Author note: *Should we change the numbering of the Nedelec elements and Raviart–Thomas elements to start at $q = 1$?*

4.10 The PEERS Element

4.10.1 Definition

The PEERS element [ABD84] provides a stable tensor space for discretizing stress in two-dimensional mixed elasticity problems. The stress tensor σ is represented as a 2×2 matrix, each row of which is discretized with a vector-valued finite element. Normally, one expects the stress tensor to be symmetric, although the PEERS element works with a variational formulation that enforces this condition weakly.

The PEERS element is based on the Raviart–Thomas element described in Section 4.11. If $\text{RT}_0(K)$ is the lowest-order Raviart–Thomas function space on a triangle K and b_K is the cubic bubble function that vanishes on ∂K , then the function space for the PEERS element is given by

$$\mathcal{P}_K = [\text{RT}_0(K) \oplus \text{span}\{\text{curl}(b_K)\}]^2.$$

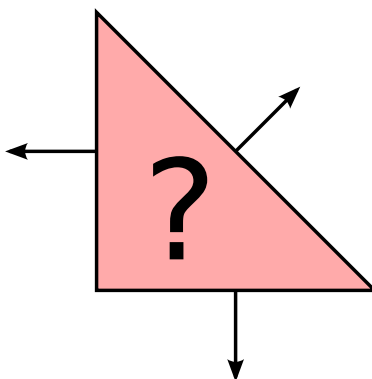


Figure 4.11: The PEERS triangle. One vector-valued component is shown.

- Author note: Which degrees of freedom in the interior? The curl?
- Author note: Is this really an element? We could also introduce other mixed elements like Taylor–Hood. But perhaps it’s suitable to include it since it is not a trivial combination of existing elements (the extra curl part).

4.10.2 Historical notes

Discretizing the mixed form of planar elasticity is quite a difficult task. Polynomial spaces of symmetric tensors providing inf-sup stability are quite rare, only appearing in the last decade [AW02]. A common technique is to relax the symmetry requirement of the tensor, imposing it weakly in a variational formulation.

This extended variational form requires the introduction of a new field discretizing the asymmetric portion of the stress tensor. When the PEERS element is used for the stress, the displacement is discretized in the space of piecewise constants, and the asymmetric part is discretized in the standard space of continuous piecewise linear elements.

The PEERS element was introduced in [ABD84], and some practical details, including postprocessing and hybridization strategies, are discussed in [AB85].

4.11 The Raviart–Thomas Element

4.11.1 Definition

The Raviart–Thomas element, like the Brezzi–Douglas–Marini and Brezzi–Douglas–Fortin–Marini elements, is an $H(\text{div})$ -conforming element. The space of order q is constructed to be the smallest polynomial space such that the divergence maps $\text{RT}_q(K)$ onto $P_q(K)$. The function space \mathcal{P}_K is given by

$$\mathcal{P}_K = P_{q-1}(K) + xP_{q-1}(K).$$

The lowest order Raviart–Thomas space thus consists of vector-valued functions of the form

$$v(x) = \alpha + \beta x,$$

where α is a vector-valued constant and β is a scalar constant.

On triangles, the degrees of freedom are the moments of the normal component up to degree q , or, alternatively, the normal component at $q + 1$ points per edge. For higher order spaces, these degrees of freedom are supplemented with integrals against a basis for $[P_{q-1}(K)]^2$.

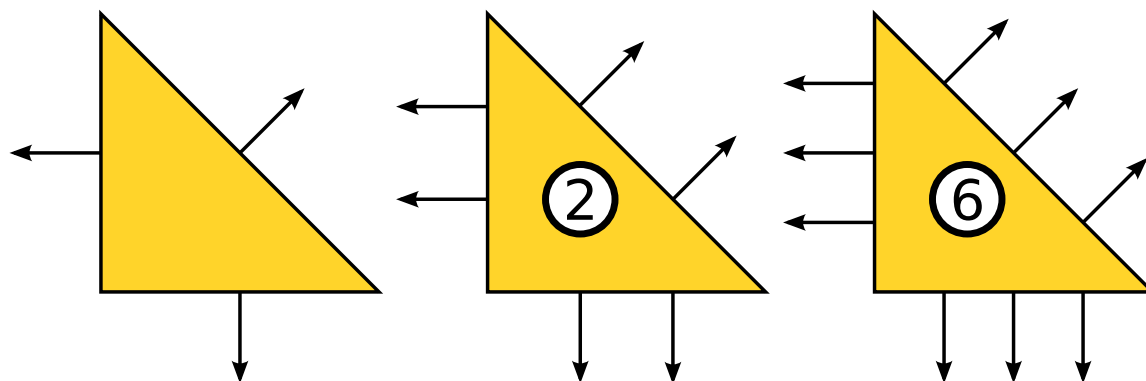


Figure 4.12: The zeroth order, linear and quadratic Raviart–Thomas triangles.

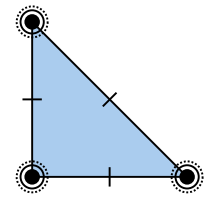
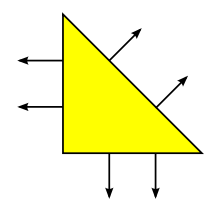
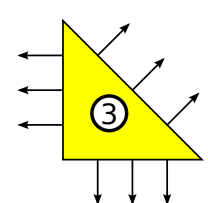
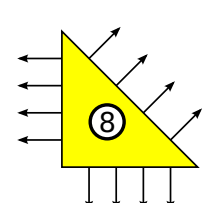
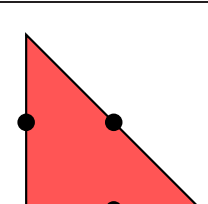
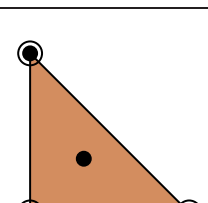
4.11.2 Historical notes

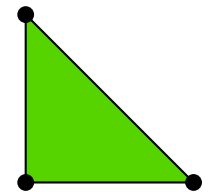
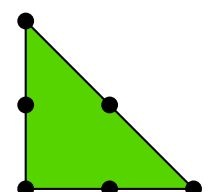
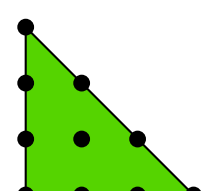
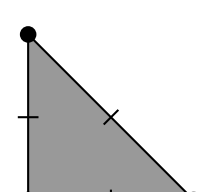
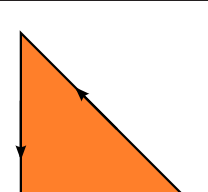
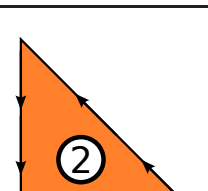
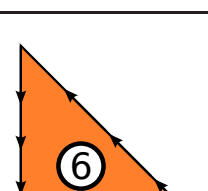
The Raviart–Thomas element was introduced in [RT77] in the late 1970’s, the first element to discretize the mixed form of second order elliptic equations. Shortly thereafter, it was extended to tetrahedra and boxes by Nédélec [?] and so is sometimes referred to as the Raviart–Thomas–Nédélec element or a first kind $H(\text{div})$ element.

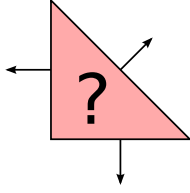
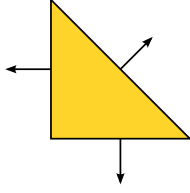
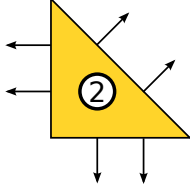
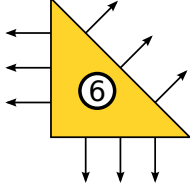
On rectangles and boxes, there is a natural relation between the lowest order Raviart–Thomas element and cell-centered finite differences. This was explored

in [RW83], where a special quadrature rule was used to diagonalize the mass matrix and eliminate the flux unknowns. Similar techniques are known for triangles [ADK⁺98], although the stencils are more complicated.

4.12 Summary

Notation	Element family	\mathcal{L}_K	$\dim \mathcal{P}_K$	References
ARG_5	Quintic Argyris		21	
BDM_1	Brezzi–Douglas–Marini		6	
BDM_2	Brezzi–Douglas–Marini		12	
BDM_3	Brezzi–Douglas–Marini		20	
CR_1	Crouzeix–Raviart		3	
HERM_q	Hermite		10	

P_1	Lagrange		3	
P_2	Lagrange		6	
P_3	Lagrange		10	
MOR_1	Morley		6	
NED_1	Nédélec		3	
NED_2	Nédélec		8	
NED_3	Nédélec		15	

PEERS	PEERS		?	
RT_0	Raviart–Thomas		3	
RT_0	Raviart–Thomas		8	
RT_0	Raviart–Thomas		15	

- ▶ Author note: *Add references to table.*
- ▶ Author note: *Indicate which elements are supported by FIAT and SyFi.*
- ▶ Author note: *Include formula for space dimension as function of q for all elements.*

Constructing General Reference Finite Elements

By Robert C. Kirby and Kent-Andre Mardal

Chapter ref: **[kirby-1]**

► Editor note: *Reuse color figures from chapter [kirby-7] for RT elements etc.*

5.1 Introduction

The finite element literature contains a huge collection of approximating spaces and degrees of freedom, many of which are surveyed in Chapter ??, Some applications, such as Cahn-Hilliard and other fourth-order problems, can benefit from very smooth finite element bases. While porous media flow requires vector fields discretized by piecewise polynomial functions with normal components continuous across cell boundaries. Many problems in electromagnetism call for the tangentially continuous vector fields obtained by using Nedelec elements [?, ?]. Many elements are carefully designed to satisfy the *inf-sup* condition [?, ?], originally developed to explain stability of discretizations of incompressible flow problems. Additionally, some problems call for low-order discretizations, while others are effectively solved with high-order polynomials.

While the automatic generation of computer programs for finite element methods requires one to confront the panoply of finite element families found in the literature, it also provides a pathway for wider employment of Raviart-Thomas, Nedelec, and other difficult-to-program elements. Ideally, one would like to describe the diverse finite element spaces at an abstract level, whence a computer code discerns how to evaluate and differentiate their basis functions. Such goals are in large part accomplished by the FIAT and SyFi projects, whose implemen-

tations are described in later chapters.

Projects like FIAT and SyFi may ultimately remain mysterious to the end user of a finite element system, as interactions with finite element bases are typically mediated through tools that construct the global finite element operators. The end user will typically be satisfied if two conditions are met. First, a finite element system should support the common elements used in the application area of interest. Second, it should provide flexibility with respect to order of approximation.

It is entirely possible to satisfy many users by *a priori* enumerating a list of finite elements and implement only those. At certain times, this would even seem ideal. For example, after the rash of research that led to elements such as the Raviart-Thomas-Nedelec and Brezzi-Douglas-Marini families, development of new families slowed considerably. Then, more recent work of lead forth by Arnold, Falk, and Winther in the context of exterior calculus has not only led to improved understanding of existing element families, but has also brought a new wave of elements with improved properties. A generative system for finite element bases can far more readily assimilate these and future developments. Automation also provides generality with respect to the order of approximation that standard libraries might not otherwise provide. Finally, the end-user might even easily define his own new element and test its numerical properties before analyzing it mathematically.

In the present chapter, we describe the mathematical formulation underlying such projects as FIAT [?, ?], SyFi [?, AM09] and Exterior [?]. This formulation starts from definitions of finite elements as given classically by Ciarlet [?]. It then uses basic linear algebra to construct the appropriate nodal basis for an abstract finite element in terms of polynomials that are easy to implement and well-behaved in floating point arithmetic. We focus on constructing nodal bases for a single, fixed reference element. As we will see in Chapter ??, form compilers such as `ffc` [Log07] and `sfc` [?] will work in terms of this single reference element.

Other approaches exist in the literature, such as the hierarchical bases studied by Szabo and Babuska [?] and extended to $H(\text{curl})$ and $H(\text{div})$ spaces in work such as [?]. These can provide greater flexibility for refining the mesh and polynomial degree in finite element methods, but they also require more care during assembly and are typically constructed on a case-by-case basis for each element family. When they are available, they may be cheaper to construct than using the technique studied here, but this present technique is easier to apply to an “arbitrary” finite element and so is considered in the context of automatic software.

5.2 Preliminaries

Both FIAT and SyFi work a slightly modified version of the abstract definition of a finite element introduced by Ciarlet [?].

Definition 5.1 (A Finite Element). *A finite element is a triple (K, P, N) with*

1. $K \subset \mathbb{R}^d$ a bounded domain with piecewise smooth boundary.
2. A finite-dimensional space P of $BC^m(K, V)$, where V is some normed vector space and BC^m is the space of m -times bounded and continuously differentiable functions from K into V .
3. A dual basis for P , written $N = \{L_i\}_{i=1}^{\dim P}$. These are bounded linear functionals in $(BC^m(K, V))'$ and frequently called the nodes or degrees of freedom.

In this definition, the term “finite element” refers not only to a particular cell in a mesh, but also to the associated function space and degrees of freedom. Typically, the domain K is some simple polygonal or polyhedral shape and the function space P consists of polynomials.

Given a finite element, a concrete basis, often called the nodal basis, for this element can be computed by using the following definition.

Definition 5.2. *The nodal basis for a finite element (K, P, N) be a finite element is the set of functions $\{\psi_i\}_{i=1}^{\dim P}$ such that for all $1 \leq i, j \leq \dim P$,*

$$L_i(\psi_j) = \delta_{i,j}. \quad (5.1)$$

The main issue at hand in this chapter is the *construction* of this nodal basis. For any given finite element, one may construct the nodal basis explicitly with elementary algebra. However, this becomes tedious as we consider many different families of elements and want arbitrary order instances of each family. Hence, we present a linear algebraic paradigm here that undergirds computer programs for automating the construction of nodal bases.

In addition to the construction of the nodal base we need to keep in mind that finite elements are patched together to form a piecewise polynomial field over a mesh. The fitness (or stability) of a particular finite element method for a particular problem relies on the continuity requirements of the problem. The degrees of freedom of a particular element are often chosen such that these continuity requirements are fulfilled.

Hence, in addition to computing the nodal basis, the mathematical structure developed here simplifies software for the following tasks:

1. Evaluate the basis function and their derivatives at points.
2. Associate the basis function (or degree of freedom) with topological facets of K such as its vertices, edges and its placement on the edges.

3. Associate the basis function with additional metadata such as a sign or a mapping that should be used together with the evaluation of the basis functions or its derivatives.
4. Provide rules for the degrees of freedom applied to arbitrary input functions determined at run-time.

The first of these is relatively simple in the framework of symbolic computation (SyFi), but they require more care if an implementation uses numerical arithmetic (FIAT). The middle two encode the necessary information for a client code to transform the reference element and assemble global degrees of freedom when the finite element is either less or more than C^0 continuous. The final task may take the form of a point at which data is evaluated or differentiated or more generally as the form of a sum over points and weights, much like a quadrature rule.

A common practice, employed throughout the FEniCS software and in many other finite element codes, is to map the nodal basis functions from this reference element to each cell in a mesh. Sometimes, this is as simple as an affine change of coordinates; in other cases it is more complicated. For completeness, we briefly describe the basics of creating the global finite elements in terms of a mapped reference element. Let therefore T be a polygon and \hat{T} the corresponding reference polygon. Between the coordinates $\mathbf{x} \in T$ and $\mathbf{x}_i \in \hat{T}$ we use the mapping

$$\mathbf{x} = \mathbf{G}(\mathbf{x}_i) + \mathbf{x}_0, \quad (5.2)$$

and define the Jacobian determinant of this mapping as

$$J(\mathbf{x}) = \left| \frac{\partial \mathbf{G}(\mathbf{x}_i)}{\partial \mathbf{x}_i} \right|. \quad (5.3)$$

The basis functions are defined in terms of the basis function on the reference element as

$$N_j(\mathbf{x}) = \hat{N}_j(\mathbf{x}_i), \quad (5.4)$$

where \hat{N}_j is basis function number j on the reference element. The integral can then be performed on the reference polygon,

$$\int_T f(\mathbf{x}) d\mathbf{x} = \int_{\hat{T}} f(\mathbf{x}_i) J d\mathbf{x}_i, \quad (5.5)$$

and the spatial derivatives are defined by the derivatives on the reference element and the geometry mapping simply by using the chain rule,

$$\frac{\partial N}{\partial x_i} = \frac{\partial N}{\partial \xi_j} \frac{\partial \xi_j}{\partial x_i}. \quad (5.6)$$

The above definition of a global element is commonly called *isoparametric* and is common for approximations in H^1 . For approximations in $H(\text{div})$ or $H(\text{curl})$ it is necessary to modify (5.4) with the Piola mapping. Furthermore, some elements like the Rannacher-Turek element [?, ?] has far better properties when defined globally compared its analogous definition in terms of a reference element.

5.3 Mathematical Framework

5.3.1 Change of basis

► **Editor note:** *Coordinate with notation in Chapter 3 where B is used for the Vandermonde matrix. Perhaps we could use \mathcal{V} ? Also use ℓ for the functionals and α for the expansion coefficients.*

The fundamental idea in constructing nodal basis is from elementary linear algebra: one constructs the desired (nodal) basis as a linear combination of a basis one has “in hand”. We will start with some basis $\{\phi\}_{i=1}^{\dim P} \subset P$. From this, we construct each nodal basis function

$$\psi_j = A_{jk}\phi_k, \quad (5.7)$$

where summation is implied over the repeated index k . The task is to compute the matrix A . Each fixed ψ_j must satisfy

$$L_i(\psi_j) = \delta_{i,j}, \quad (5.8)$$

and using the above expansion for ψ_j , we obtain

$$\delta_{i,j} = L_i(A_{jk}\phi_k) = A_{jk}L_i(\phi_k). \quad (5.9)$$

So, for a fixed j , we have a system of $\dim P$ equations

$$V_{ik}A_{jk} = e^j, \quad (5.10)$$

where

$$V_{ik} = L_i(\phi_k) \quad (5.11)$$

is a kind of generalized Vandermonde matrix and e^j is the canonical basis vector. Of course, (5.10) can be used to write a matrix equation for A as a linear system with $\dim P$ right hand sides and obtain

$$VA^t = I. \quad (5.12)$$

In practice, this, supposes that one has an implementation of the original basis for which the actions of the nodes (evaluation, differentiation, and integration) may be readily computed.

This discussion may be summarized as a proposition.

Proposition 5.3.1. *Define V and A as above. Then*

$$V = A^{-t}. \quad (5.13)$$

5.3.2 Polynomial spaces

In Definition 5.1 we defined the finite element in terms of a finite dimensional function space P . Although it is not strictly necessary, the functions used in finite elements are typically polynomials. While our general strategy will in principle accomodate non-polynomial bases, we only deal with polynomials in this chapter. A common space is \mathbb{P}_n^d , the space of polynomials of degree n in \mathbb{R}^d . There are many different ways to represent \mathbb{P}_n^d . We will discuss the power basis, orthogonal bases, and the Bernstein basis. Each of these bases has explicit representations and well-known evaluation algorithms. In addition to \mathbb{P}_n^d we will also for some elements need \mathbb{H}_n^d , the space of homogenous polynomials of degree n in d variables.

Typically, the developed techniques here are used on simplices, where polynomials do not have a nice tensor-product structure. Some rectangular elements like the Brezzi-Douglas-Marini family [], however, are not based on tensor-product polynomial spaces, and the techniques described in this chapter apply in that case as well. SyFi has explicit support for rectangular domains, but FIAT does not.

Power basis

On a line segment, $\mathbb{P}_n^1 = \mathbb{P}_n$ the monomial, or power basis is $\{x^i\}_{i=0}^n$, so that any $v \in P_n$ is written as

$$v = a_0 + a_1x + \dots a_nx^n = \sum_{i=0}^n a_i x^i. \quad (5.14)$$

In 2D on triangles, \mathbb{P}_n is spanned by functions on the form $\{x^i y^j\}_{i,j=0}^{i+j \leq n}$, with a similar definition in three dimensions.

This basis is quite easy to evaluate, differentiate, and integrate but is very ill-conditioned in numerical calculations.

Legendre basis

A popular polynomial basis for polygons that are either intervals, rectangles or boxes are the Legendre polynomials. This polynomial basis is also usable to represent polynomials of high degree. The basis is defined on the interval $[-1, 1]$, as

$$L_k(x) = \frac{1}{2^k k!} \frac{d^k}{dx^k} (x^2 - 1), \quad k = 0, 1, \dots,$$

A nice feature with these polynomials is that they are orthogonal with respect to the L_2 inner product, i.e.,

$$\int_{-1}^1 L_k(x) L_l(x) dx = \begin{cases} \frac{2}{2k+1}, & k = l, \\ 0, & k \neq l, \end{cases}$$

The Legendre polynomials are extended to rectangular domains in 2D and 3D simply by taking the tensor product,

$$L_{k,l,m}(x, y, z) = L_k(x)L_l(y)L_m(z).$$

and \mathbb{P}^n is defined by functions on the form (in 3D),

$$v = \sum_{\substack{k,l,m \leq n \\ k,l,m=0}} a_{k,l,m} L_{k,l,m}.$$

Dubiner basis

Orthogonal polynomials in simplicial domains are also known, although they lack some of the rotational symmetry of the Legendre polynomials. The Dubiner basis, frequently used in simplicial spectral elements [], is known under many names in the literature. It is an L^2 -orthogonal basis that can be constructed by mapping particular tensor products of Jacobi polynomials on a square by a singular coordinate change to a fixed triangle. Let $P_n^{\alpha,\beta}$ denote the n^{th} Jacobi polynomial with weights α, β . Then, define the new coordinates

$$\begin{aligned} \eta_1 &= 2 \left(\frac{1+x}{1-y} \right) - 1 \\ \eta_2 &= y, \end{aligned} \tag{5.15}$$

which map the square $[-1, 1]^2$ to the triangle with vertices $(-1, -1)$, $(-1, 1)$, $(1, -1)$ as shown in Figure ???. This is the natural domain for defining the Dubiner polynomials, but they may easily be mapped to other domains like the triangle with vertices $(0, 0)$, $(0, 1)$, $(1, 0)$ by an affine mapping. Then, one defines

$$\phi^{p,q}(x, y) = P_p^{0,0}(\eta_1) \left(\frac{1-\eta_2}{2} \right)^p P_q^{2p+1,0}(\eta_2). \tag{5.16}$$

Though it is not obvious from the definition, $\phi^{p,q}(x, y)$ is a polynomial in x and y of degree $p+q$. Moreover, for $(p, q) \neq (i, j)$, $\phi^{p,q}$ is L^2 -orthogonal to $\phi^{i,j}$.

While this basis is more complicated than the power basis, it is very well-conditioned for numerical calculations even with high degree polynomials. The polynomials can also be ordered hierarchically so that $\{\phi_i\}_{i=1}^{\dim P_k}$ forms a basis for polynomials of degree k for each $k > 0$. As a possible disadvantage, the basis lacks rotational symmetry that can be found in other bases.

Bernstein basis

The Bernstein basis is another well-conditioned basis that can be used in generating finite element bases. In 1D, the basis functions take the form,

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

and then P_n is spanned by $\{B_{i,n}\}_{i=0}^n$. And with this basis, \mathbb{P}_n can be spanned by functions on the form,

The terms x and $1 - x$ appearing in $B_{i,n}$ are the barycentric coordinates for $[0, 1]$, an observation that makes it easy to extend the basis to simplices in higher dimensions.

Let $b_1, b_2,$ and b_3 be the barycentric coordinates for the triangle shown in Figure ??, i.e., $b_1 = x, b_2 = y,$ and $b_3 = 1 - x - y$. Then the basis is of the form,

$$B_{i,j,k,n} = \frac{n!}{i!j!k!} b_1^i b_2^j b_3^k, \quad \text{for } i + j + k = n.$$

and a basis for \mathbb{P}_n is simply.

$$\{B_{i,j,k,n}\}_{i,j,k \geq 0}^{i+j+k=n}.$$

The Bernstein polynomials on the tetrahedron are completely analogous [?].

These polynomials, though less common in the finite element community, are well-known in graphics and splines. They have a great deal of rotational symmetry and are totally nonnegative and so give positive mass matrices, though they are not hierarchical.

Homogeneous Polynomials

Another set of polynomials which sometimes are useful are the set of homogeneous polynomials \mathbb{H}^n . These are polynomials where all terms have the same degree. \mathbb{H}^n is in 2D spanned by polynomials on the form:

$$v = \sum_{i,j, i+j=n} a_{i,j,k} x^i y^j$$

with a similar definition in n D.

Vector or Tensor valued Polynomials

It is straightforward to generalize the scalar valued polynomials discussed earlier to vector or tensor valued polynomials. Let $\{e_i\}$ be canonical basis in \mathbb{R}^d . Then a basis for the vector valued polynomials is

$$P_{ij} = P_j e_i,$$

with a similar definition extending the basis to tensors.

5.4 Examples of Elements

We include some standard finite elements to illustrate the concepts and motivate the need for different finite elements. Notice that the different continuity of the elements result in different approximation properties. We refer the reader to Chapter [] for a more thorough review of elements.

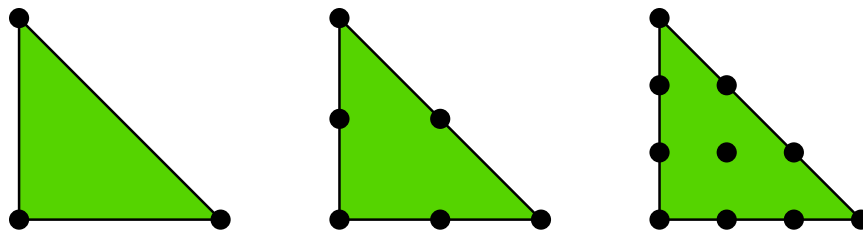


Figure 5.1: Lagrangian elements of order 1, 2, and 3.

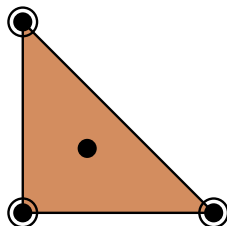


Figure 5.2: Hermite elements of order 3.

Example 5.1. The Lagrange Element

The Lagrangian element shown in Figure 5.1 is the most common element, where the black dots mean point evaluation. The first order element is shown in the leftmost triangle, it consists of three degrees of freedom in each of the vertices. The corresponding basis functions are x , y , and $1-x-y$. The second order element is shown in middle triangle, it has six degrees of freedom, three at the vertices and three at the edges. Finally, we have the third order Lagrangian element in the rightmost triangle, with ten degrees of freedom.

The Lagrangian element produce piecewise continuous polynomials and they are therefore well suited for approximation in H^1 . In general the number of degrees of freedom \mathbb{P}_n in $2D$ is $(n+2)(n+1)/2$, which is the same as the number of degrees of freedom in the Lagrange element. In fact, on a simplex in any dimension d the degrees of freedom of the Lagrange element of order n is the same as \mathbb{P}_n^d .

Example 5.2. The Hermite Element

In Figure 5.2 we show the Hermite element. The black dots mean point evaluation, while the white circles mean evaluation of derivatives in both x and y direction. Hence, the degrees of freedom for this element is three point evaluations at the vertices + six derivatives in the vertices + one internal point evaluation, which in total is ten degrees of freedom, which is the same number of degrees of

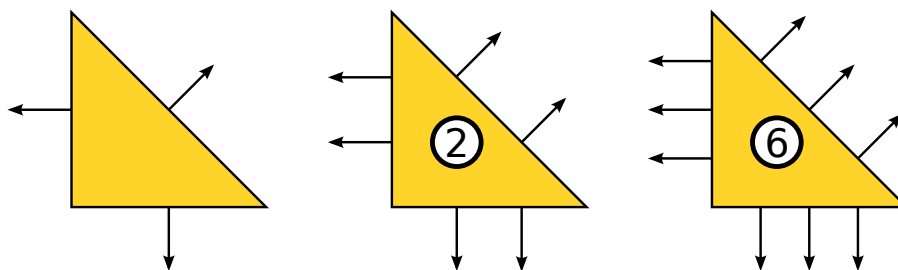


Figure 5.3: Triangular Raviart–Thomas elements of order 0, 1, and 2.

freedom as in \mathbb{P}_3^2 . The advantage of the Hermite element is that it has continuous derivatives at the vertices (it will however not necessarily result in a H^2 conforming approximation).

Example 5.3. The Raviart-Thomas Element

In Figure 5.3 we illustrate the Raviart-Thomas element. In contrast to the previous elements, this element has a vector-valued function space. The arrows represent normal vectors while the double dots indicate pointwise evaluation in both the x - and y -direction. Starting at $n = 0$, the dimension of RT_n is $(n + 1)(n + 3)$. The Raviart-Thomas element is typically used for approximations in $H(\text{div})$.

Remark 5.4.1. Earlier we saw common bases for \mathbb{P}_d^n , but elements like the Raviart-Thomas element described above use function spaces other than \mathbb{P}_d^n or vectors or tensors thereof. To fix this, we must either construct a basis for the appropriate polynomial space or work with a different element that includes full vectors of \mathbb{P}_d^n . In the case of $H(\text{div})$ elements, this corresponds to using the Brezzi-Douglas-Fortin-Marini elements.

5.4.1 Bases for other polynomial spaces

The basis presented above are suitable for constructing many finite elements, but as we have just seen, they do not work in all cases. The Raviart-Thomas function space,

$$(\mathbb{P}_n^2)^2 \oplus \begin{pmatrix} x \\ y \end{pmatrix} \mathbb{H}_n^2,$$

requires a basis for vectors of polynomials to be supplemented with an extra $\dim \mathbb{H}_n^2 = \dim \mathbb{P}_n^2 - \dim \mathbb{P}_{n-1}^2 = n$ functions. In fact, any n polynomials in $\mathbb{P}_n^2 \setminus \mathbb{P}_{n-1}^2$ will work, not just homogeneous polynomials, although the sum above will no longer be direct.

While the Raviart-Thomas element requires supplementing a standard basis with extra functions, the Brezzi-Douglas-Fortin-Marini triangle uses the function space

$$\{u \in (\mathbb{P}_n^2(K))^2 : u \cdot n \in \mathbb{P}_{n-1}^1(\gamma), \gamma \in \mathcal{E}(K)\}.$$

Obtaining a basis for this space is somewhat more subtle. FIAT and SyFi have developed different but mathematically equivalent solutions. Both rely on recognizing that the required function space sits inside $(\mathbb{P}_n^2)^2$ and can be characterized by certain functionals that vanish on this larger space.

Three such functionals describe the basis for the BDFM triangle. If μ_n^γ is the Legendre polynomial of order n on an edge γ , then the functional

$$\ell_\gamma(u) = \int_\gamma (u \cdot n) \mu_n^\gamma$$

acting on $(\mathbb{P}_n^2)^2$ must vanish on the BDFM space, for the n^{th} degree Legendre polynomial is orthogonal to all polynomials of lower degree.

Now, we define the matrix

$$V = \begin{pmatrix} V^1 \\ V^2 \end{pmatrix}. \quad (5.17)$$

Here, $V^1 \in \mathbb{R}^{2 \dim \mathbb{P}_n^2 - 3, 2 \dim \mathbb{P}_n^2}$ and $V^2 \in \mathbb{R}^{3, 2 \dim \mathbb{P}_n^2}$ are defined by

$$V_{ij}^1 = L_i(\phi_j),$$

$$V_{ij}^2 = \ell_i(\phi_j),$$

where $\{\phi_j\}_{j=1}^{2 \dim \mathbb{P}_n^2}$ is a basis for $(\mathbb{P}_n^2)^2$.

Consider now the matrix equation

$$VA^t = I_{2 \dim P_n, 2 \dim P_n - 3}, \quad (5.18)$$

where $I_{m,n}$ denotes the $m \times n$ identity matrix with $I_{i,j} = \delta_{i,j}$. As before, the columns of A still contain the expansion coefficients of the nodal basis functions ψ_i in terms of $\{\phi_j\}$. Moreover, $V_2 A = 0$, which implies that the nodal basis functions are in fact in the BDFM space.

More generally, we can think of our finite element space P being embedded in some larger space \bar{P} for which there is a readily usable basis. If we may characterize P by

$$P = \bigcap_{i=1}^{\dim \bar{P} - \dim P} \ell_i,$$

where $\ell_i : \bar{P} \rightarrow \mathbb{R}$ are linear functionals, then we may apply this technique. In this case, $V_1 \in \mathbb{R}^{\dim P, \dim \bar{P}}$ and $V_2 \in \mathbb{R}^{\dim \bar{P} - \dim P, \dim \bar{P}}$. This scenario, though somewhat abstract, is applicable not only to BDFM, but also to the Nédélec [?], Arnold-Winther [AW02], Mardal-Tai-Winther [?], Tai-Winther [?], and Bell [] element families.

Again, we summarize the discussion as a proposition.

Proposition 5.4.1. *Let (K, P, N) be a finite element with $P \subset \bar{P}$. Let $\{\phi_i\}_{i=1}^{\dim \bar{P}}$ be a basis for \bar{P} . Suppose that there exist functionals $\{\ell_i\}_{i=1}^{\dim \bar{P} - \dim P}$ on \bar{P} such that*

$$P = \bigcap_{i=1}^{\dim \bar{P} - \dim P} \text{null}(\ell_i).$$

Define the matrix A as in (5.18). Then, the nodal basis for P may be expressed as

$$\psi_i = A_{ij} \phi_j,$$

where $1 \leq i \leq \dim P$ and $1 \leq j \leq \dim \bar{P}$.

5.5 Operations on the Polynomial spaces

Here, we show various important operations may be cast in terms of linear algebra, supposing that they may be done on original basis $\{\phi_i\}_{i=1}^{\dim P}$.

5.5.1 Evaluation

In order to evaluate the nodal basis $\{\psi_i\}_{i=1}^{\dim P}$ at a given point $x \in K$, one simply computes the vector

$$\Phi_i = \phi_i(x)$$

followed by the product

$$\psi_i(x) \equiv \Psi_i = A_{ij} \Phi_j.$$

Generally, the nodal basis functions are required at an array of points $\{x_j\}_{j=1}^m \subset K$. For performance reasons, performing matrix-matrix products may be advantageous. So, define $\Phi_{ij} = \Phi_i(x_j)$ and $\Psi_{ij} = \Psi_i(x_j)$. Then all of the nodal basis functions may be evaluated by the product

$$\Psi_{ij} = A_{ik} \Phi_{kj}.$$

5.5.2 Differentiation

Differentiation is more complicated, and also presents more options. We want to handle the possibility of higher order derivatives. Let $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_d)$ be a multiindex so that

$$D^\alpha \equiv \frac{\partial^{|\alpha|}}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_d^{\alpha_d}},$$

where $|\alpha| = \sum_{i=1}^d \alpha_i$.

We want to compute the array

$$\Psi_i^\alpha = D^\alpha \psi_i(x)$$

for some $x \in K$.

One may differentiate the original basis functions $\{\phi_i\}$ to produce an array

$$\Phi_i^\alpha = D^\alpha \psi_i(x),$$

whence

$$\Psi_i^\alpha = A_{ij} \Phi_j^\alpha.$$

This presupposes that one may conveniently compute all derivatives of the $\{\phi_i\}$. This is typically true in symbolic computation or when using the power basis. Alternatively, automatic differentiation [] could be used in the power or Bernstein basis. The Dubiner basis, as typically formulated, contains a coordinate singularity that prevents automatic differentiation from working at the top vertex. Recent work [] has reformulated recurrence relations to allow for this possibility.

If one prefers (or is required by the particular starting basis), one may also compute matrices that encode first derivatives acting on the $\{\phi_i\}$ and construct higher derivatives than these. For each coordinate direction x_k , a matrix D^k is constructed so that

$$\frac{\partial \phi_i}{\partial x_k} = D_{ij}^k \phi_j.$$

How to do this depends on which bases are chosen. For particular details on the Dubiner basis, see []. Then, Ψ^α may be computed by

$$\Psi_i^\alpha = (D^\alpha A)_{ij} \Phi_j,$$

5.5.3 Integration

Integration of basis functions over K , including products of basis functions and/or their derivatives, may be performed numerically, symbolically, or exactly with some known formula. An important aspect of automation is that it allows general orders of approximation. While this is not difficult in symbolic calculations, a numerical implementation like FIAT must work harder. If Bernstein polynomials are used, we may use the formula

$$\int_K b_1^i b_2^j b_3^k dx = \frac{i!j!k!}{(i+j+k+2)!} \frac{|K|}{2}$$

on triangles and a similar formula for lines and tetrahedra to calculate integrals of things expressed in terms of these polynomials exactly. Alternately, if the Dubiner basis is used, orthogonality may be used. In either case, when derivatives occur, it may be as efficient to use numerical quadrature. On rectangular domains, tensor products of Gauss or Gauss-Lobatto quadrature can be used to give efficient quadrature families to any order accuracy. On the simplex, however, optimal quadrature is more difficult to find. Rules based on barycentric symmetry [] may be used up to a certain degree (which is frequently high enough in practice). If one needs to go beyond these rules, it is possible to use the mapping (5.15) to map tensor products of Gauss-Jacobi quadrature rules from the square to the triangle.

5.5.4 *Linear functionals*

Integration, pointwise evaluation and differentiation all are examples of linear functionals. In each case, the functionals are evaluated by their action on the $\{\phi_i\}$

CHAPTER 6

Finite Element Variational Forms

By Robert C. Kirby and Anders Logg

Chapter ref: **[kirby-5]**

Summarize notation for variational forms. Introduce global tensor A and element tensor A^K .

CHAPTER 7

Finite Element Assembly

By Anders Logg

Chapter ref: [**logg-3**]

Overview of general assembly algorithm.

CHAPTER 8

Quadrature Representation of Finite Element Variational Forms

By Kristian B. Ølgaard and Garth N. Wells

Chapter ref: **[oelgaard-2]**

Summarise work on optimizing quadrature representation using automated code generation and address automated selection of best representation.

CHAPTER 9

Tensor Representation of Finite Element Variational Forms

By Anders Logg and possibly others

Chapter ref: [**logg-4**]

Overview of tensor representation.

CHAPTER 10

Discrete Optimization of Finite Element Matrix Evaluation

By Robert C. Kirby, Matthew G. Knepley, Anders Logg, L. Ridgway Scott and Andy R.

Terrel

Chapter ref: [**kirby-4**]

The tensor contraction structure obtained in the representation results for variational forms enables not only the construction of a compiler for variational forms, but an *optimizing* compiler. For typical variational forms, the reference element tensor has significant structure that allows it to be contracted for an arbitrary element tensor in a reduced amount of arithmetic. Reducing the number of operations based on this structure leads naturally to several problems in discrete mathematics. This chapter introduces the idea of complexity-reducing relations and discusses compile-time graph and hypergraph optimization problems that form the core of the FErari project.

Parallel Adaptive Mesh Refinement

By Johan Hoffman, Johan Jansson and Niclas Jansson

Chapter ref: [**hoffman-4**]

For many interesting problems one is often interested in rather localized features of a solution, for example separation or transition to turbulence in flow problems. It is often the case that it is too expensive to uniformly refine a mesh to such an extent that these features develop. The solution is to only refine the mesh in the regions of most interest, or for example where the error is large.

This chapter is based on the work in [?]. First a brief summary of the theory behind mesh refinement is given, followed by a discussion of the issues with parallel refinement together with our proposed solution. The chapter ends with a presentation of our implementation of a fully distributed parallel adaptive mesh refinement framework on a Blue Gene/L.

11.1 A brief overview of parallel computing

In this chapter, parallel computing refers to distributed memory systems with message passing. It is assumed that the system is formed by linking compute nodes together with some interconnect, in order to form a virtual computer (or cluster) with a large amount of memory and processors.

It is also assumed that the data, the mesh in this case is distributed across the available processors. Shared entities are duplicated or “ghosted” on adjacent processors. In order to fully represent the original mesh from the smaller distributed sub meshes, it is essential that the information for the shared entities are the same for all processors.

11.2 Local mesh refinement

Local mesh refinement has been studied by several authors over the past years. The general idea is to split an element into a set of new ones, with the constraint that the produced mesh must be valid. Usually a mesh is valid if there are no “hanging nodes”, that is no node should be on another element’s facet. How to split the element in order to ensure this differs between different methods. One common theme is edge bisection.

A common edge bisection algorithm bisects all edges in the element, introducing a new vertex on each edge, and connecting them together to form the new elements (see for example [?]). Other methods bisect only one of the edges, which edge depends on the method. For example one could select the edge which where most recently refined, this method is often referred to as the newest vertex approach and where described in [?]. Another popular edge bisection method is the longest edge [?], where one always select the longest edge for refinement. In order to ensure that the mesh is free of “hanging nodes”, the algorithm recursively bisects elements until there are no “hanging nodes” left.

11.2.1 The challenge of parallel mesh refinement

Performing the refinement in parallel adds additional constraints on the refinement method. Not only should the method prevent “hanging nodes”, it must also be guaranteed to terminate in a finite number of steps.

In the parallel case, each processor owns a small part of the distributed mesh. So if a new vertex is introduced on the boundary between two processors, the algorithm must ensure that the information propagates to all neighboring processors.

For an algorithm that bisects all edges in an element, the problem reduces to a global decision problem, deciding which of the processors information that should be used on all the other processors. But for an algorithm that propagates the refinement like the longest edge, the problem becomes a synchronization problem i) to detect and handle refinement propagation between different processors and ii) to detect global termination.

The first problem could easily be solved by dividing the refinement into two different phases, one local refinement phase and one propagation phase. In the first phase elements on the processor are refined with an ordinary serial refinement method. This could create non conforming elements on the boundary between processors. These are fixed by propagating the refinement to the neighboring processor. This is done in the second propagation phase. But since the next local refinement phase could create an invalid mesh, one could get another propagation phase and the possibility of another and so forth. However, if the longest edge algorithm is used, termination is guaranteed [?]. But the problem is to detect when all these local meshes are conforming, and also when they are

conforming at the same time, that means one has to detect global termination, which is a rather difficult problem to solve efficiently, especially for massively parallel systems for which we are aiming.

There has been some other work related to parallel refinement with edge bisection. For example a parallel newest vertex algorithm has been done by Zhang [?]. Since the algorithm does not need to solve the termination detection problem, scaling is simply a question of interconnect latency. Another work is the parallel longest edge algorithm done by Castaños and Savage [?]. They solve the termination detection problem with Dijkstras general distributed termination algorithm, which simply detects termination by counting messages sent and received from some controlling processor. However, in both of these methods they only used a fairly small amount of processors, less then one hundred, so it is difficult to estimate how efficient and scalable these algorithms are. For more processors the effects of communication cost and concurrency in the communication patterns starts to be important, therefore we tried to design an algorithm that would scale well for thousands of processors.

11.2.2 A modified longest edge bisection algorithm

Instead of trying to solve the termination detection problem, one could try to modify the refinement algorithm in such a way that it would only require one synchronization step, thus less communication. With less communication overhead it should also scale better for a large number of processors.

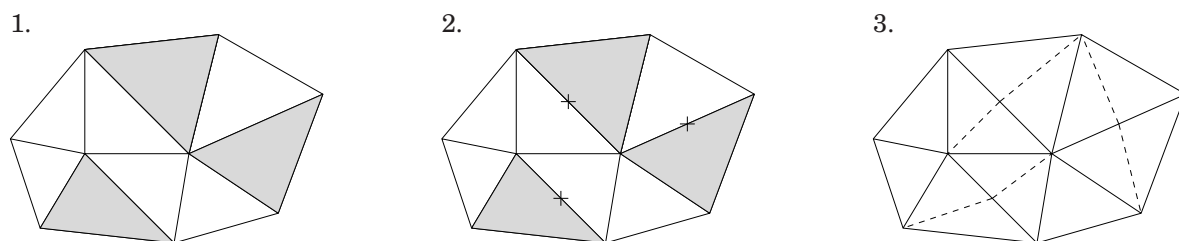


Figure 11.1: An example of the refinement algorithm used. First a set of elements are marked for refinement (1). The longest edges are found (2), and all elements connected to these edges are finally bisected, the dashed lines in (3).

When this work started, DOLFIN did not have a pure longest edge implementation. Instead of recursively fixing “hanging nodes” it bisected elements in pairs (or groups) (see figure 11.1). Since this algorithm always terminate the refinement by bisecting all elements connected to the refined edge, it is a perfect candidate for an efficient parallel algorithm. If the longest edge is shared by different processors, the algorithm must only propagate the refinement onto all

elements (processors) connected to that edge, but then no further propagation is possible (see figure 11.2).

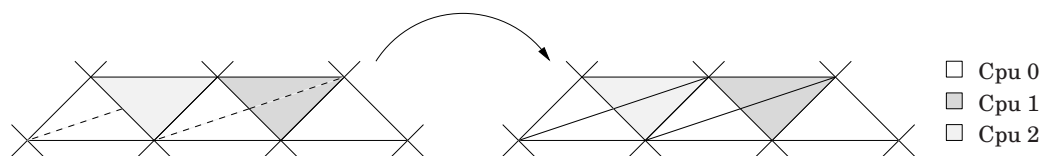


Figure 11.2: An example of the two simple cases of propagation. The dashed lines refers to how the processors wants to bisect the elements.

However, notifying an adjacent processor of propagation does not solve the problem entirely. As mentioned in section 11.1, all mesh entities shared by several processors must have the same information in order to correctly represent the distributed mesh. The refinement process must therefore guarantee that all newly created vertices are assigned the same unique information on all the neighboring processors. Another problematic case arises when processors refine the same edge and the propagation “collides” (see figure 11.2). In this case the propagation is done implicitly but the processors must decide which of the new information to use.

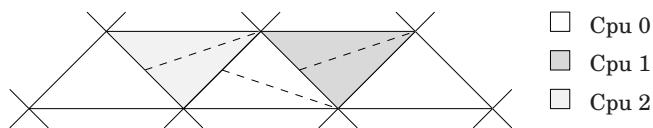


Figure 11.3: An example of the problematic case with multiple propagations. The dashed lines refers to how the processors wants to bisect the elements.

A more complicated case is when an element receives multiple propagation (possibly from different processors) on different edges (see figure 11.3). Since the modified longest edge algorithm only allows one edge to be bisected per element, one of the propagations must be selected and the other one rejected. This however adds a difficulty to the simple algorithm. First of all, how should the processors decide upon which edge to be refined? Clearly this could not be done arbitrarily since when a propagation is forbidden, all refinement done around that edge must be removed. Thus, in the worst case it could destroy the entire refinement.

To solve the edge selection problem perfectly one needs an algorithm with a global view of the mesh. In two dimensions with a triangular mesh, the problem could be solved rather simple since each propagation could only come from two different edges (one edge is always facing the interior). By exchanging the desired propagation edges processors could match theirs selected edges with the

propagated ones, in an attempt to minimize the number of forbidden propagations. However, in three dimensions the problem starts to be such complicated that multiple exchange steps are needed in order to solve the problem. Hence, it becomes too expensive to solve.

Instead we propose an algorithm which solves the problem using an edge voting scheme. Each processor refines the boundary elements, find their longest edge and cast a vote for it. These votes are then exchanged between processors, which add the received votes to its own set of votes. Now the refinement process restarts, but instead of using the longest edge criteria, edges are selected depending on the maximum numbers of votes. In the case of a tie, the edge is selected depending on a random number assigned to all votes.

Once a set of edges has been selected from the voting phase the actually propagation starts by exchanging these with the other processors. However, the voting could fail to “pair” refinements together. For example, an element could lie between two processors which otherwise does not share any common face. Each of these processors wants to propagate into the neighboring element but on different edges (see figure 11.4). Since the processors on the left and right side of the element do not receive the edge vote from each other, the exchange of votes will in this case not help with selecting an edge that would work for both processors.

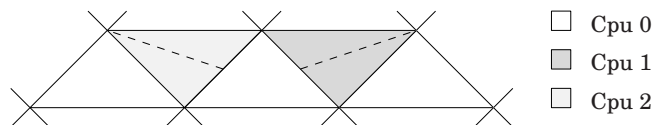


Figure 11.4: An example of the case when edge votes could be missed. The dashed lines refers to how the processors wants to bisect the elements.

To fix this, an additionally exchange step is needed and maybe another and so forth, rendering the perfect fix impossible. Instead, the propagation step ends by exchanging the refined edges which gave rise to a forbidden propagation. All processors could then remove all refinements that these edges introduced, and in the process, remove any hanging nodes on the boundary between processors.

11.3 The need of dynamic load balancing

For parallel refinement, there is an additional problem not present in the serial setting. As one refines the mesh, new vertices are added arbitrarily at any processor. Thus, rendering an initially good load balance useless. Therefore, in order to sustain a good parallel efficiency the mesh must be repartitioned and redistributed after each refinement, in other words dynamic load balancing is needed.

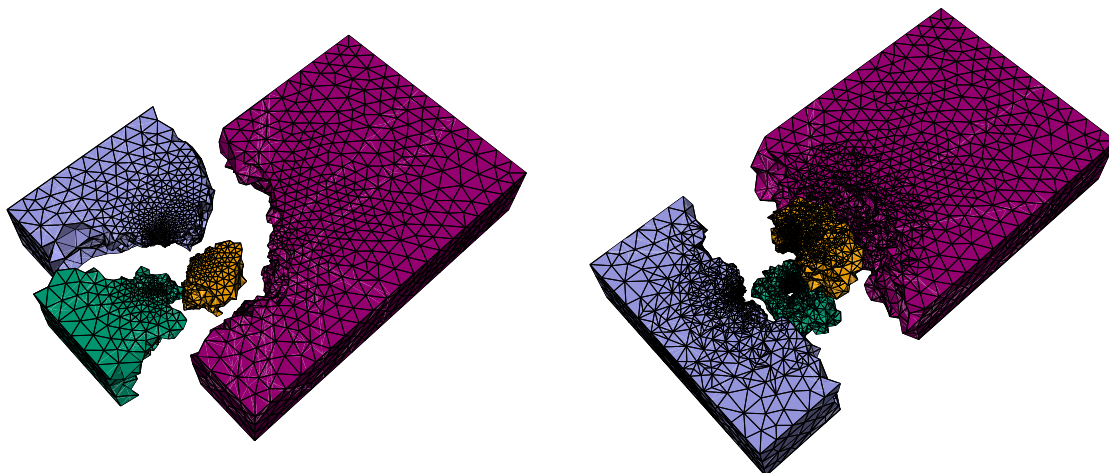


Figure 11.5: An example of load balancing where the locality of the data is considered. The mesh is refined three times around the cylinder, and for each step load balancing is performed, shading refers to processor id.

In the worst case, the load balancing routine must be invoked every time a mesh is adapted, it has to be rather efficient, and for our aim, scale well for a large number of processors. There are mainly two different load balancing methods used today, diffusive and remapping methods. Diffusive methods, like the physical meaning, by finding a diffusion solution a heavily loaded processor's vertices would move to another processor and in that way smear out the imbalance, described for example in [?, ?]. Remapping methods, relies on the partitioner's efficiency of producing good partitions from an already partitioned dataset. In order to avoid costly data movement, a remapping method tries to assign the new partitions to processors in an optimal way. For problems where the imbalance occurs rather localized, the remapping methods seems to perform better [?]. Hence, it maps perfectly to the localized imbalance from local mesh refinement.

In this work, we used the load balancing framework of PLUM [?] a remapping load balancer. The mesh is repartitioned according to an imbalance model. Repartitioning is done before refinement, this would in theory minimize data movement and speedup refinement, since a more balanced number of element would be bisected on each processor.

11.3.1 Workload modelling

The workload is modelled by a weighted dual graph of the mesh. Let $G = (V, E)$ be the dual graph of the mesh, q be one of the partitions and let w_i be the computational work (weights) assigned to each graph vertex. The processor workload

is then defined as

$$W(q) = \sum_{\forall w_i \in q} w_i \quad (11.1)$$

where communication costs are neglected. Let W_{avg} be the average workload and W_{max} be the maximum, then the graph is considered imbalanced if

$$W_{max}/W_{avg} > \kappa \quad (11.2)$$

where the threshold value κ is based on the problem or machine characteristics.

This model suits the modified longest edge algorithm (section 11.2.2) perfectly. Since the modifications reduces the algorithm to only have one propagation and/or synchronization step. The workload calculation becomes a local problem, thus it is rather cheap to compute. So if we let each element represent one unit of work, a mesh adapted by the modified algorithm would produce a dual graph with vertex weights equal to one or two. Each element is only bisected once, giving a computational weight of two elements for each element marked for refinement.

11.3.2 Remapping strategies

Remapping of the new partitions could be done in numerous ways, depending on what metric one tries to minimize. Usually one often talks about the two metrics TOTALV and MAXV. MAXV tries to minimize the redistribution time by lowering the flow of data, while TOTALV lower the redistribution time by trying to keep the largest amount of data local, for a more detailed description see [?]. We have chosen to focus on the TOTALV metric, foremost since it much cheaper to solve then MAXV, and it produces equally good (or even better) balanced partitions.

Independent of which metric one tries to solve. The result from the repartitioning is placed in a similarity matrix S , where each entry $S_{i,j}$ is the number of vertices on processor i which would be placed in the new partition j . In our case, we want to keep the largest amount of data local, hence to keep the maximum row entry in S local. This could be solved by transforming the matrix S into a bipartite graph where each edge $e_{i,j}$ is weighted with $S_{i,j}$, the problem then reduces to the maximally weighted bipartite graph problem [?].

11.4 The implementation on a massively parallel system

The adaptive refinement method described in this chapter was implemented using an early parallel version of DOLFIN, for a more detailed description see [?]. Parallelization was implemented for a message passing system, and all the algorithms were designed to scale well for a large number of processors.

The challenge of implementing a refinement algorithm on a massively parallel system is as always the communication latency. In order to avoid that the message passing dominates the runtime, it is important that the communication is kept at a minimum. Furthermore, in order to obtain a good parallel efficiency, communication patterns must be design in such way that they are highly concurrent, reducing processors idle time etc.

11.4.1 *The refinement method*

Since element bisection is a local problem, without any communication, the only part of the refinement algorithm that has to be well designed for a parallel system is the communication pattern used for refinement propagation.

For small scale parallelism, one could often afford to do the naive approach, loop over all processors and exchange messages without any problem. When the number of processors are increased, synchronization, concurrency and deadlock prevention starts to become important factors to considered when designing the communication pattern. A simple and reliable pattern is easily implemented as follows. If the processors send data to the left and receive data from the right in a circular pattern, all processors would always be busy sending and receiving data, thus no idle time.

Algorithm 1 Communication pattern

► Editor note: *Package `algorithmic.sty` conflicts with `algorithmicx.sty`. Sort out which packages to use for algorithms.*

The refinement algorithm outlined in 11.2.2 is easily implemented as a loop over all elements marked for refinement. For each element marked for refinement it finds the longest edge and bisect all elements connected to that edge. However, since an element is only allowed to be bisected once, the algorithm is only allowed to select the longest edge which is part of an unrefined element. In order to make this work in parallel, one could structure the algorithm in such a way that it first refines the shared elements, and propagate the refinement. After that it could refine all interior elements without any communication.

Algorithm 2 Refinement algorithm

► Editor note: *Package `algorithmic.sty` conflicts with `algorithmicx.sty`. Sort out which packages to use for algorithms.*

If we let \mathcal{B} be the set of elements on the boundary between processors, \mathcal{R} the set of elements marked for refinement. Then by using algorithm 1 we could efficiently express the refinement of shared entities in algorithm 2 with algorithm 3.

Algorithm 3 Refinement of shared entities

► Editor note: *Package `algorithmic.sty` conflicts with `algorithmicx.sty`. Sort out which packages to use for algorithms.*

11.4.2 The remapping scheme

The maximally weighted bipartite graph problem for the TOTALV metric could be solved in an optimal way in $O(V^2 \log V + VE)$ steps [?]. Recall that the vertices in the graph are the processors. Calculating the remapping could therefore become rather expensive if a large number of processors are used. Since the solution does not need to be optimal, a heuristic algorithm with a runtime of $O(E)$ was used.

The algorithm starts by generating a sorted list of the similarity matrix S . It then steps through the list and selects the largest value which belongs to an unassigned partition. It was proven in [?] that the algorithm’s solution is always greater than half of the optimal solution, thus it should not be a problem to solve the remapping problem in a sub optimal way. Sorting was implemented (as in the original PLUM paper) by a serial binary radix sort (the matrix S were gathered onto one processor), performing β passes and using 2^r “buckets” for counting. In order to save some memory the sorting was performed per byte of the integer instead of the binary representation. Since each integer is represented by 4 bytes (true even for most 64-bits architectures) the number of passes required was $\beta = 4$, and since each byte have 8 bits the number of “buckets” needed were 2^8 .

However, since the similarity matrix S is of the size $P \times P$ where P is the number of processors, the sorting will have a runtime of $O(P^2)$. This should not cause any problem on a small or medium size parallel computer, as the one used in the fairly old PLUM paper. But after 128-256 processors the complexity of the sorting starts to dominates in the load balancer. To solve this problem, instead of sorting S on one processor we implemented a parallel binary radix sort. The unsorted data of length N was divided into N/P parts which were assigned to the available processors. The internal β sorting phases were only modified with a parallel prefix calculation and a parallel swap phase (when the elements are moved to the new “sorted” locations).

Algorithm 4 Parallel radix sort

► Editor note: *Package `algorithmic.sty` conflicts with `algorithmicx.sty`. Sort out which packages to use for algorithms.*

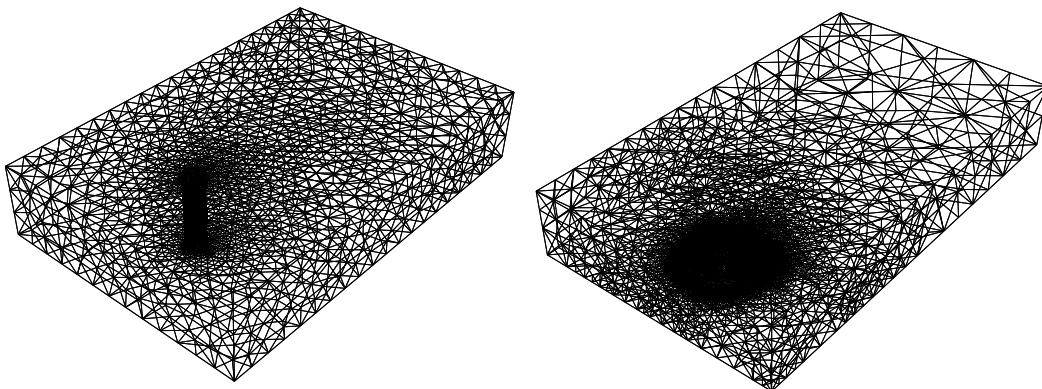


Figure 11.6: Two of the meshes used during evaluation.

11.4.3 Theoretical and experimental analysis

The adaptive refinement method described in this chapter has been successfully tested on a 1024 node Blue Gene/L, each dual-processor node could either be used to run two separate programs (virtual mode) or run in coprocessor mode where one of the processor works as an offload engine, for example handling parts of the message passing stack [?, ?].

As a test problem we used an unstructured mesh and refined all elements inside a small local region, timed mesh refinement and load balancing times. The problem were tested on $P = 32, 128, 512$ nodes both in virtual and coprocessor mode. Since the smallest number of nodes that could be allocated was 32, all speedup results are presented as a comparison against the time for 32 processors. To capture possible communication bottlenecks, three different unstructured meshes were used. First a cylinder mesh with $n_v = 566888$ vertices, secondly a hill with $n_v = 94720$ vertices and finally, the cylinder again but with $n_v = 1237628$ vertices instead.

The regions selected for refinement were around the cylinder and behind the hill. Since these regions already had the most number of elements in the mesh, refinement would certainly result in an workload imbalance. Hence, trigger the load balancing algorithms. In order to analyze the experimental results we used a performance model which decompose the total runtime T into one serial computational cost T_{comp} , and a parallel communication cost T_{comm} .

$$T = T_{comp} + T_{comm} \quad (11.3)$$

The mesh refinement has a local computational costs consisting of iterating over and bisecting all elements marked for refinement, for a mesh with N_c elements $O(N_c/P)$ steps. Communication only occurs when the boundary elements

needs to be exchanged. Thus, each processor would in the worst case communicate with $(P - 1)$ other processors. If we assume that there are N_s shared edges, the total runtime with communication becomes,

$$T_{refine} = O\left(\frac{N_c}{P}\right) \tau_f + (P - 1)(\tau_s + N_s \tau_b) \quad (11.4)$$

where τ_f is the time to perform one (floating point) operation, τ_s is the latency and τ_b the bandwidth. So based on the performance model, more processors would lower the computational time, but in the same time increase the communication time.

The most computationally expensive part of the load balancer is the remapping or assignment of the new partitions. As discussed earlier, we used an heuristic with a runtime of $O(E)$, the number of edges in the bipartite graph. Hence, in the worst case $E \approx P^2$. The sorting phase is linear, and due to the parallel implementation it runs in $O(P)$. Communication time for the parallel prefix calculation is given by, for m data it sends and calculates in m/P steps. Since the prefix consists of 2^r elements, it would take $2^r/P$ steps, and we have to do this for each β sorting phases. In the worst case the reordering phase (during sorting) needs to send away all the elements, thus P communication steps, which gives the total runtime.

$$T_{loadb} = O(P^2)\tau_f + \beta \left(\tau_s + \left(\frac{2^r}{P} + P \right) \tau_b \right) \quad (11.5)$$

According to this, load balancing should not scale well for a large number of processors (due to the $O(P^2)$ term). However the more realistic average case should be $O(P)$. So again, with more processors the runtime could go up due to the communication cost.

If we then compare with the experimental results presented in figure 11.7, we see that the performance degenerates when a large number of processors are used. The question is why? Is it solely due to the increased communication cost? Or is the load balancer's computational cost to blame?

First of all one could observe that when the number of elements per processor is small. The communication costs starts to dominate, see for example the results for the smaller hill mesh (represented by a triangle in the figure). The result is better for the medium size cylinder (represented by a diamond), and even better for the largest mesh (represented by a square). If the load balancing time was the most dominating part, a performance drop should have been observed around 128 - 256 processors. Instead performance generally drops after 256 processors. A possible explanation for this could be the small amount of local elements. Even for the larger 10^6 element mesh, with 1024 processors the number of local elements is roughly 10^3 , which is probably too few to balance the communication costs.

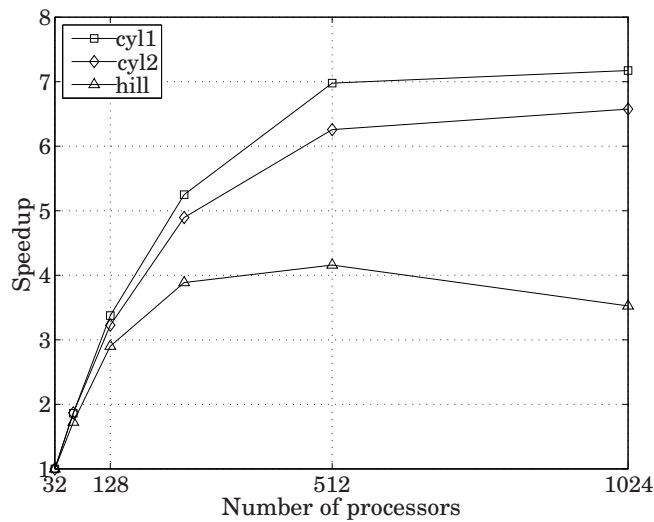


Figure 11.7: Refinement speedup (incl. load balancing)

Processors	Execution time	Speedup
256	33.50	1.0
512	23.19	1.44
1024	19.24	1.74

Table 11.1: Refinement time for a nine million vertices mesh

This illustrates the curse of massively parallel, distributed memory systems. In order to gain anything from the large amount of processors, either one has to have a large local computational cost, or one has to increase the dataset in order to mask the communication latency. To illustrate how large the mesh needs to be, we uniformly refined the larger cylinder, locally refined the same region as before and timed it for 256, 512 and 1024 processors. Now the refinement performance is better, as can be seen in table 11.1. But again performance drops between 512 and 1024 processors.

However, despite the decrease in speedup, one could see that the algorithm seems to scale well, given that the local amount of elements are fairly large. It is close to linear scaling from 32 to 256 processors, and we see no reason for it to not scale equally good between 256 and 1024 processors, given that the local part of the mesh is large enough.

11.5 Summary and outlook

In this chapter we presented some of the challenges with parallel mesh refinement. How these could be solved and finally how these solutions were implemented for a distributed memory system. In the final section some theoretical and experimental performance results were presented and explained. One could observe how well the implementation performs, until the curse of slow interconnect starts to affect the results.

One aspect of the refinement problem that has not been touched upon in this chapter is the mesh quality. The modification done to the longest edge algorithm (see section 11.2.2), unfortunately destroys the good properties of the original recursive algorithm. It was a trade off between efficiency and mesh quality. As mentioned before, the problem with the original algorithm is the efficiency of propagation and global termination detection. Currently our work is focused in overcoming these problems, implementing an efficient parallel refinement method with good quality aspects, which also performs well for thousands of processors.

Part II

Implementation

CHAPTER 12

DOLFIN: A C++/Python Finite Element Library

By Anders Logg and Garth N. Wells

Chapter ref: [**logg-2**]

Overview and tutorial of DOLFIN.

CHAPTER 13

FFC: A Finite Element Form Compiler

By Anders Logg and possibly others

Chapter ref: **[logg-1]**

► Editor note: *Oelgaard/Wells might possibly write something here on FFC quadrature evaluation, or it will be included in a separate chapter. Marie will possibly write something here on $H(\text{div})/H(\text{curl})$, or it will be included in a separate chapter.*

Overview and tutorial of FFC.

Ferari: An Optimizing Compiler for Variational Forms

By Robert C. Kirby and Anders Logg

Chapter ref: **[kirby-3]**

We describe the implementation of an optimizing compiler for variational forms based on the discrete optimization methodology described in an earlier chapter. The major issues are communicating reference tensors from FFC to the optimizer, FErari, performing the actual optimization, and communicating abstract syntax for the optimized algorithm back to FFC for code generation. We include some results indicating the kinds of speedups that may be achieved and what factors influence the effectiveness of the optimizations.

FIAT: Numerical Construction of Finite Element Basis Functions

By Robert C. Kirby

Chapter ref: **[kirby-2]**

15.1 Introduction

The FIAT project [?, ?] implements the mathematical framework described in Chapter ?? as a Python package, working mainly in terms of numerical linear algebra. Although an implementation floating-point arithmetic presents some challenges relative to symbolic computation, it can allow greater efficiency and consume fewer resources, especially for high order elements. To obtain efficiency in Python, the compute-intensive operations are expressed in terms of numerical linear algebra and performed using the widely distributed `numpy` package.

The FIAT project is one of the first FEniCS projects, providing the basis function back-end for `ffc` and enabling high-order H^1 , $H(\text{div})$ and $H(\text{curl})$ elements. It is widely distributed, with downloads on every inhabited continent and in over sixty countries, averaging about 100 downloads per month.

This chapter works in the context of a Ciarlet triple (K, P, N) [?], where K is a fixed reference domain, typically a triangle or tetrahedron. P is a finite-dimensional polynomial space, though perhaps vector- or tensor-valued and not coincident with polynomials of some fixed degree. $N = \{\ell_i\}_{i=1}^{|P|}$ is a set of linear functionals spanning P' . Recalling Chapter [], the goal is first to enumerate a convenient basis $\{\phi_i\}_{i=1}^{|P|}$ for P and then to form a generalized Vandermonde

system

$$VA = I,$$

where $V_{ij} = \ell_i(\phi_j)$. The columns of $A = V^{-1}$ store the expansion coefficients of the nodal basis for (K, P, N) in terms of the basis $\{\phi_i\}$.

15.2 Prime basis: Collapsed-coordinate polynomials

High order polynomials in floating-point arithmetic require stable evaluation algorithms. FIAT uses the so-called collapsed-coordinate polynomials [?] on the triangle and tetrahedra. Let $P_i^{\alpha,\beta}(x)$ denote the Jacobi polynomial of degree i with weights α and β . On the triangle K with vertices $(-1, 1)$, $(1, -1)$, $(-1, 1)$, the polynomials are of the form

$$D^{p,q}(x, y) = P_p^{0,0}(\eta_1) \left(\frac{1 - \eta_2}{2} \right)^i P_j^{2i+1,0}(\eta_2)$$

where

$$\begin{aligned} \eta_1 &= 2 \left(\frac{1+x}{1-y} \right) - 1 \\ \eta_2 &= y \end{aligned}$$

is called the collapsed-coordinate mapping is a singular transformation between the triangle and the biunit square. The set $\{D^{p,q}(x, y)\}_{p,q \geq 0}^{p+q \leq n}$ forms a basis for polynomials of degree n . Moreover, they are orthogonal in the $L^2(K)$ inner product. Recently [?], it has been shown that these polynomials may be computed directly on the triangle without reference to the singular mapping. This means that no special treatment of the singular point is required, allowing use of standard automatic differentiation techniques to compute derivatives.

The recurrences are obtained by rewriting the polynomials as

$$D^{p,q}(x, y) = \chi^p(x, y) \psi^{p,q}(y),$$

where

$$\chi^p(x, y) = P_p^{0,0}(\eta_1) \left(\frac{1 - \eta_2}{2} \right)^p$$

and

$$\psi^{p,q}(y) = P_q^{2p+1,0}(\eta_2) = P_q^{2p+1,0}(y).$$

This representation is not separable in η_1 and η_2 , which may seem to be a drawback to readers familiar with the usage of these polynomials in spectral methods. However, they do still admit sum-factorization techniques. More importantly for present purposes, each χ^p is in fact a polynomial in x and y and may be computed by recurrence. $\psi^{p,q}$ is just a Jacobi polynomial in y and so has a well-known three-term recurrence. The recurrences derived in [?] are presented in Algorithm 5

Algorithm 5 Computes all triangular orthogonal polynomials up to degree d by recurrence

```

1:  $D^{0,0}(x, y) := 1$ 
2:  $D^{1,0}(x, y) := \frac{1+2x+y}{2}$ 
3: for  $p \leftarrow 1, d-1$  do
4:    $D^{p+1,0}(x, y) := \left(\frac{2p+1}{p+1}\right) \left(\frac{1+2x+y}{2}\right) D^{p,0}(x, y) - \left(\frac{p}{p+1}\right) \left(\frac{1-y}{2}\right)^2 D^{p-1,0}(x, y)$ 
5: end for
6: for  $p \leftarrow 0, d-1$  do
7:    $D^{p,1}(x, y) := D^{p,0}(x, y) \left(\frac{1+2p+(3+2p)y}{2}\right)$ 
8: end for
9: for  $p \leftarrow 0, d-1$  do
10:  for  $q \leftarrow 1, d-p-1$  do
11:     $D^{p,q+1}(x, y) := (a_q^{2p+1,0}y + b_q^{2p+1,0}) D^{p,q}(x, y) - c_q^{2p+1,0} D^{p,q-1}(x, y)$ 
12:  end for
13: end for

```

15.3 Representing polynomials and functionals

Even using recurrence relations and `numpy` vectorization for arithmetic, further care is required to optimize performance. In this section, standard operations on polynomials will be translated into vector operations, and then batches of such operations cast as matrix multiplication. This helps eliminate the interpretive overhead of Python while moving numerical computation into optimized library routines, since `numpy.dot` wraps level 3 BLAS and other functions such as `numpy.svd` wrap relevant LAPACK routines.

Since polynomials and functionals over polynomials both form vector spaces, it is natural to represent each of them as vectors representing expansion coefficients in some basis. So, let $\{\phi_i\}$ be the set of polynomials described above.

Now, any $p \in P$ is written as a linear combination of the basis functions $\{\phi_i\}$. Introduce a mapping \mathcal{R} from P into $\mathbb{R}^{|P|}$ by taking the expansion coefficients of p in terms of $\{\phi_i\}$. That is,

$$p = \mathcal{R}(p)_i \phi_i,$$

where summation is implied over i .

A polynomial p may then be evaluated at a point x as follows. Let Φ_i be the basis functions tabulated at x . That is,

$$\Phi_i = \phi_i(x). \tag{15.1}$$

Then, evaluating p follows by a simple dot product:

$$p(x) = \mathcal{R}(p)_i \Phi_i. \tag{15.2}$$

More generally in FIAT, a set of polynomials $\{p_i\}$ will need to be evaluated simultaneously, such as evaluating all of the members of a finite element basis. The coefficients of the set of polynomials may be stored in the rows of a matrix C , so that

$$C_{ij} = \mathcal{R}(p_i)_j.$$

Tabulating this entire set of polynomials at a point x is simply obtained by matrix-vector multiplication. Let Φ_i be as in (15.1). Then,

$$p_i(x) = C_{ij}\Phi_j.$$

The basis functions are typically needed at a set of points, such as those of a quadrature rule. Let $\{x_j\}$ now be a collection of points in K and let

$$\Phi_{ij} = \phi_i(x_j),$$

where the rows of Φ run over the basis functions and the columns over the collection of points. As before, the set of polynomials may be tabulated at all the points by

$$p_i(x_j) = C_{ik}\Phi_{kj},$$

which is just the matrix product $C\Phi$ and may be efficiently carried out by a library operation, such as the `numpy.dot` wrapper to level 3 BLAS.

Finite element computation also requires the evaluation of derivatives of polynomials. In a symbolic context, differentiation presents no particular difficulty, but working in a numerical context requires some special care.

For some differential operator D , the derivatives $D\phi_i$ are computed at a point x , any polynomial $p = \mathcal{R}(p)_i\phi_i$ may be differentiated at x by

$$Dp(x) = \mathcal{R}(p)_i(D\phi_i),$$

which is exactly analogous to (15.2). By analogy, sets of polynomials may be differentiated at sets of points just like evaluation.

The formulae in Algorithm 5 and their tetrahedral counterpart are fairly easy to differentiate, but derivatives may also be obtained through automatic differentiation. Some experimental support for this using the AD tools in Scientific Python has been developed in an unreleased version of FIAT.

The released version of FIAT currently evaluates derivatives in terms of linear operators, which allows the coordinate singularity in the standard recurrence relations to be avoided. For each Cartesian partial derivative $\frac{\partial}{\partial x_k}$, a matrix D^k is calculated such that

$$\mathcal{R}\left(\frac{\partial p}{\partial x_k}\right)_i = D_{ij}^k \mathcal{R}(p)_j.$$

Then, derivatives of sets of polynomials may be tabulated by premultiplying the coefficient matrix C with such a D^k matrix.

This paradigm may also be extended to vector- and tensor-valued polynomials, making use of the multidimensional arrays implemented in `numpy`. Let P be a space of scalar-valued polynomials and $n > 0$ an integer. Then, a member of $(P)^m$, a vector with m components in P , may be represented as a two-dimensional array. Let $p \in (P)^m$ and p^j be the j^{th} component of p . Then $p = \mathcal{R}(p)_{jk}\phi_k$, so that $\mathcal{R}(p)_{jk}$ is the coefficient of ϕ_k for p^j .

The previous discussion of tabulating collections of functions at collections of points is naturally extended to this context. If $\{p_i\}$ is a set of members of P^m , then their coefficients may be stored in an array C_{ijk} , where C_i is the two-dimensional array $\mathcal{R}(p)_{jk}$ of coefficients for p_i . As before, $\Phi_{ij} = \phi_i(x_j)$ contains the of basis functions at a set of points. Then, the j^{th} component of v_i at the point x_k is naturally given by a three-dimensional array

$$p_i(x_k)^j = C_{ijl}\phi_{lk}.$$

Psychologically, this is just a matrix product if C_{ijl} is stored contiguously in generalized row-major format, and the operation is readily cast as dense matrix multiplication.

Returning for the moment to scalar-valued polynomials, linear functionals may also be represented as Euclidean vectors. Let $\ell : P \rightarrow \mathbb{R}$ be a linear functional. Then, for any $p \in P$,

$$\ell(p) = \ell(\mathcal{R}(p)_i\phi_i) = \mathcal{R}(p)_i\ell(\phi_i),$$

so that ℓ acting on p is determined entirely by its action on the basis $\{\phi_i\}$. As with \mathcal{R} , define $\mathcal{R}' : P' \rightarrow \mathbb{R}^{|P|}$ by

$$\mathcal{R}'(\ell)_i = \ell(\phi_i),$$

so that

$$\ell(p) = \mathcal{R}'(\ell)_i\mathcal{R}(p)_i.$$

Note that the inverse of \mathcal{R}' is the Banach-space adjoint of \mathcal{R} .

Just as with evaluation, sets of linear functionals can be applied to sets of functions via matrix multiplication. Let $\{\ell_i\}_{i=1}^N \subset P'$ and $\{u_i\}_{i=1}^N \subset P$. The functionals are represented by a matrix

$$L_{ij} = \mathcal{R}'(\ell_i)_j$$

and the functions by

$$C_{ij} = \mathcal{R}(u_i)_j$$

Then, evaluating all of the functionals on all of the functions is computed by the matrix product

$$A_{ij} = L_{ik}C_{jk}, \tag{15.3}$$

or $A = LC^t$. This is especially useful in the setting of the next section, where the basis for the finite element space needs to be expressed as a linear combination of orthogonal polynomials.

Also, the formalism of \mathcal{R}' may be generalized to functionals over vector-valued spaces. As before, let P be a space of degree n with basis $\{\phi_i\}_{i=1}^{|P|}$ and to each $v \in (P)^m$ associate the representation $v^i = \mathcal{R}(v)_{ij}\phi_j$. In this notation, $v = \mathcal{R}(v)_{ij}\phi_j$ is the vector indexed over i . For any functional $\ell \in ((P)^m)'$, a representation $\mathcal{R}'(\ell)_{ij}$ must be defined such that

$$\ell(v) = \mathcal{R}'(\ell)_{ij}\mathcal{R}(v)_{ij},$$

with summation implied over i and j . To determine the representation of $\mathcal{R}'(f)$, let e^j be the canonical basis vector with $(e^j)_i = \delta_{ij}$ and write

$$\begin{aligned} \ell(v) &= \ell(\mathcal{R}_{ij}\phi_j) \\ &= \ell(\mathcal{R}(v)_{ij}\delta_{ik}e^k\phi_j) \\ &= \ell(\mathcal{R}(v)_{ij}e^i\phi_j) \\ &= \mathcal{R}(v)_{ij}\ell(e^i\phi_j). \end{aligned} \tag{15.4}$$

From this, it is seen that $\mathcal{R}'(\ell)_{ij} = \ell(e^i\phi_j)$.

Now, let $\{v_i\}_{i=1}^N$ be a set of vector-valued polynomials and $\{\ell_i\}_{i=1}^M$ a set of linear functionals acting on them. The polynomials may be stored by a coefficient tensor $C_{ijk} = \mathcal{R}(v_i)_{jk}$. The functionals may be represented by a tensor $L_{ijk} = \mathcal{R}'(\ell_i)_{jk}$. The matrix $A_{ij} = \ell_i(v_j)$ is readily computed by the contraction

$$A_{ij} = L_{ikl}C_{jkl}.$$

Despite having three indices, this calculation may still be performed by matrix multiplication. Since `numpy` stores arrays in row-major format, a simple reshaping may be performed without data motion so that $A = \tilde{L}\tilde{C}^t$, for \tilde{L} and \tilde{C} reshaped to two-dimensional arrays by combining the second and third axes.

15.4 Other polynomial spaces

Many of the complicated elements that motivate the development of a tool like FIAT polynomial spaces that are not polynomials of some complete degree (or vectors or tensors of such). Once a process for providing bases for such spaces is described, the techniques of the previous section may be applied directly. Most finite element polynomial spaces are described either by adding a few basis functions to some polynomials of complete degree or else by constraining such a space by some linear functionals.

15.4.1 Supplemented polynomial spaces

A classic example of the first case is the Raviart-Thomas element, where the function space of order r is

$$RT_r = (P_r(K))^d \oplus \left(\tilde{P}_r(K) \right) x,$$

where $x \in \mathbb{R}^d$ is the coordinate vector and \tilde{P}_r is the space of homogeneous polynomials of degree r . Given any basis $\{\phi_i\}$ for $P_r(K)$ such as the Dubiner basis, it is easy to obtain a basis for $(P_r(K))^d$ by taking vectors where one component is some ϕ_i and the rest are zero. The issue is obtaining a basis for the entire space.

Consider the case $d = 2$ (triangles). While monomials of the form $x^i y^{r-i}$ span the space of homogeneous polynomials, they are subject to ill-conditioning in numerical computations. On the other hand, the Dubiner basis of order r , $\{\phi_i\}_{i=1}^{|P_r|}$ may be ordered so that the last $r + 1$ functions, $\{\phi_i\}_{i=|P_r|-r}^{|P_r|}$, have degree exactly r . While they do not span \tilde{P}_r , the span of $\{x\phi_i\}_{i=|P_r|-r}^{|P_r|}$ together with a basis for $(P_r(K))^2$ does span RT_r .

So, this gives a basis for the Raviart-Thomas space that can be evaluated and differentiated using the recurrence relations described above. A similar technique may be used to construct elements that consist of standard elements augmented with some kind of bubble function, such as the PEERS element of elasticity or MINI element for Stokes flow.

15.4.2 Constrained polynomial spaces

An example of the second case is the Brezzi-Douglas-Fortin-Marini element [?]. Let $\mathcal{E}(K)$ be the set of dimension one cofacets of K (edges in 2d, faces in 3d). Then the function space is

$$BDFM_r(K) = \{u \in (P_r(K))^d : u \cdot n|_\gamma \in P_{r-1}(\gamma), \gamma \in \mathcal{E}(K)\}$$

This space is naturally interpreted as taking a function space, $(P_r(K))^d$, and imposing linear constraints. For the case $d = 2$, there are exactly three such constraints. For $\gamma \in \mathcal{E}(K)$, let μ^γ be the Legendre polynomial of degree r mapped to γ . Then, if a function $u \in (P_r(K))^d$, it is in $BDFM_r(K)$ iff

$$\int_\gamma (u \cdot n) \mu^\gamma ds = 0$$

for each $\gamma \in \mathcal{E}(K)$.

Number the edges by $\{\gamma_i\}_{i=1}^3$ and introduce linear functionals $\ell_i(u) = \int_{\gamma_i} (u \cdot n) \mu^{\gamma_i} ds$. Then,

$$BDFM_r(K) = \cap_{i=1}^3 \text{null}(\ell_i).$$

This may naturally be cast into linear algebra and so evaluated with LAPACK. Following the techniques for constructing Vandermonde matrices, a *constraint matrix* may be constructed. Let $\{\bar{\phi}_i\}$ be a basis for $(P_r(K))^2$. Define the $3 \times |(P_r)|^2$ matrix

$$C_{ij} = \ell_i(\phi_j).$$

Then, a basis for the null space of this matrix is constructed using the singular value decomposition [?]. The vectors of this null-space basis are readily seen to contain the expansion coefficients of a basis for $BDFM_r$ in terms of a basis for $P_r(K)^2$. With this basis in hand, the nodal basis for $BDFM_r(K)$ is obtained by constructing the generalized Vandermonde matrix.

This technique may be generalized to three dimensions, and it also applies to Nédélec [?], Arnold-Winther [?], Mardal-Tai-Winther [?], and many other elements.

15.5 Conveying topological information to clients

Most of this chapter has provided techniques for constructing finite element bases and evaluating and differentiating them. FIAT must also indicate which degrees of freedom are associated with which entities of the reference element. This information is required when local-global mappings are generated by a form compiler such as `ffc`.

The topological information is provided by a graded incidence relation and is similar to the presentation of finite element meshes in [?]. Each entity in the reference element is labeled by its topological dimension (e.g. 0 for vertices and 1 for edges), and then the entities of the same dimension are ordered by some convention. To each entity, a list of the local nodes is associated. For example, the reference triangle with entities labeled is shown in Figure 15.5, and the cubic Lagrange triangle with nodes in the dual basis labeled is shown in Figure 15.5.

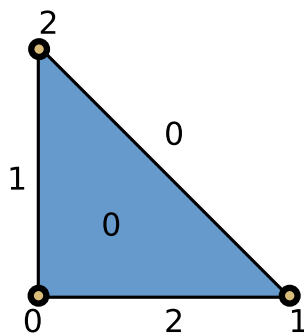


Figure 15.1: The reference triangle, with vertices, edges, and the face numbered.

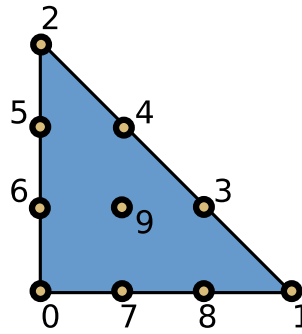


Figure 15.2: The cubic Lagrange triangle, with nodes in the dual basis labelled.

For this example, the graded incidence relation is stored as

$$\begin{aligned} & \{ 0: \{ 0: [0] , \\ & \quad 1: [1] , \\ & \quad 2: [2] \} , \\ 1: & \{ 0: [3 , 4] , \\ & \quad 1: [5 , 6] , \\ & \quad 2: [7 , 8] \} , \\ 2: & \{ 0: [9] \} \} \end{aligned}$$

15.6 Functional evaluation

In order to construct nodal interpolants or strongly enforce boundary conditions, FIAT also provides information to numerically evaluate linear functionals. These rules are typically exact for a certain degree polynomial and only approximate on general functions. For scalar functions, these rules may be represented by a collection of points and corresponding weights $\{x_i\}, \{w_i\}$ so that

$$\ell(f) \approx w_i f(x_i).$$

For example, pointwise evaluation at a point x is simply represented by the coordinates of x together with a weight of 1. If the functional is an integral moment, such as

$$\ell(f) = \int_K g f \, dx,$$

then the points $\{x_i\}$ will be those of some quadrature rule and the weights will be $w_i = \omega_i g(x_i)$, where the ω_i are the quadrature weights.

This framework is extended to support vector- and tensor-valued function spaces, by including a component corresponding to each point and weight. If v is

a vector-valued function and v_α is its component, then functionals are written in the form

$$\ell(v) \approx w_i v_{\alpha_i}(x_i),$$

so that the sets of weights, components, and points must be conveyed to the client.

This framework may also support derivative-based degrees of freedom by including a multiindex at each point corresponding to a particular partial derivative.

15.7 Overview of fundamental class structure

Many FEniCS users will never directly use FIAT; for them, interaction will be moderated through a form compiler such as `ffc`. Others will want to use the FIAT basis functions in other contexts. At a basic level, a user will access FIAT through top-level classes such as `Lagrange` and `RaviartThomas` that implement the elements. Typically, the class constructors accept the reference element and order of function space as arguments. This gives an interface that is parametrized by dimension and degree. The classes such as `Lagrange` derive from a base class `FiniteElement` that provides access to the three components of the Ciarlet triple.

The currently released version of FIAT stores the reference element as a flag indicating the simplex dimension, although a development version provides an actual class describing reference element geometry and topology. This will allow future releases of FIAT to be parametrized over the particular reference element shape and topology.

The function space P is modelled by the base class `PolynomialSet`, while contains a rule for constructing the base polynomials ϕ_i (e.g. the Dubiner basis) and a multidimensional array of expansion coefficients for the basis of P . Special subclasses of this provide (possibly array-valued) orthogonal bases as well as the rules for constructing supplemented and constrained bases. These classes provide mechanisms for tabulating and differentiating the polynomials at input points as well as basic queries such as the dimension of the space.

The set of finite element nodes is similarly modeled by a class `DualBasis`. This provides the functionals of the dual basis as well as their connection to the reference element facets. The functionals are modeled by a `FunctionalSet` object, which is a collection of `Functional` objects. Each `Functional` object contains a reference to the `PolynomialSet` over which it is defined and the array of coefficients representing it and owns a `FunctionalType` class providing the information described in the previous section. The `FunctionalSet` class batches these coefficients together in a single large array.

The constructor for the `FiniteElement` class takes a `PolynomialSet` modeling the starting basis and a `DualBasis` defined over this basis and constructs

a new `PolynomialSet` by building and inverting the generalized Vandermonde matrix.

Beyond this basic finite element structure, FIAT provides quadrature such as Gauss-Jacobi rules in one dimension and collapsed-coordinate rules in higher dimensions. It also provides routines for constructing lattices of points on each of the reference element shapes and their facets.

In the future, FIAT will include the developments discussed already (more general reference element geometry/topology and automatic differentiation). Automatic differentiation will make it easier to construct finite elements with derivative-type degrees of freedom such as Hermite, Morley, and Argyris. Additionally, we hope to expand the collection of quadrature rules and provide more advanced point distributions, such as Warburton's warp-blend points [?].

Instant: Just-in-Time Compilation of C/C++ Code in Python

By Ilmar M. Wilbers, Kent-Andre Mardal and Martin S. Alnæs

Chapter ref: **[wilbers]**

16.1 Introduction

Instant is a small Python module for just-in-time compilation (or inlining) of C/C++ code based on SWIG [?] and Distutils¹. Just-in-time compilation can significantly speed up, e.g., your NumPy [?] code in a clean and readable way. This makes Instant a very convenient tool in combination with code generation. Before we demonstrate the use of Instant in a series of examples, we briefly step through the basic ideas behind the implementation. Instant relies on SWIG for the generation of wrapper code needed for making the C/C++ code usable from Python [?]. SWIG is a mature and well-documented tool for wrapping C/C++ code in many languages. We refer to its website for a comprehensive user manual and we also discuss some common tricks and troubles in Chapter [?]. The code to be inlined, in addition to the wrapper code, is then compiled into a Python extension module (a shared library with functionality as specified by the Python C-API) by using Distutils. To check whether the C/C++ code has changed since the last execution, Instant computes the SHA1 sum² of the code and compares it to the SHA1 checksum of the code used in the previous execution. Finally,

¹<http://www.python.org/doc/2.5.2/lib/module-distutils.html>

²<http://www.apps.ietf.org/rfc/rfc3174.html>

Instant has implemented a set of SWIG typemaps, allowing the user to transfer NumPy arrays between the Python code and the C/C++ code.

There exist several packages that are similar to Instant. Worth mentioning here are Weave [?], Cython [?], and F2PY [?]. Weave allows us to inline C code directly in our Python code. Unlike Instant, Weave does not require the specification of the function signature and the return argument. For specific examples of Weave and the other mentioned packages, we refer to [?, ?]. Weave is part of SciPy [?]. F2PY is currently part of NumPy, and is primarily intended for wrapping Fortran code. F2PY can also be used for wrapping C code. Cython is a rather new project, branched from the more well-known Pyrex project [?]. Cython is attractive because of its integration with NumPy arrays. Cython differs from the other projects by being a programming language of its own. Cython extends Python with concepts such as static typing, hence allowing the user to incrementally speed up the code.

Instant accepts plain C/C++. This makes it particularly attractive to combine Instant with tools capable of generating C/C++ code such as FFC [?], SFC [?], Swiginac [?], and Sympy [?]. In fact, tools like these have been the main motivation behind Instant, and both FFC and SFC employ Instant. Instant is released under a BSD license, see the file LICENSE in the source directory.

In this chapter we will begin with several examples in Section 16.2. Section (16.3) explains how Instant works, while Section (16.4) gives a detailed description of the API.

16.2 Examples

All code from the examples in this section can be found online.

³ We will refer to this location as `$examples`.

16.2.1 *Installing Instant*

Before trying to run the examples, you need to install Instant. The latest Instant release can be downloaded from the FEniCS website⁴. It is available both as a source code tarball and as a Debian package. In addition, the latest source code can be checked out using Mercurial⁵:

```
hg clone http://www.fenics.org/hg/instant
```

Installing Instant from the source code is done with a regular Distutils script, i.e,

³<http://www.fenics.org/pub/documents/book/instant/examples>

⁴<http://www.fenics.org/wiki/Download>

⁵<http://www.selenic.com/mercurial/wiki/>

```
python setup.py install
```

After successfully installing Instant, one can verify the installation by running the scripts `run_tests.py` followed by `rerun_tests.py` in the `tests`-directory of the source code. The first will run all the examples after having cleaned the Instant cache, the second will run all examples using the compiled modules found in the Instant cache from the previous execution.

16.2.2 *Hello World*

Our first example demonstrate the usage of Instant in a very simple case:

```
from instant import inline
c_code = r'''
double add(double a, double b)
{
    printf("Hello world! C function add is being called...\n");
    return a+b;
}'''
add_func = inline(c_code)
sum = add_func(3, 4.5)
print 'The sum of 3 and 4.5 is', sum
```

Here Instant will wrap the C-function `add` into a Python extension module by using SWIG and Distutils. The inlined function is written in standard C. SWIG supports almost all of C and C++, including object orientation and templates. When running this Python snippet the first time, compiling the C code takes a few seconds. Next time we run it, however, the compilation is omitted, given that no changes are made to the C source code.

Note that a raw string is used in this example, to avoid Python interpreting escape sequences such as `'\n'`. Alternatively, special characters can be escaped using a backslash.

Although Instant notifies the user when it is compiling, it might sometimes be necessary, e.g. when debugging, to see the details of the Instant internals. We can do this by setting the logging level before calling any other Instant functions:

```
from instant import output
output.set_logging_level('DEBUG')
```

The intrinsic Python module `logging` is used. First, the `build_function` arguments are displayed, whereafter the different steps performed by Instant are shown in detail, e.g whether the module is found in cache and the arguments to the Distutils file when building the module. This example can be found in the file `$examples/ex1.py`.

16.2.3 NumPy Arrays

One basic problem with wrapping C and C++ code is how to handle dynamically allocated arrays. Arrays allocated dynamically are typically represented in C/C++ by a pointer to the first element of an array and a separate integer variable holding the array size. In Python the array variable is itself an object contains the data array, array size, type information etc. However, a pointer in C/C++ does not necessarily represent an array. Therefore, SWIG provides the `typemap` functionality that allows the user to specify a mapping between Python and C/C++ types. We will not go into details on `typemaps` in this chapter, but the reader should be aware that it is a powerful tool that may greatly enhance your code, but also lead to mysterious bugs when used wrongly. `Typemaps` are discussed in Chapter [?] and at length at the SWIG webpage. In this chapter, it is sufficient to illustrate how to deal with arrays in Instant using the NumPy module. More details on how Instant NumPy arrays can be found in Section 16.3.1.

16.2.4 Ordinary Differential Equations

We introduce a solver for an ordinary differential equation (ODE) modeling blood pressure by using a Windkessel model. The ODE is as follows:

$$\frac{d}{dt}p(t) = BQ(t) - Ap(t), \quad t \in (0, 1), \quad (16.1)$$

$$p(0) = p_0. \quad (16.2)$$

Here $p(t)$ is the blood pressure, $Q(t)$ is the volume flux of blood, A is ... and B is An explicit scheme is:

$$p_i = p_{i-1} + \Delta t(BQ_i - Ap_{i-1}), \quad \text{for } i = 1, \dots, N-1, \quad (16.3)$$

$$p_0 = p_0. \quad (16.4)$$

The scheme can be implemented in Python as follows using NumPy arrays:

```
def time_loop_py(p, Q, A, B, dt, N, p0):
    p[0] = p0
    for i in range(1, N):
        p[i] = p[i-1] + dt*(B*Q[i] - A*p[i-1])
```

The C code given as argument to the Instant function `inline_with_numpy` looks like:

```
void time_loop_c(int n, double* p,
                int m, double* Q,
                double A, double B,
```

```

        double dt, int N, double p0)
{
    if ( n != m || N != m )
    {
        printf("n, m and N should be equal\n");
        return;
    }

    p[0] = p0;
    for (int i=1; i<n; i++)
    {
        p[i] = p[i-1] + dt*(B*Q[i] - A*p[i-1]);
    }
}

```

In this example, `(int n, double* p)` represents an array of doubles with length n . However, this can not be determined by the function signature:

```
void time_loop_C(int n, double* p, int m, double* Q, ...)
```

For example, `double* p` may be an array of length m or it may simply be output. In Instant you can specify 1-dimensional arrays as follows:

```
time_loop_c = inline_with_numpy(c_code,
                                arrays = [['n', 'p'],
                                           ['m', 'Q']])
```

Here we tell Instant that `(int n, double* p)` and `(int m, double* Q)` are NumPy arrays (and Instant then employs a few typemaps). We may then call the `time_loop` function as follows:

```
time_loop_c(p, Q, 1.0, 1.0, 1.0/(N-1), N, 1.0)
```

In the above example we obtain a speed-up of about a factor 400 when using 100000 time steps compared to the pure Python with NumPy version, see Table 16.1. This is about the same as a pure C program. The result of solving the ODE can be seen in Figure 38.1. The comparison between NumPy and Instant is not really fair, as NumPy primarily gives a speed-up for code that can be vectorized, something that is not the case with our current ODE. In fact, utilizing pure Python lists instead of NumPy arrays, reduces the speed-up to a factor 100. For code that can be vectorized, the speed-up is about one order of magnitude when we use Instant instead [?].

The complete code for this example can be found in `$examples/ex2.py`

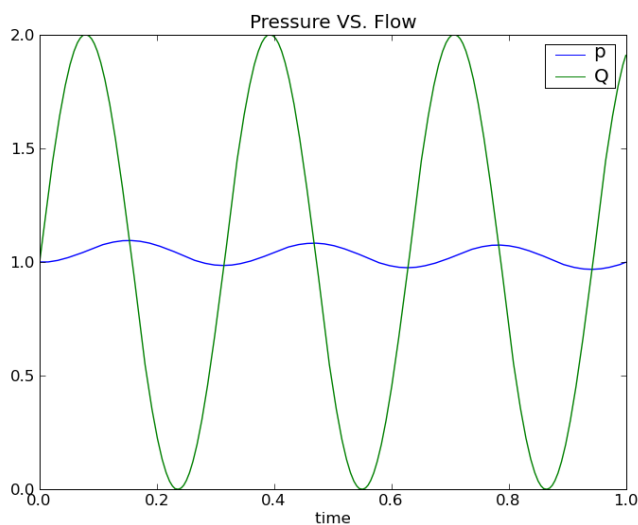


Figure 16.1: Plot of pressure and blood volume flux computed by solving the Windkessel model.

N	100	1000	10000	100000	1000000
CPU time with NumPy	3.9e-4	3.9e-3	3.8e-2	3.8e-1	3.8
CPU time with Python	0.7e-4	0.7e-3	0.7e-2	0.7e-1	0.7
CPU time with Instant	5.0e-6	1.4e-5	1.0e-4	1.0e-3	1.1e-2
CPU time with C	4.0e-6	1.1e-5	1.0e-4	1.0e-3	1.1e-2

Table 16.1: CPU times of Windkessel model for different implementations (in seconds).

16.2.5 Numpy Arrays and OpenMP

It is easy to speed up code on parallel computers with OpenMP. We will not describe OpenMP in any detail here, the reader is referred to [?]. However, note that preprocessor directives like `#pragma omp ...` are OpenMP directives and that OpenMP functions start with `omp`. In this example, we want to solve a standard 2-dimensional wave equation in a heterogeneous medium with local wave velocity k :

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot [k \nabla u]. \quad (16.5)$$

We set the boundary condition to $u = 0$ for the whole boundary of a rectangular domain $\Omega = (0, 1) \times (0, 1)$. Further, u has the initial value $I(x, y)$ at $t = 0$ while $\partial u / \partial t = 0$. We solve the wave equation using the following finite difference

scheme:

$$\begin{aligned}
 u_{i,j}^l = & \left(\frac{\Delta t}{\Delta x} \right)^2 [k_{i+\frac{1}{2},j}(u_{i+1,j} - u_{i,j}) - k_{i-\frac{1}{2},j}(u_{i,j} - u_{i-1,j})]^{l-1} \\
 & + \left(\frac{\Delta t}{\Delta y} \right)^2 [k_{i,j+\frac{1}{2}}(u_{i,j+1} - u_{i,j}) - k_{i,j-\frac{1}{2}}(u_{i,j} - u_{i,j-1})]^{l-1}.
 \end{aligned} \tag{16.6}$$

Here, $u_{i,j}^l$ represents u at the grid point x_i and y_j at time level t_l , where

$$\begin{aligned}
 x_i &= i\Delta x, i = 0, \dots, n \\
 y_j &= j\Delta y, j = 0, \dots, m \text{ and} \\
 t_l &= l\Delta t,
 \end{aligned}$$

Also, $k_{i+\frac{1}{2},j}$ is short for $k(x_{i+\frac{1}{2}}, y_j)$.

The code for calculating the next time step using OpenMP looks like:

```

void stencil(double dt, double dx, double dy,
            int ux, int uy, double* u,
            int umx, int umy, double* um,
            int kx, int ky, double* k,
            int upn, double* up){
#define index(u, i, j) u[(i)*m + (j)]
    int i=0, j=0, m = ux, n = uy;
    double hx, hy, k_c, k_ip, k_im, k_jp, k_jm;
    hx = pow(dt/dx, 2);
    hy = pow(dt/dy, 2);
    j = 0;    for (i=0; i<m; i++) index(up, i, j) = 0;
    j = n-1; for (i=0; i<m; i++) index(up, i, j) = 0;
    i = 0;    for (j=0; j<n; j++) index(up, i, j) = 0;
    i = m-1; for (j=0; j<n; j++) index(up, i, j) = 0;
    #pragma omp for
    for (i=1; i<m-1; i++){
        for (j=1; j<n-1; j++){
            k_c = index(k, i, j);
            k_ip = 0.5*(k_c + index(k, i+1, j));
            k_im = 0.5*(k_c + index(k, i-1, j));
            k_jp = 0.5*(k_c + index(k, i, j+1));
            k_jm = 0.5*(k_c + index(k, i, j-1));
            index(up, i, j) = 2*index(u, i, j) - index(um, i, j) +
                hx*(k_ip*(index(u, i+1, j) - index(u, i, j)) -
                    k_im*(index(u, i, j) - index(u, i-1, j))) +
                hy*(k_jp*(index(u, i, j+1) - index(u, i, j)) -
                    k_jm*(index(u, i, j) - index(u, i, j-1)));
        }
    }
}
    
```

```

    }
}
}

```

We also need to add the OpenMP header `omp.h` and compile with the flag `-fopenmp` and link with the OpenMP shared library, e.g. `libgomp.so` for Linux (specified with `-lgomp`). This can be done as follows:

```

instant_ext = \
    build_module(code=c_code,
                 system_headers=['numpy/arrayobject.h',
                                'omp.h'],
                 include_dirs=[numpy.get_include()],
                 init_code='import_array();',
                 cppargs=['-fopenmp'],
                 lddargs=['-lgomp'],
                 arrays=[['ux', 'uy', 'u'],
                        ['umx', 'umy', 'um'],
                        ['kx', 'ky', 'k'],
                        ['upn', 'up', 'out']])

```

Note that the arguments `include_headers`, `init_code`, and the first element of `system_headers` could have been omitted had we chosen to use `inline_module_with_numpy` instead of `build_module`. We could also have used `inline_with_numpy`, which would have returned only the function, not the whole module. For more details, see the next section. The complete code can be found in `$examples/ex3.py`. It might very well be possible to write more efficient code for many of these examples, but the primary objective is to exemplify different Instant features.

16.3 Instant Explained

The previous section concentrated on the usage of Instant and it may appear mysterious how it actually works since it is unclear what files that are made during execution and where they are located. In this section we explain this.

We will again use our first example, but this time with the keyword argument `modulename` set explicitly. The file can be found under `$examples/ex4.py`:

```

from instant import inline
code = r'''
double add(double a, double b)
{

```

```

    printf("Hello world! C function add is being called...\n");
    return a+b;
}'''
add_func = inline(code, modulename='ex4_cache')
sum = add_func(3, 4.5)
print 'The sum of 3 and 4.5 is', sum

```

Upon calling Instant the first time for some C/C++ code, Instant compiles this code and stores the resulting files in a directory `ex4_cache`. The output from running the code the first time is:

```

--- Instant: compiling ---
Hello world! C function add is being called...
The sum of 3 and 4.5 is 7.5

```

Next time we ask Instant to call this code, it will check if the compiled files are available either in cache or locally, and further whether we need to rebuild these files based on the checksum of the source files and the arguments to the Instant function. This means that Instant will perform the compile step *only* if changes are made to the source code or arguments. More details about the different caching options can be found in Section 16.3.2.

The resulting module files can be found in a directory reflecting the name of the module, in this case `ex4_cache`:

```

ilmarw@multiboot:~/instant_doc/code$ cd ex4_cache/
ilmarw@multiboot:~/instant_doc/code/ex4_cache$ ls -g
total 224
drwxr-xr-x 4 ilmarw  4096 2009-05-18 16:52 build
-rw-r--r-- 1 ilmarw   844 2009-05-18 16:52 compile.log
-rw-r--r-- 1 ilmarw   183 2009-05-18 16:52 ex4_cache-0.0.0.egg-info
-rw-r--r-- 1 ilmarw    40 2009-05-18 16:52 ex4_cache.checksum
-rw-r--r-- 1 ilmarw   402 2009-05-18 16:53 ex4_cache.i
-rw-r--r-- 1 ilmarw  1866 2009-05-18 16:52 ex4_cache.py
-rw-r--r-- 1 ilmarw  2669 2009-05-18 16:52 ex4_cache.pyc
-rwxr-xr-x 1 ilmarw 82066 2009-05-18 16:52 _ex4_cache.so
-rw-r--r-- 1 ilmarw 94700 2009-05-18 16:52 ex4_cache_wrap.cxx
-rw-r--r-- 1 ilmarw   23 2009-05-18 16:53 __init__.py
-rw-r--r-- 1 ilmarw   448 2009-05-18 16:53 setup.py

```

When building a new module, Instant creates a new directory with a number of files. The first file it generates is the SWIG interface file, named `ex4_cache.i` in this example. Then the Distutils file `setup.py` is generated based and executed. During execution, `setup.py` first runs SWIG in the interface file, producing `ex4_cache_wrap.cxx` and `ex4_cache.py`. The first file is then compiled into a shared library `_ex4_cache.so` (note the leading underscore). A file `ex4_cache-0.0.0.egg-info` and a directory `build` will also be present as a

result of these steps. The output from executing the `Distutils` file is stored in the file `compile.log`. Finally, a checksum file named `ex4_cache.checksum` is generated, containing a checksum based on the files present in the directory. The final step consists of moving the whole directory from its temporary location to either `cache` or a user-specified directory. The `__init__.py` imports the module `ex4_cache`.

The script `instant-clean` removes compiled modules from the Instant cache, located in the directory `.instant` in the home directory of the user running it. In addition, all Instant modules located in the temporary directory where they were first generated and compiled. It does not clean modules located elsewhere.

The script `instant-showcache` allow you to see the modules currently located in the Instant cache:

```
Found 1 modules in Instant cache:
test_cache
Found 1 lock files in Instant cache:
test_cache.lock
```

Arguments to this script will output the files matching the specified pattern, for example will `instant-showcache 'test*.i'` show the content of the SWIG interface file for any module beginning with the letters `test`.

16.3.1 Arrays and Typemaps

Instant has support for converting NumPy arrays to C arrays and vice versa. For arrays with up to three dimensions, the SWIG interface file from NumPy is used, with a few modifications. When installing Instant, this file is included as well. `arrays` should be a list, each entry containing information about a specific array. This entry should contain a list with strings, so the `arrays` argument is a nested list.

Each array (i.e. each element in `arrays`) is a list containing the names of the variables describing that array in the C code. For a 1-dimensional array, this means the names of the variables containing the length of the array (an `int`), and the array pointer (can have several types, but the default is `double`). For 2-dimensional arrays we need three strings, two for the length in each dimension, and the third for the array pointer. For 3-dimensional arrays, there will be three variables first. This example should make things clearer

```
arrays = [['len', 'a'],
          ['len_bx', 'len_by', 'b'],
          ['len_cx', 'len_cy', 'len_cz', 'c']]
```

These variables names specified reflect the variable names in the C function signature. It is important that the order of the variables in the signature is retained for each array, e.g. one cannot write:

```
c_code = """
double sum (int len_a, int len_bx, int len_by,
            double* a, double* b)
{
    ...
}
"""
```

The correct code would be:

```
c_code = """
double sum (int len_a, double* a,
            int len_bx,
            int len_by, double* b)
{
    ...
}
"""
```

The order of the arrays can be changed, as long as the arguments in the Python function are changed as well accordingly.

Data Types

Default, all arrays are assumed to be of type `double`, but several other types are supported. These are `float`, `short`, `int`, `long`, `long long`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`. The type can be specified by adding an additional element to the list describing the array, e.g.

```
arrays = [['len', 'a', 'long']]
```

It is important that there is correspondance between the type of the NumPy array and the type in the signature of the C function. For arrays that are changed in-place, the types have to match exactly. For arrays that are input or output (see next section), one has to make sure that the implicit casting is done to a type with higher accuracy. For input arrays, the C type must be of higher (or the same) accuracy than the NumPy array, while for output arrays the NumPy array type must be of higher (or the same) accuracy than the C array. The NumPy type `float32` corresponds to the C type `float`, while `float64` corresponds to `double`. The NumPy type `float` is the same as `float64`. For integer arrays, the mapping between NumPy types and C types depends on your system. Using `long` as the C type will work in most cases.

Input/Output Arrays

All arrays are assumed to be both input and output arrays, i.e. any changes to arrays in the C code result in the NumPy array being changed in-place. For performance purposes, this is desirable, as we avoid unnecessary copying of data. The NumPy SWIG interface file has support for both input and output arrays in addition to changing arrays in-place. Input arrays do not need to be NumPy arrays, but can be any type of sequence, e.g. lists and tuples. The default behaviour of the NumPy SWIG interface file is to create new objects for sequences that are not NumPy arrays, while using mere pointers to the data of NumPy arrays. Instant deviates from this behaviour by taking copies of all input data, allowing for the modification of the array in the C code, as might be necessary for certain applications, while retaining the array as seen from the Python code. An array is marked as input only by adding the additional element 'in' to the list describing the array:

```
arrays = [['len', 'a', 'in']]
```

It is also possible to create output arrays in the C code. Instead of creating an array in the Python code and sending it as an in-place array to the C code, the array is created by the wrapper code and returned. If there are multiple output arrays or the C function has a return argument, the wrapper function returns a tuple with the different arguments. This approach is more Python-like than changing arrays in-place.

We only need to specify the length of the array when calling the wrapper function. The limitation is that only 1-dimensional arrays are supported, which means that we need to set the shape of the array manually after calling the wrapper function. In the C code all arrays are treated as 1-dimensional, so this does not affect the C code. An array is marked as input only by adding the additional element 'out' to the list describing the array. The following code shows an example where we calculate matrix-vector multiplication $x = Ab$. The matrix A is marked as input, the vector b as in-place, and the vector x as output. The example is only meant for illustrating the use of the different array options, and can be found in the file `$examples/ex5.py`. We verify that the result is correct by using the dot product from NumPy:

```
from instant import inline_with_numpy
from numpy import arange, dot

c_code = '''
void dot_c(int Am, int An, long* A, int bn, int* b,
          int xn, double* x)
{
    for (int i=0; i<Am; i++)
```

```

{
    x[i] = 0;
    for (int j=0; j<An; j++)
    {
        x[i] += A[i*Am + j]*b[j];
    }
}
'''
dot_c = \
    inline_with_numpy(c_code,
                      arrays = [['Am', 'An', 'A', 'in', 'long'],
                                ['bn', 'b', 'int'],
                                ['xn', 'x', 'out']])

a = arange(9)
a.shape = (3, 3)
b = arange(3)

c1 = dot_c(a, b, a.shape[1])
c2 = dot(a, b)
print c1
print c2

```

Multi-dimensional Arrays

If one needs to work with arrays that are more than 3-dimensional, this is possible. However, the typemaps used for this employ less error checking, and can only be used for the C type `double`. The list describing the array should contain the variable name for holding the number of dimensions, the variable name for an integer arrays holding the size in each dimension, the variable name for the array, and the argument `'multi'`, indicating that it has more than 3 dimensions. The `arrays` argument could for example be:

```

arrays = [['m', 'mp', 'ar1', 'multi'],
          ['n', 'np', 'ar2', 'multi']]

```

In this case, the C function signature should look like:

```

void sum (int m, int* mp, double* ar1, int n,
          int* np, double* ar2)

```

In the C code, all arrays are 1-dimensional. Indexing a 3-dimensional arrays becomes rather complicated because of striding. For instance, instead of writing

$u(i, j, k)$ we need to write $u[i*ny*nz + j*ny + k]$, where nx , ny , and nz are the lengths of the array in each direction. One way of achieving a simpler syntax is to use the `#define` macro in C:

```
#define index(u, i, j, k) u[(i)*nz*ny + (j)*ny + (k)]
```

which allows us to write `index(u, i, j, k)` instead.

16.3.2 *Module name, signature, and cache*

The Instant cache resides in the directory `.instant` in the directory of the user. It is possible to specify a different directory, but the `instant-clean` script will not remove these when executed. The three keyword arguments `modulename`, `signature`, and `cache_dir` are connected. If none of them are given, the default behaviour is to create a signature from the contents of the files and arguments to the `build_module` function, resulting in a name starting with `instant_``module_` followed by a long checksum. The resulting code is copied to Instant cache unless `cache_dir` is set to a specific directory. Note that changing the arguments or any of the files will result in a new directory in the Instant cache, as the checksum no longer is the same. Before compiling a module, Instant will always check if it is cached in both the Instant cache and in the current working directory.

If `modulename` is used, the directory with the resulting code is named accordingly, but not copied to the Instant cache. Instead, it is stored in the current working directory. Any changes to the argument or the source files will automatically result in a recompilation. The argument `cache_dir` is ignored.

When `signature` is given as argument, Instant uses this instead of calculating checksums. The resulting directory has the same name as the signature, provided the signature does not contain more than 100 characters containing only letters, numbers, or an underscore. If the signature contains any of these characters, the module name is generated based on the checksum of this string, resulting in a module name starting with `instant_module_` followed by the checksum. Because the user specifies the signature herself, changes in the arguments or source code will not cause a recompilation. The use of signatures is primarily intended for external software making use of Instant, e.g. SFC. Sometimes, the code output by this software might be different from the code used previously by Instant, without these changes affecting the result of running this code (e.g. comments are inserted to the code). By using signatures, the external program can decide when recompilation is necessary instead of leaving this to Instant. Unless otherwise specified, the modules is stored in the Instant cache.

It is not possible to specify both the module name and the signature. If both are given, Instant will issue an error.

In addition to the disk cache discussed so far, Instant also has a memory cache. All modules used during the life-time of a program are stored in memory for faster access. The memory cache is always checked before the disk cache.

16.3.3 *Locking*

Instant provides file locking functionality for cache modules. If multiple processes are working on the same module, race conditions could potentially occur where two or more processes believe the module is missing from the cache and try to write it simultaneously. To avoid race conditions, lock files were introduced. The lock files reside in the Instant cache, and locking is only enabled for modules that should be cached, i.e. where the module name is not given explicitly as argument to `build_module` or one of its wrapper functions. The first process to reach the stage where the module is copied from its temporary location to the Instant cache, will acquire a lock, and other processes cannot access this module while it is being copied.

16.4 Instant API

In this section we will describe the various Instant functions and their arguments visible to the user. The first ten functions are the core Instant functions, with `build_module` being the main one, while the next eight are wrapper functions around this function. Further, there are four more helper functions available, intended for using Instant with other applications.

16.4.1 *build_module*

This function is the most important one in Instant, and for most applications the only one that developers need to use, combined with the existing wrapper functions around this function. The return argument is the compiled module, hence it can be used directly in the calling code (rather than importing it as a Python module). It is also possible to import the module manually if compiled in the same directory as the calling code.

There are a number of keyword arguments, and we will explain them in detail here. Although one of the aims of Instant is to minimize the direct interaction with SWIG, some of the keywords require a good knowledge of SWIG in order to make sense. In this way, Instant can be used both by programmers new to the use of extension languages for Python, as well as by experienced SWIG programmers. The keywords arguments are as follows:

- `modulename`
 - Default: `None`

- Type: String
- Comment: The name you want for the module. If specified, the module will not be cached. If missing, a name will be constructed based on a checksum of the other arguments, and the module will be placed in the global cache. See Section 16.3.2 for more details.
- `source_directory`
 - Default: `'.'`
 - Type: String
 - Comment: The directory where user supplied files reside. The files given in `sources`, `wrap_headers`, and `local_headers` are expected to exist in this directory.
- `code`
 - Default: `''`
 - Type: String
 - Comment: The C or C++ code to be compiled and wrapped.
- `init_code`
 - Default: `''`
 - Type: String
 - Comment: Code that should be executed when the Instant module is imported. This code is inserted in the SWIG interface file, and is used for instance for calling `import_array()` used for the initialization of NumPy arrays.
- `additional_definitions`
 - Default: `''`
 - Type: String
 - Comment: Additional definitions (typically needed for inheritance) for interface file. These definitions should be given as triple-quoted strings in the case they span multiple lines, and are placed both in the initial block for C/C++ code (`%{ , %}`-block), and the main section of the interface file.
- `additional_declarations`
 - Default: `''`
 - Type: String

- Comment: Additional declarations (typically needed for inheritance) for interface file. These declarations should be given as triple-quoted strings in the case they span multiple lines, and are placed in the main section of the interface file.
- sources
 - Default: []
 - Type: List of strings
 - Comment: Source files to compile and link with the module. These files are compiled together with the SWIG-generated wrapper file into the final library file. Should reside in directory specified in `source_directory`.
- wrap_headers
 - Default: []
 - Type: List of strings
 - Comment: Local header files that should be wrapped by SWIG. The files specified will be included both in the initial block for C/C++ code (with a C directive) and in the main section of the interface file (with a SWIG directive). Should reside in directory specified in `source_directory`.
- local_headers
 - Default: []
 - Type: List of strings
 - Comment: Local header files required to compile the wrapped code. The files specified will be included in the initial block for C/C++ code (with a C directive). Should reside in directory specified in `source_directory`.
- system_headers
 - Default: []
 - Type: List of strings
 - Comment: System header files required to compile the wrapped code. The files specified will be included in the initial block for C/C++ code (with a C directive).
- include_dirs
 - Default: []

- Type: List of strings
- Comment: Directories to search for header files for building the extension module. Needs to be absolute path names.
- `library_dirs`
 - Default: []
 - Type: List of strings
 - Comment: Directories to search for libraries (-l) for building the extension module. Needs to be absolute paths.
- `libraries`
 - Default: []
 - Type: List of strings
 - Comment: Libraries needed by the Instant module. The libraries will be linked in from the shared object file. The initial -l is added automatically.
- `swigargs`
 - Default: ['-c++', '-fcompact', '-O', '-I.', '-small']
 - Type: List of strings
 - Comment: Arguments to swig, e.g. ['-lpointers.i'] to include the SWIG pointers.i library.
- `swig_include_dirs`
 - Default: []
 - Type: List of strings
 - Comment: Directories to include in the 'swig' command.
- `cppargs`
 - Default: ['-O2']
 - Type: List of strings
 - Comment: Arguments to the C++ compiler, other than include directories, e.g. ['-Wall', '-fopenmp'].
- `lddargs`
 - Default: []
 - Type: List of strings

- Comment: Arguments to the linker, other than libraries and library directories, e.g. ['-E', '-U'].
- arrays
 - Default: []
 - Type: List of strings
 - Comment: A nested list describing the C arrays to be made from NumPy arrays. The SWIG interface for fil NumPy is used. For 1D arrays, the inner list should contain strings with the variable names for the length of the arrays and the array itself. 2D matrices should contain the names of the dimensions in the two directions as well as the name of the array, and 3D tensors should contain the names of the dimensions in the three directions in addition to the name of the array. If the NumPy array has more than four dimensions, the inner list should contain strings with variable names for the number of dimensions, the length in each dimension as a pointer, and the array itself, respectively. For more details, see section 16.3.1.
- generate_interface
 - Default: True
 - Type: Boolean
 - Comment: Indicate whether you want to generate the interface files.
- generate_setup
 - Default: True
 - Type: Boolean
 - Comment: Indicate if you want to generate the setup.py file.
- signature
 - Default: None
 - Type: String
 - Comment: A signature string to identify the form instead of the source code. See Section 16.3.2.
- cache_dir
 - Default: None
 - Type: String

- Comment: A directory to look for cached modules and place new ones. If missing, a default directory is used. Note that the module will not be cached if `modulename` is specified. The cache directory should not be used for anything else.

16.4.2 *inline*

The function `inline` creates a module given that the input is a valid C/C++ function. It is only possible to inline one C/C++ function each time. One mandatory argument, which is the C/C++ code to be compiled.

The default keyword arguments from `build_module` are used, with `c_code` as the C/C++ code given as argument to `inline`. These keyword argument can be overridden, however, by giving them as arguments to `inline`, with the obvious exception of `code`. The function tries to return the single C/C++ function to be compiled rather than the whole module, if it fails, the module is returned.

16.4.3 *inline_module*

The same as `inline`, but returns the whole module rather than a single function. Except for the C/C++ code being a mandatory argument, the exact same as `build_module`.

16.4.4 *inline_with_numpy*

The difference between this function and the `inline` function is that C-arrays can be used. This means that the necessary arguments (`init_code`, `system_headers`, and `include_dirs`) for converting NumPy arrays to C arrays are set by the function.

16.4.5 *inline_module_with_numpy*

The difference between this function and the `inline_module` function is that C-arrays can be used. This means that the necessary arguments (`init_code`, `system_headers`, and `include_dirs`) for converting NumPy arrays to C arrays are set by the function.

16.4.6 *import_module*

This function can be used to import cached modules from the current work directory or the Instant cache. It has one mandatory argument, `moduleid`, and one keyword argument `cache_dir`. If the latter is given, Instant searches the specified directory instead of the Instant cache, if this directory exists. If the

module is not found, `None` is returned. The `moduleid` arguments can be either the module name, a signature, or an object with a function signature.

Using the module name or signature, assuming the module `instant_ext` exists in the current working directory or the Instant cache, we import a module in the following way:

```
instant_ext = import_module('instant_ext')
```

Using an object as argument, assuming this object includes a function signature() and the module is located in the directory `/tmp`:

```
instant_ext = import_module(signature_object, '/tmp')
```

The imported module, if found, is also placed in the memory cache.

16.4.7 *header_and_libs_from_pkgconfig*

This function returns a list of include files, flags, libraries and library directories obtain from a `pkg-config`⁶ file. It takes any number of arguments, one string for every package name. It returns four or five arguments. Unless the keyword argument `returnLinkFlags` is given with the value `True`, it returns lists with the include directories, the compile flags, the libraries, and the library directories of the package names given as arguments. If `returnLinkFlags` is `True`, the link flags are returned as a fifth list. Let's look at an example:

```
inc_dirs, comp_flags, libs, lib_dirs, link_flags = \  
header_and_libs_from_pkgconfig('ufc-1', 'libxml-2.0', 'numpy-1', \  
                               returnLinkFlags=True)
```

This makes it a easy to write C code that makes use of a package providing a `pkg-config` file, as we can use the returned lists for compiling and linking our module correctly.

16.4.8 *get_status_output*

This function provides a platform-independent way of running processes in the terminal and extracting the output using the Python module `subprocess`⁷. The one mandatory argument is the command we want to run. Further, there are three keyword arguments. The first is `input`, which should be a string containing input to the process once it is running. The other two are `cwd` and `env`. We refer to the documentation of `subprocess` for a more detailed description of these, but in short the first is the directory in which the process should be executed, while the second is used for setting the necessary environment variables.

⁶<http://pkg-config.freedesktop.org/wiki>

⁷<http://docs.python.org/library/subprocess.html>

16.4.9 get_swig_version

Returns the SWIG version installed on the system as a string, for instance '1.3.36'. Accepts no arguments.

16.4.10 check_swig_version

Takes a single argument, which should be a string on the same format as the output of `get_swig_version`. Returns `True` if the version of the installed SWIG is equal or greater than the version passed to the function. It also has one keyword argument, `same`. If it is `True`, the function returns `True` if and only if the two versions are the same.

SyFi: Symbolic Construction of Finite Element Basis Functions

By Martin S. Alnæs and Kent-Andre Mardal

Chapter ref: **[alnes-3]**

SyFi is a C++ library for definition of finite elements based on symbolic computations. By solving linear systems of equations symbolically, symbolic expressions for the basis functions of a finite element can be obtained. SyFi contains a collection of such elements.

The SyFi Form Compiler, SFC, is a Python module for generation of finite element code based on symbolic computations. Using equations in UFL format as input and basis functions from SyFi, SFC can generate C++ code which implements the UFC interface for computation of the discretized element tensors. SFC supports generating code based on quadrature or using symbolic integration prior to code generation to produce highly optimized code.

UFC: A Finite Element Code Generation Interface

By Martin S. Alnæs, Anders Logg and Kent-Andre Mardal

Chapter ref: [**alnes-2**]

When combining handwritten libraries with automatically generated code like we do in FEniCS, it is important to have clear boundaries between the two. This is best done by having the generated code implement a fixed interface, such that the library and generated code can be as independent as possible. Such an interface is specified in the project Unified Form-assembly Code (UFC) for finite elements and discrete variational forms. This interface consists of a small set of abstract classes in a single header file, which is well documented. The details of the UFC interface should rarely be visible to the end-user, but can be important for developers and technical users to understand how FEniCS projects fit together. In this chapter we discuss the main design ideas behind the UFC interface, including current limitations and possible future improvements.

UFL: A Finite Element Form Language

By Martin Sandve Alnæs

Chapter ref: **[alnes-1]**

► Editor note: *Sort out what to do with all UFL specific macros and bold math fonts.*

The Unified Form Language – UFL [?, ?] – is a domain specific language for the declaration of finite element discretizations of variational forms and functionals. More precisely, the language defines a flexible user interface for defining finite element spaces and expressions for weak forms in a notation close to mathematical notation.

The FEniCS project [?, ?, ?] provides a framework for building applications for solving partial differential equations (PDEs). UFL is one of the core components of this framework. It defines the language you *express* your PDEs in. It is the input language and front-end of the form compilers FFC [?, ?, ?, ?, ?, ?] and SFC [?, ?]. The UFL implementation provides algorithms that the form compilers can use to simplify the compilation process. The output from these form compilers is UFC [?, ?, ?] conforming C++ [?] code. This code can be used with the C++ library DOLFIN¹ [?, ?, ?] to efficiently assemble linear systems and compute solution to PDEs.

The combination of domain specific languages and symbolic computing with finite element methods has been pursued from other angles in several other projects. Sundance [?, ?, ?] implements a symbolic engine directly in C++ to define variational forms, and has support for automatic differentiation. The Life [?, ?] project uses a domain specific language embedded in C++, based on expression template techniques to specify variational forms. SfePy [?] uses SymPy

¹Note that in PyDOLFIN, some parts of UFL is wrapped to blend in with other software components and make the compilation process hidden from the user. This is not discussed here.

as a symbolic engine, extending it with finite element methods. GetDP [?, ?] is another project using a domain specific language for variational forms. The Mathematica package AceGen [?, ?] uses the symbolic capabilities of Mathematica to generate efficient code for finite element methods. All these packages have in common a focus on high level descriptions of partial differential equations to achieve higher human efficiency in the development of simulation software.

UFL almost resembles a library for symbolic computing, but its scope, goals and priorities are different from generic symbolic computing projects such as GiNaC [?, ?], swiginac [?] and SymPy [?]. Intended as a domain specific language and form compiler frontend, UFL is not suitable for large scale symbolic computing.

This chapter is intended both for the FEniCS user who wants to learn how to express her equations, and for other FEniCS developers and technical users who wants to know how UFL works on the inside. Therefore, the sections of this chapter are organized with an increasing amount of technical details. Sections 19.1-19.5 give an overview of the language as seen by the end-user and is intended for all audiences. Sections 19.6-19.9 explain the design of the implementation and dive into some implementation details. Many details of the language has to be omitted in a text such as this, and we refer to the UFL manual [?] for a more thorough description. Note that this chapter refers to UFL version 0.3, and both the user interface and the implementation may change in future versions.

Starting with a brief overview, we mention the main design goals for UFL and show an example implementation of a non-trivial PDE in Section 19.1. Next we will look at how to define finite element spaces in Section 19.2, followed by the overall structure of forms and their declaration in Section 19.3. The main part of the language is concerned with defining expressions from a set of data types and operators, which are discussed in Section 19.4. Operators applying to entire forms is the topic of Section 19.5.

The technical part of the chapter begins with Section 19.6 which discusses the representation of expressions. Building on the notation and data structures defined there, how to compute derivatives is discussed in Section 19.7. Some central internal algorithms and key issues in their implementation are discussed in Section 19.8. Implementation details, some of which are specific to the programming language Python [?], is the topic of Section 19.9. Finally, Section 19.10 discusses future prospects of the UFL project.

19.1 Overview

19.1.1 *Design goals*

UFL is a unification, refinement and reimplementaion of the form languages used in previous versions of FFC and SFC. The development of this language

has been motivated by several factors, the most important being:

- A richer form language, especially for expressing nonlinear PDEs.
- Automatic differentiation of expressions and forms.
- Improving the performance of the form compiler technology to handle more complicated equations efficiently.

UFL fulfils all these requirements, and by this it represents a major step forward in the capabilities of the FEniCS project.

Tensor algebra and index notation support is modeled after the FFC form language and generalized further. Several nonlinear operators and functions which only SFC supported before have been included in the language. Differentiation of expressions and forms has become an integrated part of the language, and is much easier to use than the way these features were implemented in SFC before. In summary, UFL combines the best of FFC and SFC in one unified form language and adds additional capabilities.

The efficiency of code generated by the new generation of form compilers based on UFL has been verified to match previous form compiler benchmarks [?, ?]. The form compilation process is now fast enough to blend into the regular application build process. Complicated forms that previously required too much memory to compile, or took tens of minutes or even hours to compile, now compile in seconds with both SFC and FFC.

19.1.2 *Motivational example*

One major motivating example during the initial development of UFL has been the equations for elasticity with large deformations. In particular, models of biological tissue use complicated hyperelastic constitutive laws with anisotropies and strong nonlinearities. To implement these equations with FEniCS, all three design goals listed above had to be addressed. Below, one version of the hyperelasticity equations and their corresponding UFL implementation is shown. Keep in mind that this is only intended as an illustration of the close correspondence between the form language and the natural formulation of the equations. The meaning of equations is not necessary for the reader to understand. Note that many other examples are distributed together with UFL.

In the formulation of the hyperelasticity equations presented here, the unknown function is the displacement vector field \mathbf{u} . The material coefficients c_1 and c_2 are scalar constants. The second Piola-Kirchoff stress tensor \mathbf{S} is computed from the strain energy function $W(\mathbf{C})$. W defines the constitutive law, here a simple Mooney-Rivlin law. The equations relating the displacement and

stresses read:

$$\begin{aligned}
 \mathbf{F} &= \mathbf{I} + (\nabla \mathbf{u})^T, \\
 \mathbf{C} &= \mathbf{F}^T \mathbf{F}, \\
 I_C &= \text{tr}(\mathbf{C}), \\
 II_C &= \frac{1}{2}(\text{tr}(\mathbf{C})^2 - \text{tr}(\mathbf{C}\mathbf{C})), \\
 W &= c_1(I_C - 3) + c_2(II_C - 3), \\
 \mathbf{S} &= 2 \frac{\partial W}{\partial \mathbf{C}}, \\
 \mathbf{P} &= \mathbf{F}\mathbf{S}.
 \end{aligned} \tag{19.1}$$

Approximating the displacement field as $\mathbf{u} = \sum_k u_k \phi_k^1$, the weak forms of the equations are as follows (ignoring boundary conditions):

$$L(\phi^0; \mathbf{u}, c_1, c_2) = \int_{\Omega} \mathbf{P} : (\nabla \phi^0)^T dx, \tag{19.2}$$

$$a(\phi^0, \phi_k^1; \mathbf{u}, c_1, c_2) = \frac{\partial L}{\partial u_k}. \tag{19.3}$$

Figure 19.1.2 shows an implementation of these equations in UFL. Notice the close relation between the mathematical notation and the UFL source code. In particular, note the automated differentiation of both the constitutive law and the residual equation. This means a new material law can be implemented by simply changing W , the rest is automatic. In the following sections, the notation, definitions and operators used in this implementation are explained.

19.2 Defining finite element spaces

A polygonal cell is defined by a basic shape and a degree², and is declared

```
cell = Cell(shape, degree)
```

UFL defines a set of valid polygonal cell shapes: “interval”, “triangle”, “tetrahedron”, “quadrilateral”, and “hexahedron”. Linear cells of all basic shapes are predefined and can be used instead by writing

```
cell = tetrahedron
```

²Note that at the time of writing, the other components of FEniCS does not yet handle higher degree cells.


```

# Finite element spaces
cell = tetrahedron
element = VectorElement("CG", cell, 1)

# Form arguments
phi0 = TestFunction(element)
phi1 = TrialFunction(element)
u = Function(element)
c1 = Constant(cell)
c2 = Constant(cell)

# Deformation gradient  $F_{ij} = dx_i/dx_j$ 
I = Identity(cell.d)
F = I + grad(u).T

# Right Cauchy-Green strain tensor C with invariants
C = variable(F.T*F)
I_C = tr(C)
II_C = (I_C**2 - tr(C*C))/2

# Mooney-Rivlin constitutive law
W = c1*(I_C-3) + c2*(II_C-3)

# Second Piola-Kirchhoff stress tensor
S = 2*diff(W, C)

# Weak forms
L = inner(F*S, grad(phi0).T)*dx
a = derivative(L, u, phi1)

```

Figure 19.1: UFL implementation of hyperelasticity equations with a Mooney-Rivlin material law.

In the rest of this chapter, a variable name `cell` will be used where any cell is a valid argument, to make the examples dimension independent wherever possible.

UFL defines syntax for *declaring* finite element spaces, but does not know anything about the actual polynomial basis or degrees of freedom. The polynomial basis is selected implicitly by choosing among predefined basic element families and providing a polynomial degree, but UFL only assumes that there *exists* a basis with a fixed ordering for each finite element space V_h , i.e.

$$V_h = \text{span} \{ \phi_j \}_{j=1}^n. \quad (19.4)$$

Basic scalar elements can be combined to form vector elements or tensor elements, and elements can easily be combined in arbitrary mixed element hierarchies.

The set of predefined³ element family names in UFL includes “Lagrange” (short name “CG”), representing scalar Lagrange finite elements (continuous piecewise polynomial functions), “Discontinuous Lagrange” (short name “DG”), representing scalar discontinuous Lagrange finite elements (discontinuous piecewise polynomial functions), and a range of other families that can be found in the manual. Each family name has an associated short name for convenience. To print all valid families to screen from Python, call `show_elements()`.

The syntax for declaring elements is best explained with some examples.

```
cell = tetrahedron

P = FiniteElement("Lagrange", cell, 1)
V = VectorElement("Lagrange", cell, 2)
T = TensorElement("DG", cell, 0, symmetry=True)

TH = V + P
ME = MixedElement(T, V, P)
```

In the first line a polygonal cell is selected from the set of predefined linear cells. Then a scalar linear Lagrange element `P` is declared, as well as a quadratic vector Lagrange element `V`. Next a symmetric rank 2 tensor element `T` is defined, which is also piecewise constant on each cell. The code proceeds to declare a mixed element `TH`, which combines the quadratic vector element `V` and the linear scalar element `P`. This element is known as the Taylor-Hood element. Finally another mixed element with three sub elements is declared. Note that writing `T + V + P` would not result in a mixed element with three direct sub elements, but rather `MixedElement(MixedElement(T + V), P)`.

³Form compilers can register additional element families.

19.3 Defining forms

Consider Poisson's equation with two different boundary conditions on $\partial\Omega_0$ and $\partial\Omega_1$,

$$a(v, u; w) = \int_{\Omega} w \nabla u \cdot \nabla v \, dx, \quad (19.5)$$

$$L(v; f, g, h) = \int_{\Omega} f v \, dx + \int_{\partial\Omega_0} g^2 v \, ds + \int_{\partial\Omega_1} h v \, ds. \quad (19.6)$$

These forms can be expressed in UFL as

```
a = dot(grad(u), grad(v))*dx
L = f*v*dx + g**2*v*ds(0) + h*v*ds(1)
```

where multiplication by the measures dx , $ds(0)$ and $ds(1)$ represent the integrals $\int_{\Omega_0}(\cdot) \, dx$, $\int_{\partial\Omega_0}(\cdot) \, ds$, and $\int_{\partial\Omega_1}(\cdot) \, ds$ respectively.

Forms expressed in UFL are intended for finite element discretization followed by compilation to efficient code for computing the element tensor. Considering the above example, the bilinear form a with one coefficient function w is assumed to be evaluated at a later point with a range of basis functions and the coefficient function fixed, that is

$$V_h^1 = \text{span} \{ \phi_k^1 \}, \quad V_h^2 = \text{span} \{ \phi_k^2 \}, \quad V_h^3 = \text{span} \{ \phi_k^3 \}, \quad (19.7)$$

$$w = \sum_{k=1}^{|V_h^2|} w_k \phi_k^3, \quad \{w_k\} \text{ given}, \quad (19.8)$$

$$A_{ij} = a(\phi_i^1, \phi_j^2; w), \quad i = 1, \dots, |V_h^1|, \quad j = 1, \dots, |V_h^2|. \quad (19.9)$$

In general, UFL is designed to express forms of the following generalized form:

$$a(\phi^1, \dots, \phi^r; w^1, \dots, w^n) = \sum_{k=1}^{n_c} \int_{\Omega_k} I_k^c \, dx + \sum_{k=1}^{n_e} \int_{\partial\Omega_k} I_k^e \, ds + \sum_{k=1}^{n_i} \int_{\Gamma_k} I_k^i \, dS. \quad (19.10)$$

Most of this chapter deals with ways to define the integrand expressions I_k^c , I_k^e and I_k^i . The rest of the notation will be explained below.

The form arguments are divided in two groups, the basis functions ϕ^1, \dots, ϕ^r and the coefficient functions w^1, \dots, w^n . All $\{\phi^k\}$ and $\{w^k\}$ are functions in some discrete function space with a basis. Note that the actual basis functions $\{\phi_j^k\}$ and the coefficients $\{w_k\}$ are never known to UFL, but we assume that the ordering of the basis for each finite element space is fixed. A fixed ordering only matters when differentiating forms, explained in Section 19.7.

Each term of a valid form expression must be a scalar-valued expression integrated exactly once, and they must be linear in $\{\phi^k\}$. Any term may have

nonlinear dependencies on coefficient functions. A form with one or two basis function arguments ($r = 1, 2$) is called a linear or bilinear form respectively, ignoring its dependency on coefficient functions. These will be assembled to vectors and matrices when used in an application. A form depending only on coefficient functions ($r = 0$) is called a functional, since it will be assembled to a real number.

The entire domain is denoted Ω , the external boundary is denoted $\partial\Omega$, while the set of interior facets of the triangulation is denoted Γ . Sub domains are marked with a suffix, e.g., $\Omega_k \subset \Omega$. As mentioned above, integration is expressed by multiplication with a measure, and UFL defines the measures dx , ds and dS . In summary, there are three kinds of integrals with corresponding UFL representations

- $\int_{\Omega_k} (\cdot) dx \leftrightarrow (\cdot) * dx(k)$, called a *cell integral*,
- $\int_{\partial\Omega_k} (\cdot) ds \leftrightarrow (\cdot) * ds(k)$, called an *exterior facet integral*,
- $\int_{\Gamma_k} (\cdot) dS \leftrightarrow (\cdot) * dS(k)$, called an *interior facet integral*,

Defining a different quadrature order for each term in a form can be achieved by attaching meta data to measure objects, e.g.,

```
dx02 = dx(0, { "integration_order": 2 })
dx14 = dx(1, { "integration_order": 4 })
dx12 = dx(1, { "integration_order": 2 })
L = f*v*dx02 + g*v*dx14 + h*v*dx12
```

Meta data can also be used to override other form compiler specific options separately for each term. For more details on this feature see the manuals of UFL and the form compilers.

19.4 Defining expressions

Most of UFL deals with how to declare expressions such as the integrand expressions in Equation 19.10. The most basic expressions are terminal values, which do not depend on other expressions. Other expressions are called operators, which are discussed in sections 19.4.2-19.4.5.

Terminal value types in UFL include form arguments (which is the topic of Section 19.4.1), geometric quantities, and literal constants. Among the literal constants are scalar integer and floating point values, as well as the d by d identity matrix $\mathbf{I} = \text{Identity}(d)$. To get unit vectors, simply use rows or columns of the identity matrix, e.g., $e_0 = \mathbf{I}[0, :]$. Similarly, $\mathbf{I}[i, j]$ represents the Dirac delta function δ_{ij} (see Section 19.4.2 for details on index notation). Available geometric values are the spatial coordinates $\mathbf{x} \leftrightarrow \text{cell.x}$ and the facet normal $\mathbf{n} \leftrightarrow \text{cell.n}$. The geometric dimension is available as `cell.d`.

19.4.1 Form arguments

Basis functions and coefficient functions are represented by `BasisFunction` and `Function` respectively. The ordering of the arguments to a form is decided by the order in which the form arguments were declared in the UFL code. Each basis function argument represents any function in the basis of its finite element space

$$\phi^j \in \{\phi_k^j\}, \quad V_h^j = \text{span} \{\phi_k^j\}. \quad (19.11)$$

with the intention that the form is later evaluated for all ϕ_k such as in equation (19.9). Each coefficient function w represents a discrete function in some finite element space V_h ; it is usually a sum of basis functions $\phi_k \in V_h$ with coefficients w_k

$$w = \sum_{k=1}^{|V_h|} w_k \phi_k. \quad (19.12)$$

The exception is coefficient functions that can only be evaluated pointwise, which are declared with a finite element with family “Quadrature”. Basis functions are declared for an arbitrary element as in the following manner:

```
phi = BasisFunction(element)
v = TestFunction(element)
u = TrialFunction(element)
```

By using `TestFunction` and `TrialFunction` in declarations instead of `BasisFunction` you can ignore their relative ordering. The only time `BasisFunction` is needed is for forms of arity $r > 2$.

Coefficient functions are declared similarly for an arbitrary element, and shorthand notation exists for declaring piecewise constant functions:

```
w = Function(element)
c = Constant(cell)
v = VectorConstant(cell)
M = TensorConstant(cell)
```

If a form argument u in a mixed finite element space $V_h = V_h^0 \times V_h^1$ is desired, but the form is more easily expressed using sub functions $u_0 \in V_h^0$ and $u_1 \in V_h^1$, you can split the mixed function or basis function into its sub functions in a generic way using `split`:

```
V = V0 + V1
u = Function(V)
u0, u1 = split(u)
```

The `split` function can handle arbitrary mixed elements. Alternatively, a handy shorthand notation for argument declaration followed by `split` is

```
v0, v1 = TestFunctions(V)
u0, u1 = TrialFunctions(V)
f0, f1 = Functions(V)
```

19.4.2 Index notation

UFL allows working with tensor expressions of arbitrary rank, using both tensor algebra and index notation. A basic familiarity with tensor algebra and index notation is assumed. The focus here is on how index notation is expressed in UFL.

Assuming a standard orthonormal Euclidean basis $\langle \mathbf{e}_k \rangle_{k=1}^d$ for \mathbb{R}^d , a vector can be expressed with its scalar components in this basis. Tensors of rank two can be expressed using their scalar components in a dyadic basis $\{\mathbf{e}_i \otimes \mathbf{e}_j\}_{i,j=1}^d$. Arbitrary rank tensors can be expressed the same way, as illustrated here.

$$\mathbf{v} = \sum_{k=1}^d v_k \mathbf{e}_k, \quad (19.13)$$

$$\mathbf{A} = \sum_{i=1}^d \sum_{j=1}^d A_{ij} \mathbf{e}_i \otimes \mathbf{e}_j, \quad (19.14)$$

$$\mathbf{C} = \sum_{i=1}^d \sum_{j=1}^d \sum_k C_{ijk} \mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k. \quad (19.15)$$

Here, \mathbf{v} , \mathbf{A} and \mathbf{C} are rank 1, 2 and 3 tensors respectively. Indices are called *free* if they have no assigned value, such as i in v_i , and *fixed* if they have a fixed value such as 1 in v_1 . An expression with free indices represents any expression you can get by assigning fixed values to the indices. The expression A_{ij} is scalar valued, and represents any component (i, j) of the tensor \mathbf{A} in the Euclidean basis. When working on paper, it is easy to switch between tensor notation (\mathbf{A}) and index notation (A_{ij}) with the knowledge that the tensor and its components are different representations of the same physical quantity. In a programming language, we must express the operations mapping from tensor to scalar components and back explicitly. Mapping from a tensor to its components, for a rank 2 tensor defined as

$$A_{ij} = \mathbf{A} : (\mathbf{e}_i \otimes \mathbf{e}_j), \quad (19.16)$$

$$(19.17)$$

is accomplished using indexing with the notation $A[i, j]$. Defining a tensor A from component values A_{ij} is defined as

$$\mathbf{A} = A_{ij} \mathbf{e}_i \otimes \mathbf{e}_j, \quad (19.18)$$

and is accomplished using the function `as_vector(Aij, (i, j))`. To illustrate, consider the outer product of two vectors $\mathbf{A} = \mathbf{u} \otimes \mathbf{v} = u_i v_j \mathbf{e}_i \otimes \mathbf{e}_j$, and the corresponding scalar components A_{ij} . One way to implement this is

```
A = outer(u, v)
Aij = A[i, j]
```

Alternatively, the components of \mathbf{A} can be expressed directly using index notation, such as $A_{ij} = u_i v_j$. A_{ij} can then be mapped to \mathbf{A} in the following manner:

```
Aij = v[j]*u[i]
A = as_tensor(Aij, (i, j))
```

These two pairs of lines are mathematically equivalent, and the result of either pair is that the variable \mathbf{A} represents the tensor \mathbf{A} and the variable A_{ij} represents the tensor A_{ij} . Note that free indices have no ordering, so their order of appearance in the expression `v[j]*u[i]` is insignificant. Instead of `as_tensor`, the specialized functions `as_vector` and `as_matrix` can be used. Although a rank two tensor was used for the examples above, the mappings generalize to arbitrary rank tensors.

When indexing expressions, fixed indices can also be used such as in `A[0, 1]` which represents a single scalar component. Fixed indices can also be mixed with free indices such as in `A[0, i]`. In addition, slices can be used in place of an index. An example of using slices is `A[0, :]` which is a vector expression that represents row 0 of \mathbf{A} . To create new indices, you can either make a single one or make several at once:

```
i = Index()
j, k, l = indices(3)
```

A set of indices i, j, k, l and p, q, r, s are predefined, and these should suffice for most applications.

If your components are not represented as an expression with free indices, but as separate unrelated scalar expressions, you can build a tensor from them using `as_tensor` and its peers. As an example, let's define a 2D rotation matrix and rotate a vector expression by $\frac{\pi}{2}$:

```
th = pi/2
A = as_matrix([[ cos(th), -sin(th)],
               [ sin(th),  cos(th)]]])
u = A*v
```

When indices are repeated in a term, summation over those indices is implied in accordance with the Einstein convention. In particular, indices can be repeated when indexing a tensor of rank two or higher ($A[i, i]$), when differentiating an expression with a free index ($v[i].dx(i)$), or when multiplying two expressions with shared free indices ($u[i]*v[i]$).

$$A_{ii} \equiv \sum_i A_{ii}, \quad v_i u_i \equiv \sum_i v_i u_i, \quad v_{i,i} \equiv \sum_i v_{i,i}. \quad (19.19)$$

An expression $A_{ij} = A[i, j]$ is represented internally using the `Indexed` class. A_{ij} will reference `A`, keeping the representation of the original tensor expression `A` unchanged. Implicit summation is represented explicitly in the expression tree using the class `IndexSum`. Many algorithms become easier to implement with this explicit representation, since e.g. a `Product` instance can never implicitly represent a sum. More details on representation classes are found in Section 19.6.

19.4.3 Algebraic operators and functions

UFL defines a comprehensive set of operators that can be used for composing expressions. The elementary algebraic operators $+$, $-$, $*$, $/$ can be used between most UFL expressions with a few limitations. Division requires a scalar expression with no free indices in the denominator. The operands to a sum must have the same shape and set of free indices.

The multiplication operator $*$ is valid between two scalars, a scalar and any tensor, a matrix and a vector, and two matrices. Other products could have been defined, but for clarity we use tensor algebra operators and index notation for those rare cases. A product of two expressions with shared free indices implies summation over those indices, see Section 19.4.2 for more about index notation.

Three often used operators are `dot(a, b)`, `inner(a, b)`, and `outer(a, b)`. The dot product of two tensors of arbitrary rank is the sum over the last index of the first tensor and the first index of the second tensor. Some examples are

$$\mathbf{v} \cdot \mathbf{u} = v_i u_i, \quad (19.20)$$

$$\mathbf{A} \cdot \mathbf{u} = A_{ij} u_j \mathbf{e}_i, \quad (19.21)$$

$$\mathbf{A} \cdot \mathbf{B} = A_{ik} B_{kj} \mathbf{e}_i \mathbf{e}_j, \quad (19.22)$$

$$\mathbf{C} \cdot \mathbf{A} = C_{ijk} A_{kl} \mathbf{e}_i \mathbf{e}_j \mathbf{e}_l. \quad (19.23)$$

The inner product is the sum over all indices, for example

$$\mathbf{v} : \mathbf{u} = v_i u_i, \quad (19.24)$$

$$\mathbf{A} : \mathbf{B} = A_{ij} B_{ij}, \quad (19.25)$$

$$\mathbf{C} : \mathbf{D} = C_{ijkl} D_{ijkl}. \quad (19.26)$$

Some examples of the outer product are

$$\mathbf{v} \otimes \mathbf{u} = v_i u_j \mathbf{e}_i \mathbf{e}_j, \quad (19.27)$$

$$\mathbf{A} \otimes \mathbf{u} = A_{ij} u_k \mathbf{e}_i \mathbf{e}_j \mathbf{e}_k, \quad (19.28)$$

$$\mathbf{A} \otimes \mathbf{B} = A_{ij} B_{kl} \mathbf{e}_i \mathbf{e}_j \mathbf{e}_k \mathbf{e}_l \quad (19.29)$$

Other common tensor algebra operators are `cross(u, v)`, `transpose(A)` (or `A.T`), `tr(A)`, `det(A)`, `inv(A)`, `cofac(A)`, `dev(A)`, `skew(A)`, and `sym(A)`. Most of these tensor algebra operators expect tensors without free indices. The detailed definitions of these operators are found in the manual.

A set of common elementary functions operating on scalar expressions without free indices are included, in particular `abs(f)`, `pow(f, g)`, `sqrt(f)`, `exp(f)`, `ln(f)`, `sin(f)`, `cos(f)`, and `sign(f)`.

19.4.4 Differential operators

UFL implements derivatives w.r.t. three different kinds of variables. The most used kind is spatial derivatives. Expressions can also be differentiated w.r.t. arbitrary user defined variables. And the final kind of derivatives are derivatives of a form or functional w.r.t. the coefficients of a `Function`. Form derivatives are explained in Section 19.5.1.

Note that derivatives are not computed immediately when declared. A discussion of how derivatives are computed is found in Section 19.7.

Spatial derivatives

Basic spatial derivatives $\frac{\partial f}{\partial x_i}$ can be expressed in two equivalent ways:

$$\begin{aligned} \text{df} &= \text{Dx}(f, i) \\ \text{df} &= f.\text{dx}(i) \end{aligned}$$

Here, `df` represents the derivative of `f` in the spatial direction x_i . The index `i` can either be an integer, representing differentiation in one fixed spatial direction x_i , or an `Index`, representing differentiation in the direction of a free index. The notation `f.dx(i)` is intended to mirror the index notation $f_{,i}$, which is shorthand for $\frac{\partial f}{\partial x_i}$. Repeated indices imply summation, such that the divergence of a vector can be written $v_{i,i}$, or `v[i].dx(i)`.

Several common compound spatial derivative operators are defined, namely `div`, `grad`, `curl` and `rot` (`rot` is a synonym for `curl`). The definition of these operators in UFL follow from the vector of partial derivatives

$$\nabla \equiv \mathbf{e}_k \frac{\partial}{\partial x_k}, \quad (19.30)$$

and the definition of the dot product, outer product, and cross product. Hence,

$$\operatorname{div}(\mathcal{C}) \equiv \nabla \cdot \mathcal{C}, \quad (19.31)$$

$$\operatorname{grad}(\mathcal{C}) \equiv \nabla \otimes \mathcal{C}, \quad (19.32)$$

$$\operatorname{curl}(\mathbf{v}) \equiv \nabla \times \mathbf{v}. \quad (19.33)$$

Note that there are two common ways to define grad and div . This way of defining these operators correspond to writing the convection term from, e.g., the Navier-Stokes equations as

$$\mathbf{w} \cdot \nabla \mathbf{u} = (\mathbf{w} \cdot \nabla) \mathbf{u} = \mathbf{w} \cdot (\nabla \mathbf{u}) = w_i u_{j,i}, \quad (19.34)$$

which is expressed in UFL as

`dot(w, grad(u))`

Another illustrative example is the anisotropic diffusion term from, e.g., the bidomain equations, which reads

$$(\mathbf{A} \nabla u) \cdot \mathbf{v} = A_{ij} u_{,j} v_i, \quad (19.35)$$

and is expressed in UFL as

`dot(A*grad(u), v)`

In other words, the divergence sums over the *first* index of its operand, and the gradient *prepends* an axis to the tensor shape of its operand. The above definition of curl is only valid for 3D vector expressions. For 2D vector and scalar expressions the definitions are:

$$\operatorname{curl}(\mathbf{u}) \equiv u_{1,0} - u_{0,1}, \quad (19.36)$$

$$\operatorname{curl}(f) \equiv f_{,1} \mathbf{e}_0 - f_{,0} \mathbf{e}_1. \quad (19.37)$$

User defined variables

The second kind of differentiation variables are user-defined variables, which can represent arbitrary expressions. Automating derivatives w.r.t. arbitrary quantities is useful for several tasks, from differentiation of material laws to computing sensitivities. An arbitrary expression g can be assigned to a variable v . An expression f defined as a function of v can be differentiated f w.r.t. v :

$$v = g, \quad (19.38)$$

$$f = f(v), \quad (19.39)$$

$$h(v) = \frac{\partial f(v)}{\partial v}. \quad (19.40)$$

Setting $g = \sin(x_0)$ and $f = e^{v^2}$, gives $h = 2ve^{v^2} = 2\sin(x_0)e^{\sin^2(x_0)}$, which can be implemented as follows:

```
g = sin(cell.x[0])
v = variable(g)
f = exp(v**2)
h = diff(f, v)
```

Try running this code in a Python session and print the expressions. The result is

```
>>> print v
var0(sin((x)[0]))
>>> print h
d/d[var0(sin((x)[0]))] (exp((var0(sin((x)[0]))) ** 2))
```

Note that the variable has a label 0 (“var0”), and that h still represents the abstract derivative. Section 19.7 explains how derivatives are computed.

19.4.5 Other operators

A few operators are provided for the implementation of discontinuous Galerkin methods. The basic concept is restricting an expression to the positive or negative side of an interior facet, which is expressed simply as $v('+')$ or $v('-')$ respectively. On top of this, the operators `avg` and `jump` are implemented, defined as

$$\text{avg}(v) = \frac{1}{2}(v^+ + v^-), \quad (19.41)$$

$$\text{jump}(v) = v^+ - v^-. \quad (19.42)$$

These operators can only be used when integrating over the interior facets (`*dS`).

The only control flow construct included in UFL is conditional expressions. A conditional expression takes on one of two values depending on the result of a boolean logic expression. The syntax for this is

```
f = conditional(condition, true_value, false_value)
```

which is interpreted as

$$f = \begin{cases} t, & \text{if condition is true,} \\ f, & \text{otherwise.} \end{cases} \quad (19.43)$$

The condition can be one of

- `lt(a, b) ↔ (a < b)`
- `gt(a, b) ↔ (a > b)`
- `le(a, b) ↔ (a ≤ b)`
- `ge(a, b) ↔ (a ≥ b)`
- `eq(a, b) ↔ (a = b)`
- `ne(a, b) ↔ (a ≠ b)`

19.5 Form operators

Once you have defined some forms, there are several ways to compute related forms from them. While operators in the previous section are used to define expressions, the operators discussed in this section are applied to forms, producing new forms. Form operators can both make form definitions more compact and reduce the chances of bugs since changes in the original form will propagate to forms computed from it automatically. These form operators can be combined arbitrarily; given a semi-linear form only a few lines are needed to compute the action of the adjoint of the Jacobi. Since these computations are done prior to processing by the form compilers, there is no overhead at run-time.

19.5.1 Differentiating forms

The form operator `derivative` declares the derivative of a form w.r.t. coefficients of a discrete function (`Function`). This functionality can be used for example to linearize your nonlinear residual equation (linear form) automatically for use with the Newton-Raphson method. It can also be applied multiple times, which is useful to derive a linear system from a convex functional, in order to find the function that minimizes the functional. For non-trivial equations such expressions can be tedious to calculate by hand. Other areas in which this feature can be useful include optimal control and inverse methods, as well as sensitivity analysis.

In its simplest form, the declaration of the derivative of a form `L` w.r.t. the coefficients of a function `w` reads

```
a = derivative(L, w, u)
```

The form `a` depends on an additional basis function argument `u`, which must be in the same finite element space as the function `w`. If the last argument is omitted, a new basis function argument is created.

Let us step through an example of how to apply `derivative` twice to a functional to derive a linear system. In the following, V_h is a finite element space with some basis, w is a function in V_h , and f is a functional we want to minimize.

Derived from f is a linear form F , and a bilinear form J .

$$V_h = \text{span} \{ \phi_k \}, \quad (19.44)$$

$$w(x) = \sum_{k=1}^{|V_h|} w_k \phi_k(x), \quad (19.45)$$

$$f : V_h \rightarrow \mathbb{R}, \quad (19.46)$$

$$F(\phi_i; w) = \frac{\partial}{\partial w_i} f(w), \quad (19.47)$$

$$J(\phi_i, \phi_j; w) = \frac{\partial}{\partial w_j} F(\phi_i; w). \quad (19.48)$$

For a concrete functional $f(w) = \int_{\Omega} \frac{1}{2} w^2 dx$, we can implement this as

```
v = TestFunction(element)
u = TrialFunction(element)
w = Function(element)
f = 0.5 * w**2 * dx
F = derivative(f, w, v)
J = derivative(F, w, u)
```

This code declares two forms F and J . The linear form F represents the standard load vector $w*v*dx$ and the bilinear form J represents the mass matrix $u*v*dx$.

Derivatives can also be defined w.r.t. coefficients of a function in a mixed finite element space. Consider the Harmonic map equations derived from the functional

$$f(\mathbf{x}, \lambda) = \int_{\Omega} \nabla \mathbf{x} : \nabla \mathbf{x} + \lambda \mathbf{x} \cdot \mathbf{x} dx, \quad (19.49)$$

where \mathbf{x} is a function in a vector finite element space V_h^d and λ is a function in a scalar finite element space V_h . The linear and bilinear forms derived from the functional in Equation 19.49 have basis function arguments in the mixed space $V_h^d + V_h$. The implementation of these forms with automatic linearization reads

```
Vx = VectorElement("CG", triangle, 1)
Vy = FiniteElement("CG", triangle, 1)
u = Function(Vx + Vy)
x, y = split(u)
f = inner(grad(x), grad(x))*dx + y*dot(x,x)*dx
F = derivative(f, u)
J = derivative(F, u)
```

Note that the functional is expressed in terms of the subfunctions `x` and `y`, while the argument to `derivative` must be the single mixed function `u`. In this example the basis function arguments to `derivative` are omitted and thus provided automatically in the right function spaces.

Note that in computing derivatives of forms, we have assumed that

$$\frac{\partial}{\partial w_k} \int_{\Omega} I \, dx = \int_{\Omega} \frac{\partial}{\partial w_k} I \, dx, \quad (19.50)$$

or in particular that the domain Ω is independent of w . Furthermore, note that there is no restriction on the choice of element in this framework, in particular arbitrary mixed elements are supported.

19.5.2 Adjoint

Another form operator is the adjoint a^* of a bilinear form a , defined as $a^*(u, v) = a(v, u)$, which is similar to taking the transpose of the assembled sparse matrix. In UFL this is implemented simply by swapping the test and trial functions, and can be written:

```
a = inner(M*grad(u), grad(v))*dx
ad = adjoint(a)
```

which corresponds to

$$a(M; v, u) = \int_{\Omega} (\mathbf{M}\nabla\mathbf{u}) : \nabla\mathbf{v} \, dx = \int_{\Omega} M_{ik} u_{j,k} v_{j,i} \, dx, \quad (19.51)$$

$$a^*(M; v, u) = a(M; u, v) = \int_{\Omega} (\mathbf{M}\nabla\mathbf{v}) : \nabla\mathbf{u} \, dx. \quad (19.52)$$

This automatic transformation is particularly useful if we need the adjoint of nonsymmetric bilinear forms computed using `derivative`, since the explicit expressions for a are not at hand. Several of the form operators below are most useful when used in conjunction with `derivative`.

19.5.3 Replacing functions

Evaluating a form with new definitions of form arguments can be done by replacing terminal objects with other values. Lets say you have defined a form `L` that depends on some functions `f` and `g`. You can then specialize the form by replacing these functions with other functions or fixed values, such as

$$L(f, g; v) = \int_{\Omega} (f^2/(2g))v \, dx, \quad (19.53)$$

$$L_2(f, g; v) = L(g, 3; v) = \int_{\Omega} (g^2/6)v \, dx. \quad (19.54)$$

This feature is implemented with `replace`, as illustrated in this case:

```
L = f**2 / (2*g) * v * dx
L2 = replace(L, { f: g, g: 3})
L3 = g**2 / 6 * v * dx
```

Here L2 and L3 represents exactly the same form. Since they depend only on g , the code generated for these forms can be more efficient.

19.5.4 Action

Sparse matrix-vector multiplication is an important operation in PDE solver applications. In some cases the matrix is not needed explicitly, only the action of the matrix on a vector, the result of the matrix-vector multiplication. You can assemble the action of the matrix on a vector directly by defining a linear form for the action of a bilinear form on a function, simply writing $L = \text{action}(a, w)$ or $L = a*w$, with a any bilinear form and w being any `Function` defined on the same finite element as the trial function in a .

19.5.5 Splitting a system

If you prefer to write your PDEs with all terms on one side such as

$$a(v, u) - L(v) = 0, \quad (19.55)$$

you can declare forms with both linear and bilinear terms and split the equations afterwards:

```
pde = u*v*dx - f*v*dx
a, L = system(pde)
```

Here `system` is used to split the PDE into its bilinear and linear parts. Alternatively, `lhs` and `rhs` can be used to obtain the two parts separately.

19.5.6 Computing the sensitivity of a function

If you have found the solution u to Equation (19.55), and u depends on some constant scalar value c , you can compute the sensitivity of u w.r.t. changes in c . If u is represented by a coefficient vector x that is the solution to the algebraic linear system $Ax = b$, the coefficients of $\frac{\partial u}{\partial c}$ are $\frac{\partial x}{\partial c}$. Applying $\frac{\partial}{\partial c}$ to $Ax = b$ and using the chain rule, we can write

$$A \frac{\partial x}{\partial c} = \frac{\partial b}{\partial c} - \frac{\partial A}{\partial c} x, \quad (19.56)$$

and thus $\frac{\partial x}{\partial c}$ can be found by solving the same algebraic linear system used to compute x , only with a different right hand side. The linear form corresponding to the right hand side of Equation (19.56) can be written

```
u = Function(element)
sL = diff(L, c) - action(diff(a, c), u)
```

or you can use the equivalent form transformation

```
sL = sensitivity_rhs(a, u, L, c)
```

Note that the solution u must be represented by a `Function`, while u in $a(v, u)$ is represented by a `BasisFunction`.

19.6 Expression representation

19.6.1 The structure of an expression

Most of the UFL implementation is concerned with expressing, representing, and manipulating expressions. To explain and reason about expression representations and algorithms operating on them, we need an abstract notation for the structure of an expression. UFL expressions are representations of programs, and the notation should allow us to see this connection without the burden of implementation details.

The most basic UFL expressions are expressions with no dependencies on other expressions, called *terminals*. Other expressions are the result of applying some *operator* to one or more existing expressions. All expressions are immutable; once constructed an expression will never change. Manipulating an expression always results in a new expression being created.

Consider an arbitrary (non-terminal) expression z . This expression depends on a set of terminal values $\{t_i\}$, and is computed using a set of operators $\{f_i\}$. If each subexpression of z is labeled with an integer, an abstract program can be written to compute z by computing a sequence of subexpressions $\langle y_i \rangle_{i=1}^n$ and setting $z = y_n$. Algorithm 6 shows such a program.

Algorithm 6 Program to compute an expression z

```
for  $i = 1, \dots, m$ :
     $y_i = t_i =$  terminal expression
for  $i = m + 1, \dots, n$ :
     $y_i = f_i(\langle y_j \rangle_{j \in \mathcal{J}_i})$ 
 $z = y_n$ 
```



Figure 19.2: Expression class hierarchy.

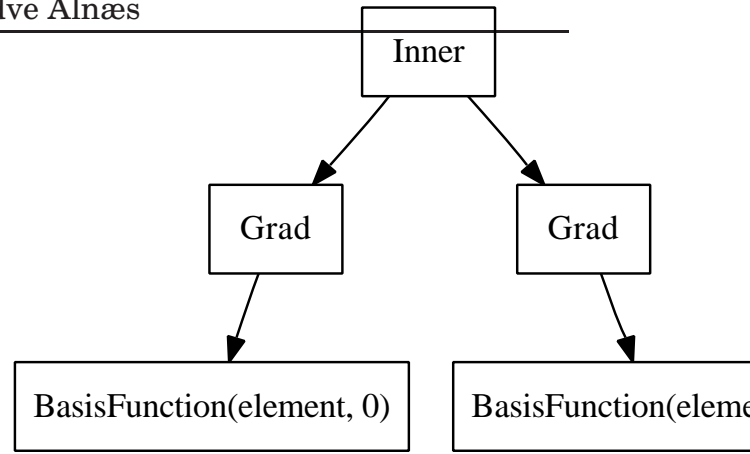


Figure 19.3: Expression tree for $\nabla_{\mathbf{v}} \mathbf{u}$.

Each terminal expression $y_i = t_i$ is a literal constant or input arguments to the program. A non-terminal subexpression y_i is the result of applying an operator f_i to a sequence of previously computed expressions $\langle y_j \rangle_{j \in \mathcal{I}_i}$, where \mathcal{I}_i is a set of expression labels. Note that the order in which subexpressions are computed can be arbitrarily chosen, except that we require $j < i \forall j \in \mathcal{I}_i$, such that all dependencies of a subexpression y_i has been computed before y_i . In particular, all terminals are numbered first in this algorithm for notational convenience only.

The program can be represented as a graph, where each expression y_i corresponds to a graph vertex and each direct dependency between two expressions is a graph edge. More formally,

$$G = (V, E), \tag{19.57}$$

$$V = \langle v_i \rangle_{i=1}^n = \langle y_i \rangle_{i=1}^n, \tag{19.58}$$

$$E = \{e_i\} = \bigcup_{i=1}^n \{(i, j) \forall j \in \mathcal{I}_i\}. \tag{19.59}$$

This graph is clearly directed, since dependencies have a direction. It is acyclic, since an expression can only be constructed from existing expressions and never be modified. Thus we can say that an UFL expression represents a program, and can be represented using a directed acyclic graph (DAG). There are two ways this DAG can be represented in UFL, a linked representation called the expression tree, and a linearized representation called the computational graph.

19.6.2 Tree representation

► Editor note: Redraw these figures in Inkscape.

An expression is usually represented as an expression tree. Each subexpression is represented by a tree node, which is the root of a tree of its own. The

leaves of the tree are terminal expressions, and operators have their operands as children. An expression tree for the stiffness term $\nabla \mathbf{u} : \nabla \mathbf{v}$ is illustrated in Figure 19.3. The terminals \mathbf{u} and \mathbf{v} have no children, and the term $\nabla \mathbf{u}$ is itself represented by a tree with two nodes. The names in this figure, `Grad`, `Inner` and `BasisFunction`, reflect the names of the classes used in UFL to represent the expression nodes. Taking the gradient of an expression with `grad(u)` gives an expression representation `Grad(u)`, and `inner(a, b)` gives an expression representation `Inner(a, b)`. In general, each expression node is an instance of some subclass of `Expr`. The class `Expr` is the superclass of a hierarchy containing all terminal types and operator types UFL supports. `Expr` has two direct subclasses, `Terminal` and `Operator`, as illustrated in Figure 19.2.

Each expression node represents a single vertex v_i in the DAG. Recall from Algorithm 6 that non-terminals are expressions $y_i = f_i(\langle y_j \rangle_{j \in \mathcal{I}_i})$. The operator f_i is represented by the class of the expression node, while the expression y_i is represented by the instance of this class. The edges of the DAG is not stored explicitly in the tree representation. However, from an expression node representing the vertex v_i , a tuple with the vertices $\langle y_j \rangle_{j \in \mathcal{I}_i}$ can be obtained by calling `yi.operands()`. These expression nodes represent the graph vertices that have edges pointing to them from y_i . Note that this generalizes to terminals where there are no outgoing edges and `t.operands()` returns an empty tuple.

19.6.3 Expression node properties

Any expression node e (an `Expr` instance) has certain generic properties, and the most important ones will be explained here. Above it was mentioned that `e.operands()` returns a tuple with the child nodes. Any expression node can be reconstructed with modified operands using `e.reconstruct(operands)`, where `operands` is a tuple of expression nodes. The invariant `e.reconstruct(e.operands) == e` should always hold. This function is required because expression nodes are immutable, they should never be modified. The immutable property ensures that expression nodes can be reused and shared between expressions without side effects in other parts of a program.

► *Editor note: Stick ugly text sticking out in margin.*

In Section 19.4.2 the tensor algebra and index notation capabilities of UFL was discussed. Expressions can be scalar or tensor-valued, with arbitrary rank and shape. Therefore, each expression node has a value `shape()`, which is a tuple of integers with the dimensions in each tensor axis. Scalar expressions have `shape()`. Another important property is the set of free indices in an expression, obtained as a tuple using `e.free_indices()`. Although the free indices have no ordering, they are represented with a tuple of `Index` instances for simplicity. Thus the ordering within the tuple carries no meaning.

UFL expressions are referentially transparent with some exceptions. Ref-

erential transparency means that a subexpression can be replaced by another representation of its value without changing the meaning of the expression. A key point here is that the value of an expression in this context includes the tensor shape and set of free indices. Another important point is that the derivative of a function $f(v)$ in a point, $f'(v)|_{v=g}$, depends on function values in the vicinity of $v = g$. The effect of this dependency is that operator types matter when differentiating, not only the current value of the differentiation variable. In particular, a `Variable` cannot be replaced by the expression it represents, because `diff` depends on the `Variable` instance and not the expression it has the value of. Similarly, replacing a `Function` with some value will change the meaning of an expression that contains derivatives w.r.t. function coefficients.

The following example illustrate this issue.

```
e = 0
v = variable(e)
f = sin(v)
g = diff(f, v)
```

Here `v` is a variable that takes on the value 0, but `sin(v)` cannot be simplified to 0 since the derivative of `f` then would be 0. The correct result here is `g = cos(v)`.

19.6.4 Linearized graph representation

A linearized representation of the DAG is useful for several internal algorithms, either to achieve a more convenient formulation of an algorithm or for improved performance. UFL includes tools to build a linearized representation of the DAG, the *computational graph*, from any expression tree. The computational graph $G = V, E$ is a data structure based on flat arrays, directly mirroring the definition of the graph in equations (19.57)-(19.59). This simple data structure makes some algorithms easier to implement or more efficient than the recursive tree representation. One array (Python list) `v` is used to store the vertices $\langle v_i \rangle_{i=1}^n$ of the DAG. For each vertex v_i an expression node y_i is stored to represent it. Thus the expression tree for each vertex is also directly available, since each expression node is the root of its own expression tree. The edges are stored in an array `E` with integer tuples (i, j) representing an edge from v_i to v_j , i.e. that v_j is an operand of v_i . The graph is built using a post-order traversal, which guarantees that the vertices are ordered such that $j < i \forall j \in \mathcal{J}_i$.

From the edges E , related arrays can be computed efficiently; in particular the vertex indices of dependencies of a vertex v_i in both directions are useful:

$$\begin{aligned} V_{out} &= \langle \mathcal{J}_i \rangle_{i=1}^n, \\ V_{in} &= \langle \{j | i \in \mathcal{J}_j\} \rangle_{i=1}^n \end{aligned} \tag{19.60}$$

These data structures can be easily constructed for any expression:

```
G = Graph(expression)
V, E = G
Vin = G.Vin()
Vout = G.Vout()
```

A nice property of the computational graph built by UFL is that no two vertices will represent the same identical expression. During graph building, subexpressions are inserted in a hash map (Python dict) to achieve this.

Free indices in expression nodes can complicate the interpretation of the linearized graph when implementing some algorithms. One solution to that can be to apply `expand_indices` before constructing the graph. Note however that free indices cannot be regained after expansion.

19.6.5 Partitioning

UFL is intended as a front-end for form compilers. Since the end goal is generation of code from expressions, some utilities are provided for the code generation process. In principle, correct code can be generated for an expression from its computational graph simply by iterating over the vertices and generating code for each operation separately, basically mirroring Algorithm 6. However, a good form compiler should be able to produce better code. UFL provides utilities for partitioning the computational graph into subgraphs (partitions) based on dependencies of subexpressions, which enables quadrature based form compilers to easily place subexpressions inside the right sets of loops. The function `partition` implements this feature. Each partition is represented by a simple array of vertex indices.

19.7 Computing derivatives

When a derivative expression is declared by the end-user of the form language, an expression node is constructed to represent it, but nothing is computed. The type of this expression node is a subclass of `Derivative`. Differential operators cannot be expressed natively in a language such as C++. Before code can be generated from the derivative expression, some kind of algorithm to evaluate derivatives must be applied. Computing exact derivatives is important, which rules out approximations by divided differences. Several alternative algorithms exist for computing exact derivatives. All relevant algorithms are based on the chain rule combined with differentiation rules for each expression node type. The main differences between the algorithms are in the extent of which subexpressions are reused, and in the way subexpressions are accumulated.

Below, the differences and similarities between some of the simplest algorithms are discussed. After the algorithm currently implemented in UFL has been explained, extensions to tensor and index notation and higher order derivatives are discussed. Finally, the section is closed with some remarks about the differentiation rules for terminal expressions.

19.7.1 *Relations to form compiler approaches*

Before discussing the choice of algorithm for computing derivatives, let us consider the context in which the results will be used. Although UFL does not generate code, some form compiler issues are relevant to this context.

Mixing derivative computation into the code generation strategy of each form compiler would lead to a significant duplication of implementation effort. To separate concerns and keep the code manageable, differentiation is implemented as part of UFL in such a way that the form compilers are independent of the chosen differentiation strategy. Before expressions are interpreted by a form compiler, differential operators should be evaluated such that the only operators left are non-differential operators⁴. Therefore, it is advantageous to use the same representation for the evaluated derivative expressions and other expressions.

The properties of each differentiation algorithm is strongly related to the structure of the expression representation. However, UFL has no control over the final expression representation used by the form compilers. The main difference between the current form compilers is the way in which expressions are integrated. For large classes of equations, symbolic integration or a specialized tensor representation have proven highly efficient ways to evaluate element tensors [?, ?, ?]. However, when applied to more complex equations, the run-time performance of both these approaches is beaten by code generated with quadrature loops [?, ?]. To apply symbolic differentiation, polynomials are expanded which destroys the structure of the expressions, gives potential exponential growth of expression sizes, and hides opportunities for subexpression reuse. Similarly, the tensor representation demands a canonical representation of the integral expressions.

In summary, both current non-quadrature form compiler approaches change the structure of the expressions they get from UFL. This change makes the interaction between the differentiation algorithm and the form compiler approach hard to control. However, this will only become a problem for complex equations, in which case quadrature loop based code is more suitable. Code generation using quadrature loops can more easily mirror the inherent structure of UFL expressions.

⁴An exception is made for spatial derivatives of terminals which are unknown to UFL because they are provided by the form compilers.

19.7.2 Approaches to computing derivatives

Algorithms for computing derivatives are designed with different end goals in mind. Symbolic Differentiation (SD) takes as input a single symbolic expression and produces a new symbolic expression for the derivative of the input. Automatic Differentiation (AD) takes as input a program to compute a function and produces a new program to compute the derivative of the function. Several variants of AD algorithms exist, the two most common being Forward Mode AD and Reverse Mode AD [?]. More advanced algorithms exist, and is an active research topic. is a symbolic expression, represented by an expression tree. But the expression tree is a directed acyclic graph that represents a program to evaluate said expression. Thus it seems the line between SD and AD becomes less distinct in this context.

Naively applied, SD can result in huge expressions, which can both require a lot of memory during the computation and be highly inefficient if written to code directly. However, some illustrations of the inefficiency of symbolic differentiation, such as in [?], are based on computing closed form expressions of derivatives in some stand-alone computer algebra system (CAS). Copying the resulting large expressions directly into a computer code can lead to very inefficient code. The compiler may not be able to detect common subexpressions, in particular if simplification and rewriting rules in the CAS has changed the structure of subexpressions with a potential for reuse.

In general, AD is capable of handling algorithms that SD can not. A tool for applying AD to a generic source code must handle many complications such as subroutines, global variables, arbitrary loops and branches [?, ?, ?]. Since the support for program flow constructs in UFL is very limited, the AD implementation in UFL will not run into such complications. In Section 19.7.3 the similarity between SD and forward mode AD in the context of UFL is explained in more detail.

19.7.3 Forward mode Automatic Differentiation

Recall Algorithm 6, which represents a program for computing an expression z from a set of terminal values $\{t_i\}$ and a set of elementary operations $\{f_i\}$. Assume for a moment that there are no differential operators among $\{f_i\}$. The algorithm can then be extended to compute the derivative $\frac{dz}{dv}$, where v represents a differentiation variable of any kind. This extension gives Algorithm 7.

This way of extending a program to simultaneously compute the expression z and its derivative $\frac{dz}{dv}$ is called forward mode automatic differentiation (AD). By renaming y_i and $\frac{dy_i}{dv}$ to a new sequence of values $\langle \hat{y}_j \rangle_{j=1}^{\hat{n}}$, Algorithm 7 can be rewritten as shown in Algorithm 8, which is isomorphic to Algorithm 6 (they have exactly the same structure).

Since the program in Algorithm 6 can be represented as a DAG, and Algo-

Algorithm 7 Forward mode AD on Algorithm 6

for $i = 1, \dots, m$:
 $y_i = t_i$
 $\frac{dy_i}{dv} = \frac{dt_i}{dv}$
for $i = m + 1, \dots, n$:
 $y_i = f_i(\langle y_j \rangle_{j \in \mathcal{I}_i})$
 $\frac{dy_i}{dv} = \sum_{k \in \mathcal{I}_i} \frac{\partial f_i}{\partial y_k} \frac{dy_k}{dv}$
 $z = y_n$
 $\frac{dz}{dv} = \frac{dy_n}{dv}$

Algorithm 8 Program to compute $\frac{dz}{dv}$ produced by forward mode AD

for $i = 1, \dots, \hat{m}$:
 $\hat{y}_i = \hat{t}_i$
for $i = \hat{m} + 1, \dots, \hat{n}$:
 $\hat{y}_i = \hat{f}_i(\langle \hat{y}_j \rangle_{j \in \hat{\mathcal{I}}_i})$
 $\frac{dz}{dv} = \hat{y}_{\hat{n}}$

Algorithm 8 is isomorphic to Algorithm 6, the program in Algorithm 8 can also be represented as a DAG. Thus a program to compute $\frac{dz}{dv}$ can be represented by an expression tree built from terminal values and non-differential operators.

The currently implemented algorithm for computing derivatives in UFL follows forward mode AD closely. Since the result is a new expression tree, the algorithm can also be called symbolic differentiation. In this context, the differences between the two are implementation details. To ensure that we can reuse expressions properly, simplification rules in UFL avoids modifying the operands of an operator. Naturally repeated patterns in the expression can therefore be detected easily by the form compilers. Efficient common subexpression elimination can then be implemented by placing subexpressions in a hash map. However, there are simplifications such as $0 * f \rightarrow 0$ and $1 * f \rightarrow f$ which simplify the result of the differentiation algorithm automatically as it is being constructed. These simplifications are crucial for the memory use during derivative computations, and the performance of the resulting program.

19.7.4 Extensions to tensors and indexed expressions

So far we have not considered derivatives of non-scalar expression and expressions with free indices. This issue does not affect the overall algorithms, but it does affect the local derivative rules for each expression type.

Consider the expression $\text{diff}(A, B)$ with A and B matrix expressions. The meaning of derivatives of tensors w.r.t. to tensors is easily defined via index

notation, which is heavily used within the differentiation rules:

$$\frac{d\mathbf{A}}{d\mathbf{B}} = \frac{dA_{ij}}{dB_{kl}} \mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k \otimes \mathbf{e}_l \quad (19.61)$$

Derivatives of subexpressions are frequently evaluated to literal constants. For indexed expressions, it is important that free indices are propagated correctly with the derivatives. Therefore, differentiated expressions will some times include literal constants annotated with free indices.

There is one rare and tricky corner case when an index sum binds an index i such as in $(v_i v_i)$ and the derivative w.r.t. x_i is attempted. The simplest example of this is the expression $(v_i v_i)_{,j}$, which has one free index j . If j is replaced by i , the expression can still be well defined, but you would never write $(v_i v_i)_{,i}$ manually. If the expression in the parenthesis is defined in a variable $e = v[i]*v[i]$, the expression $e.dx(i)$ looks innocent. However, this will cause problems as derivatives (including the index i) are propagated up to terminals. If this case is encountered it will be detected and an error message will be triggered. To avoid it, simply use different index instances. In the future, this case may be handled by relabeling indices to change this expression into $(v_j v_j)_{,i} u_i$.

19.7.5 Higher order derivatives

A simple forward mode AD implementation such as Algorithm 7 only considers one differentiation variable. Higher order or nested differential operators must also be supported, with any combination of differentiation variables. A simple example illustrating such an expression can be

$$a = \frac{d}{dx} \left(\frac{d}{dx} f(x) + 2 \frac{d}{dy} g(x, y) \right). \quad (19.62)$$

Considerations for implementations of nested derivatives in a functional⁵ framework have been explored in several papers [?, ?, ?].

In the current UFL implementation this is solved in a different fashion. Considering Equation (19.62), the approach is simply to compute the innermost derivatives $\frac{d}{dx} f(x)$ and $\frac{d}{dy} g(x, y)$ first, and then computing the outer derivatives. This approach is possible because the result of a derivative computation is represented as an expression tree just as any other expression. Mainly this approach was chosen because it is simple to implement and easy to verify. Whether other approaches are faster has not been investigated. Furthermore, alternative AD algorithms such as reverse mode can be experimented with in the future without concern for nested derivatives in the first implementations.

An outer controller function `apply_ad` handles the application of a single variable AD routine to an expression with possibly nested derivatives. The AD

⁵Functional as in functional languages.

routine is a function accepting a derivative expression node and returning an expression where the single variable derivative has been computed. This routine can be an implementation of Algorithm 8. The result of `apply_ad` is mathematically equivalent to the input, but with no derivative expression nodes left⁶.

The function `apply_ad` works by traversing the tree recursively in post-order, discovering subtrees where the root represents a derivative, and applying the provided AD routine to the derivative subtree. Since the children of the derivative node has already been visited by `apply_ad`, they are guaranteed to be free of derivative expression nodes and the AD routine only needs to handle the case discussed above with algorithms 7 and 8.

The complexity of the `ad_routine` should be $O(n)$, with n being the size of the expression tree. The size of the derivative expression is proportional to the original expression. If there are d derivative expression nodes in the expression tree, the complexity of this algorithm is $O(dn)$, since `ad_routine` is applied to subexpressions d times. As a result the worst case complexity of `apply_ad` is $O(n^2)$, but in practice $d \ll n$. A recursive implementation of this algorithm is shown in Figure 19.4.

```
def apply_ad(e, ad_routine):
    if isinstance(e, Terminal):
        return e
    ops = [apply_ad(o, ad_routine) for o in e.operands()]
    e = e.reconstruct(*ops)
    if isinstance(e, Derivative):
        e = ad_routine(e)
    return e
```

Figure 19.4: Simple implementation of recursive `apply_ad` procedure.

19.7.6 Basic differentiation rules

To implement the algorithm descriptions above, we must implement differentiation rules for all expression node types. Derivatives of operators can be implemented as generic rules independent of the differentiation variable, and these are well known and not mentioned here. Derivatives of terminals depend on the differentiation variable type. Derivatives of literal constants are of course always zero, and only spatial derivatives of geometric quantities are non-zero. Since form arguments are unknown to UFL (they are provided externally by the form compilers), their spatial derivatives ($\frac{\partial \phi^k}{\partial x_i}$ and $\frac{\partial w^k}{\partial x_i}$) are considered input arguments as well. In all derivative computations, the assumption is made that

⁶Except direct spatial derivatives of form arguments, but that is an implementation detail.

form coefficients have no dependencies on the differentiation variable. Two more cases needs explaining, the user defined variables and derivatives w.r.t. the coefficients of a `Function`.

If v is a `Variable`, then we define $\frac{dt}{dv} \equiv 0$ for any terminal t . If v is scalar valued then $\frac{dv}{dv} \equiv 1$. Furthermore, if \mathbf{V} is a tensor valued `Variable`, its derivative w.r.t. itself is

$$\frac{d\mathbf{V}}{d\mathbf{V}} = \frac{dV_{ij}}{dV_{kl}} \mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k \otimes \mathbf{e}_l = \delta_{ik} \delta_{jl} \mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k \otimes \mathbf{e}_l. \quad (19.63)$$

In addition, the derivative of a variable w.r.t. something else than itself equals the derivative of the expression it represents:

$$v = g, \quad (19.64)$$

$$\frac{dv}{dz} = \frac{dg}{dz}. \quad (19.65)$$

Finally, we consider the operator derivative, which represents differentiation w.r.t. all coefficients $\{w_k\}$ of a function w . Consider an object element which represents a finite element space V_h with a basis $\{\phi_k\}$. Next consider form arguments defined in this space:

```
v = BasisFunction(element)
w = Function(element)
```

The `BasisFunction` instance v represents any $v \in \{\phi_k\}$, while the `Function` instance w represents the sum

$$w = \sum_k w_k \phi_k(x). \quad (19.66)$$

The derivative of w w.r.t. any w_k is the corresponding basis function in V_h ,

$$\frac{\partial w}{\partial w_k} = \phi_k, \quad k = 1, \dots, |V_h|, \quad (19.67)$$

$$(19.68)$$

which can be represented by v , since

$$v \in \langle \phi_k \rangle_{k=1}^{|V_h|} = \left\langle \frac{\partial w}{\partial w_k} \right\rangle_{k=1}^{|V_h|}. \quad (19.69)$$

Note that v should be a basis function instance that has not already been used in the form.

19.8 Algorithms

In this section, some central algorithms and key implementation issues are discussed, much of which relates to the Python programming language. Thus, this section is mainly intended for developers and others who need to relate to UFL on a technical level.

19.8.1 *Effective tree traversal in Python*

Applying some action to all nodes in a tree is naturally expressed using recursion:

```
def walk(expression, pre_action, post_action):
    pre_action(expression)
    for o in expression.operands():
        walk(o)
    post_action(expression)
```

This implementation simultaneously covers pre-order traversal, where each node is visited before its children, and post-order traversal, where each node is visited after its children.

A more “pythonic” way to implement iteration over a collection of nodes is using generators. A minimal implementation of this could be

```
def post_traversal(root):
    for o in root.operands():
        yield post_traversal(o)
    yield root
```

which then enables the natural Python syntax for iteration over expression nodes:

```
for e in post_traversal(expression):
    post_action(e)
```

For efficiency, the actual implementation of `post_traversal` in UFL is not using recursion. Function calls are very expensive in Python, which makes the non-recursive implementation an order of magnitude faster than the above.

19.8.2 *Type based function dispatch in Python*

► Editor note: *Make code fit in box.*

A common task in both symbolic computing and compiler implementation is the selection of some operation based on the type of an expression node. For a

```
class ExampleFunction(MultiFunction):
    def __init__(self):
        MultiFunction.__init__(self)

    def terminal(self, expression):
        return "Got a Terminal subtype %s." % type(expression)

    def operator(self, expression):
        return "Got an Operator subtype %s." % type(expression)

    def basis_function(self, expression):
        return "Got a BasisFunction."

    def sum(self, expression):
        return "Got a Sum."

m = ExampleFunction()

cell = triangle
element = FiniteElement("CG", cell, 1)
x = cell.x
print m(BasisFunction(element))
print m(x)
print m(x[0] + x[1])
print m(x[0] * x[1])
```

Figure 19.5: Example declaration and use of a multifunction

selected few operations, this is done using overloading of functions in the subclasses of `Expr`, but this is not suitable for all operations.

In many cases type-specific operations must be implemented together in the algorithm instead of distributed across class definitions. One way to implement type based operation selection is to use a type switch, or a sequence of if-tests such as this:

```
if isinstance(expression, IntValue):
    result = int_operation(expression)
elif isinstance(expression, Sum):
    result = sum_operation(expression)
# etc.
```

There are several problems with this approach, one of which is efficiency when

there are many types to check. A type based function dispatch mechanism with efficiency independent of the number of types is implemented as an alternative through the class `MultiFunction`. The underlying mechanism is a dict lookup (which is $O(1)$) based on the type of the input argument, followed by a call to the function found in the dict. The lookup table is built in the `MultiFunction` constructor. Functions to insert in the table are discovered automatically using the introspection capabilities of Python.

A multifunction is declared as a subclass of `MultiFunction`. For each type that should be handled particularly, a member function is declared in the subclass. The `Expr` classes use the `CamelCaps` naming convention, which is automatically converted to `underscore_notation` for corresponding function names, such as `BasisFunction` and `basis_function`. If a handler function is not declared for a type, the closest superclass handler function is used instead. Note that the `MultiFunction` implementation is specialized to types in the `Expr` class hierarchy. The declaration and use of a multifunction is illustrated in Figure 19.5. Note that `basis_function` and `sum` will handle instances of the exact types `BasisFunction` and `Sum`, while `terminal` and `operator` will handle the types `SpatialCoordinate` and `Product` since they have no specific handlers.

19.8.3 *Implementing expression transformations*

Many transformations of expressions can be implemented recursively with some type-specific operation applied to each expression node. Examples of operations are converting an expression node to a string representation, an expression representation using an symbolic external library, or an UFL representation with some different properties. A simple variant of this pattern can be implemented using a multifunction to represent the type-specific operation:

```
def apply(e, multifunction):
    ops = [apply(o, multifunction) for o in e.operands()]
    return multifunction(e, *ops)
```

The basic idea is as follows. Given an expression node `e`, begin with applying the transformation to each child node. Then return the result of some operation specialized according to the type of `e`, using the already transformed children as input.

The `Transformer` class implements this pattern. Defining a new algorithm using this pattern involves declaring a `Transformer` subclass, and implementing the type specific operations as member functions of this class just as with `MultiFunction`. The difference is that member functions take one additional argument for each operand of the expression node. The transformed child nodes are supplied as these additional arguments. The following code replaces terminal objects with objects found in a dict mapping, and reconstructs operators with

the transformed expression trees. The algorithm is applied to an expression by calling the function `visit`, named after the similar Visitor pattern.

```
class Replacer(Transformer):
    def __init__(self, mapping):
        Transformer.__init__(self)
        self.mapping = mapping

    def operator(self, e, *ops):
        return e.reconstruct(*ops)

    def terminal(self, e):
        return self.mapping.get(e, e)

f = Constant(triangle)
r = Replacer({f: f**2})
g = r.visit(2*f)
```

After running this code the result is $g = 2f^2$. The actual implementation of the `replace` function is similar to this code.

In some cases, child nodes should not be visited before their parent node. This distinction is easily expressed using `Transformer`, simply by omitting the member function arguments for the transformed operands. See the source code for many examples of algorithms using this pattern.

19.8.4 Important transformations

There are many ways in which expression representations can be manipulated. Here, we describe a few particularly important transformations. Note that each of these algorithms removes some abstractions, and hence may remove some opportunities for analysis or optimization.

Some operators in UFL are termed “compound” operators, meaning they can be represented by other elementary operators. Try defining an expression `e = inner(grad(u), grad(v))`, and `print repr(e)`. As you will see, the representation of `e` is `Inner(Grad(u), Grad(v))` (with some more details for `u` and `v`). This way the input expressions are easier to recognize in the representation, and rendering of expressions to for example \LaTeX format can show the original compound operators as written by the end-user.

However, since many algorithms must implement actions for each operator type, the function `expand_compounds` is used to replace all expression nodes of “compound” types with equivalent expressions using basic types. When this operation is applied to the input forms from the user, algorithms in both UFL and the form compilers can still be written purely in terms of basic operators.

Another important transformation is `expand_derivatives`, which applies automatic differentiation to expressions, recursively and for all kinds of derivatives. The end result is that most derivatives are evaluated, and the only derivative operator types left in the expression tree applies to terminals. The precondition for this algorithm is that `expand_compounds` has been applied.

Index notation and the `IndexSum` expression node type complicate interpretation of an expression tree in some contexts, since free indices in its summand expression will take on multiple values. In some cases, the transformation `expand_indices` comes in handy, the end result of which is that there are no free indices left in the expression. The precondition for this algorithm is that `expand_compounds` and `expand_derivatives` have been applied.

19.8.5 *Evaluating expressions*

Even though UFL expressions are intended to be compiled by form compilers, it can be useful to evaluate them to floating point values directly. In particular, this makes testing and debugging of UFL much easier, and is used extensively in the unit tests. To evaluate an UFL expression, values of form arguments and geometric quantities must be specified. Expressions depending only on spatial coordinates can be evaluated by passing a tuple with the coordinates to the call operator. The following code from an interactive Python session shows the syntax:

```
>>> cell = triangle
>>> x = cell.x
>>> e = x[0]+x[1]
>>> print e((0.5,0.7))
1.2
```

Other terminals can be specified using a dictionary that maps from terminal instances to values. This code extends the above code with a mapping:

```
c = Constant(cell)
e = c*(x[0]+x[1])
print e((0.5,0.7), { c: 10 })
```

If functions and basis functions depend on the spatial coordinates, the mapping can specify a Python callable instead of a literal constant. The callable must take the spatial coordinates as input and return a floating point value. If the function being mapped is a vector function, the callable must return a tuple of values instead. These extensions can be seen in the following code:

```
element = VectorElement("CG", cell, 1)
f = Function(element)
```



```
e = c*(f[0] + f[1])
def fh(x):
    return (x[0], x[1])
print e((0.5,0.7), { c: 10, f: fh })
```

To use expression evaluation for validating that the derivative computations are correct, spatial derivatives of form arguments can also be specified. The callable must then take a second argument which is called with a tuple of integers specifying the spatial directions in which to differentiate. A final example code computing $g^2 + g_{,0}^2 + g_{,1}^2$ for $g = x_0x_1$ is shown below.

```
element = FiniteElement("CG", cell, 1)
g = Function(element)
e = g**2 + g.dx(0)**2 + g.dx(1)**2
def gh(x, der=()):
    if der == ():    return x[0]*x[1]
    if der == (0,): return x[1]
    if der == (1,): return x[0]
print e((2, 3), { g: gh })
```

19.8.6 Viewing expressions

Expressions can be formatted in various ways for inspection, which is particularly useful while debugging. The Python built in string conversion operator `str(e)` provides a compact human readable string. If you type `print e` in an interactive Python session, `str(e)` is shown. Another Python built in string operator is `repr(e)`. UFL implements `repr` correctly such that `e == eval(repr(e))` for any expression `e`. The string `repr(e)` reflects all the exact representation types used in an expression, and can therefore be useful for debugging. Another formatting function is `tree_format(e)`, which produces an indented multi-line string that shows the tree structure of an expression clearly, as opposed to `repr` which can return quite long and hard to read strings. Information about formatting of expressions as \LaTeX and the dot graph visualization format can be found in the manual.

19.9 Implementation issues

19.9.1 Python as a basis for a domain specific language

Many of the implementation details detailed in this section are influenced by the initial choice of implementing UFL as an embedded language in Python.

Therefore some words about why Python is suitable for this, and why not, are appropriate here.

Python provides a simple syntax that is often said to be close to pseudo-code. This is a good starting point for a domain specific language. Object orientation and operator overloading is well supported, and this is fundamental to the design of UFL. The functional programming features of Python (such as generator expressions) are useful in the implementation of algorithms and form compilers. The built-in data structures `list`, `dict` and `set` play a central role in fast implementations of scalable algorithms.

There is one problem with operator overloading in Python, and that is the comparison operators. The problem stems from the fact that `__eq__` or `__cmp__` are used by the built-in data structures `dict` and `set` to compare keys, meaning that `a == b` must return a boolean value for `Expr` to be used as keys. The result is that `__eq__` can not be overloaded to return some `Expr` type representation such as `Equals(a, b)` for later processing by form compilers. The other problem is that `and` and `or` cannot be overloaded, and therefore cannot be used in conditional expressions. There are good reasons for these design choices in Python. This conflict is the reason for the somewhat non-intuitive design of the comparison operators in UFL.

19.9.2 *Ensuring unique form signatures*

The form compilers need to compute a unique signature of each form for use in a cache system to avoid recompilations. A convenient way to define a signature is using `repr(form)`, since the definition of this in Python is `eval(repr(form)) == form`. Therefore `__repr__` is implemented for all `Expr` subclasses.

Some forms are equivalent even though their representation is not exactly the same. UFL does not use a truly canonical form for its expressions, but takes some measures to ensure that trivially equivalent forms are recognized as such.

Some of the types in the `Expr` class hierarchy (subclasses of `Counted`), has a global counter to identify the order in which they were created. This counter is used by form arguments (both `BasisFunction` and `Function`) to identify their relative ordering in the argument list of the form. Other counted types are `Index` and `Label`, which only use the counter as a unique identifier. Algorithms are implemented for renumbering of all `Counted` types such that all counts start from 0.

In addition, some operator types such as `Sum` and `Product` maintains a sorted list of operands such that `a+b` and `b+a` are both represented as `Sum(a, b)`. The numbering of indices does not affect this ordering because a renumbering of the indices would lead to a new ordering which would lead to a different index renumbering if applied again. The operand sorting and renumbering combined ensure that the signature of equal forms will stay the same. To get the signature with renumbering applied, use `repr(form.form_data().form)`. Note that the

representation, and thus the signature, of a form may change with versions of UFL.

19.9.3 *Efficiency considerations*

By writing UFL in Python, we clearly do not put peak performance as a first priority. If the form compilation process can blend into the application build process, the performance is sufficient. We do, however, care about scaling performance to handle complicated equations efficiently, and therefore about the asymptotic complexity of the algorithms we use.

To write clear and efficient algorithms in Python, it is important to use the built in data structures correctly. These data structures include in particular `list`, `dict` and `set`. CPython [?], the reference implementation of Python, implements the data structure `list` as an array, which means `append`, and `pop`, and random read or write access are all $O(1)$ operations. Random insertion, however, is $O(n)$. Both `dict` and `set` are implemented as hash maps, the latter simply with no value associated with the keys. In a hash map, random read, write, insertion and deletion of items are all $O(1)$ operations, as long as the key types implement `__hash__` and `__eq__` efficiently. Thus to enjoy efficient use of these containers, all `Expr` subclasses must implement these two special functions efficiently. The `dict` data structure is used extensively by the Python language, and therefore particular attention has been given to make it efficient [?].

19.10 Future directions

Many additional features can be introduced to UFL. Which features are added will depend on the needs of FEniCS users and developers. Some features can be implemented in UFL alone, while other features will require updates to other parts of the FEniCS project.

Improvements to finite element declarations is likely easy to do in UFL. The added complexity will mostly be in the form compilers. Among the current suggestions are space-time elements and related time derivatives, and enrichment of finite element spaces. Additional geometry mappings and finite element spaces with non-uniform cell types are also possible extensions.

Additional operators can be added to make the language more expressive. Some operators are easy to add because their implementation only affects a small part of the code. More compound operators that can be expressed using elementary operations is easy to add. Additional special functions are easy to add as well, as long as their derivatives are known. Other features may require more thorough design considerations, such as support for complex numbers which may affect many parts of the code.

User friendly notation and support for rapid development are core values in the design of UFL. Having a notation close to the mathematical abstractions allows expression of particular ideas more easily, which can reduce the probability of bugs in user code. However, the notion of metaprogramming and code generation adds another layer of abstraction which can make understanding the framework more difficult for end-users. Good error checks everywhere are therefore very important, to detect user errors as close as possible to the user input. The error messages, documentation, and unit test suite should be improved to help avoid frequently repeated errors and misunderstandings among new users.

Several algorithms in UFL can probably be optimized if bottlenecks are found as more complicated applications are attempted. The focus in the development has not been on achieving peak performance, which is not important in a tool like UFL.

To support form compiler improvements, algorithms and utilities for generating better code more efficiently can be implemented in UFL. In this area, more work on alternative automatic differentiation algorithms [?, ?] can be useful. Another possibility for code improvement is operation scheduling, or reordering of the vertices of a graph partition to improve the efficiency of the generated code by better use of hardware cache and registers. Since modern C++ compilers are quite good at optimizing low level code, the focus should be on high level optimizations when considering potential code improvement in UFL and the form compilers. At the time of writing, operation scheduling is not implemented in UFL, and the value of implementing such an operation is an open question. However, results from [?] indicates that a high level scheduling algorithm could improve the efficiency of the generated code.

To summarize, UFL brings important improvements to the FEniCS framework: a richer form language, automatic differentiation and improved form compiler efficiency. These are useful features in rapid development of applications for efficiently solving partial differential equations. UFL improves upon the Automation of Discretization that has been the core feature of this framework, and adds Automation of Linearization. In conclusion, UFL brings FEniCS one step closer to its overall goal Automation of Mathematical Modeling.

19.11 Acknowledgements

This work has been supported by the Norwegian Research Council (grant 162730) and Simula Research Laboratory. I wish to thank everyone who has helped improving UFL with suggestions and testing, in particular Anders Logg, Kristian Ølgaard, Garth Wells, and Harish Narayanan. Both Kent-André Mardal and Marie Rognes performed critical reviews which greatly improved this manuscript.

Unicorn: A Unified Continuum Mechanics Solver

By Johan Hoffman, Johan Jansson, Niclas Jansson and Murtazo Nazarov

Chapter ref: [**hoffman-2**]

Unicorn is solver technology (models, methods, algorithms and software implementations) targeting simulation of realistic continuum mechanics applications, such as drag/lift computation for fixed or flexible objects (fluid-structure interaction) in turbulent incompressible or compressible flow (airplane/bird flight, car aerodynamics). The basis for Unicorn is Unified Continuum (UC) modeling, where we define conservation equations for mass, momentum and energy over the whole domain as one continuum, together with a Cauchy stress and phase variable as data for defining material properties and constitutive equation. For the discretization we use a stabilized adaptive finite element method which we refer to as General Galerkin (G2), which has been shown to accurately compute quantities of interest in both laminar and turbulent flow [?, ?, ?, ?, ?], where the methodology includes deforming geometries with an Arbitrary Lagrangian-Eulerian (ALE) discretization [?, ?].

This chapter provides a description of the technology in Unicorn focusing on efficient and general algorithms and software implementation of the UC concept and the adaptive G2 discretization. We describe how Unicorn fits into the FEniCS framework, how it interfaces to other FEniCS components (FIAT, FFC, DOLFIN) and what interfaces and functionality Unicorn provides itself and how the implementation is designed. We also give application examples in incompressible turbulent flow, fluid-structure interaction and compressible flow for illustration.

Unicorn realizes automated computational modeling in the form of tensor assembly, time-stepping, adaptive fixed-point iteration for solving discrete sys-

tems, duality-based adaptive error control, mesh adaptivity by local cell operations (split, collapse, swap) and cell quality optimization (elastic mesh moothing). We also describe the implementation of key concepts for efficient computation of large-scale turbulent flow problems: friction boundary conditions and parallelization of tensor assembly and mesh refinement.

20.1 Unified Continuum modeling

We define an incompressible unified continuum model in a fixed Euler coordinate system consisting of:

- conservation of mass
- conservation of momentum
- phase convection equation
- constitutive equations for stress as data

where the stress is the Cauchy (laboratory) stress and the phase variable is used to define material data such as constitutive equation for the stress and material parameters. Note that in this continuum description the coordinate system is fixed (Euler), and a phase function (marker) is convected according to the phase convection equation.

We start with conservation of mass, momentum and energy, together with a convection equation for a phase function θ over a space-time domain $Q = [\Omega \times [0, T]]$ with Ω an open domain in R^3 with boundary Γ :

$$\begin{aligned}
 D_t \rho + D_{x_j}(u_j \rho) &= 0 && \text{(Mass conservation)} \\
 D_t m_i + D_{x_j}(u_j m_i) &= D_{x_j} \sigma_i && \text{(Momentum conservation)} \\
 D_t e + D_{x_j}(u_j e) &= D_{x_j} \sigma_i u_i && \text{(Energy conservation)} \\
 D_t \theta + D_{x_j} u_j \theta &= 0 && \text{(Phase convection equation)}
 \end{aligned}
 \tag{20.1}$$

together with initial and boundary conditions. We can then pose constitutive relations between the constitutive (Cauchy) stress component σ and other variables such as the velocity u .

We define incompressibility as:

$$D_t \rho + u_j D_{x_j} \rho = 0$$

which together with mass and momentum conservation gives:

$$\begin{aligned}
 \rho(D_t u_i + u_j D_j u_i) &= D_{x_j} \sigma_{ij} \\
 D_{x_j} u_j &= 0
 \end{aligned}$$

where now the energy equation is decoupled and we can omit it.

We decompose the total stress into constitutive and forcing stresses:

$$D_{x_j}\sigma_{ij} = D_{x_j}\sigma_{ij} + D_{x_j}\sigma_{ij}^f = D_{x_j}\sigma_{ij} + f_i$$

Summarizing, we end up with the incompressible UC formulation:

$$\begin{aligned} \rho(D_t u_i + u_j D_{x_j} u_i) &= D_{x_j} \sigma_{ij} + f_i \\ D_{x_j} u_j &= 0 \\ D_t \theta + D_{x_j} u_j \theta &= 0 \end{aligned} \tag{20.2}$$

The UC modeling framework is simple and compact, close to the formulation of the original conservation equations, without mappings between coordinate systems. This allows simple manipulation and processing for error estimation and implementation. It is also general, we can choose the constitutive equations to model simple or complex solids and fluids, possibly both in interaction, with individual parameters.

20.1.1 Automated computational modeling and software design

One key design choice of UC modeling is to define the Cauchy stress σ as data, which means the conservation equations for momentum and mass are fixed and explicitly defined regardless of the choice of constitutive equation. This gives a generality in software design, where a modification of constitutive equation impacts the implementation of the constitutive equation, but not the implementation of the conservation equations.

20.2 Space-time General Galerkin discretization

The General Galerkin (G2) method has been developed as an adaptive stabilized finite element method for turbulent incompressible/compressible flow [?, ?, ?, ?, ?, ?, ?]. G2 has been shown to be cheap, since the adaptive mesh refinement is minimizing the number of degrees of freedom, general, since there are no model parameters to fit, and reliable, since the method is based on quantitative error control in a chosen output.

We begin by describing the standard FEM applied to the model to establish basic notation, and proceed to describe streamline diffusion stabilization and local ALE map over a mesh T^h with mesh size h together with adaptive error control based on duality.

20.2.1 Standard Galerkin

We begin by formulating the standard cG(1)cG(1) FEM [?] with piecewise continuous linear solution in time and space for 20.1 by defining the exact solution: $w = [u, p, \theta]$, the discrete solution $W = [U, P, \Theta]$ and the residual $R(W) = [R_u(W), R_p(W), R_\theta(W)]$:

$$\begin{aligned} R_u(W) &= \rho(D_t U_i + U_j D_{x_j} U_i) - D_{x_j} \Sigma_{ij} - f_i \\ R_p(W) &= D_{x_j} U_j \\ R_\theta(W) &= D_t \Theta + u_j D_{x_j} \Theta \end{aligned}$$

where $R(w) = 0$ and Σ denotes a discrete piecewise constant stress.

To determine the degrees of freedom ξ we enforce the Galerkin orthogonality $(R(W), v) = 0, \forall v \in V_h$ where v are test functions in the space of piecewise linear continuous functions in space and piecewise constant discontinuous functions in time and (\cdot, \cdot) denotes the space-time L_2 inner product over \mathbb{Q} . We thus have the weak formulation:

$$\begin{aligned} (R^u(W), v^u) &= (\rho(D_t U_i + U_j D_j U_i) - f_i, v_i^u) + (\Sigma_{ij}, D_{x_j} v_i^u) - \int_{t_{n-1}}^{t_n} \int_{\Gamma} \Sigma_{ij} v_i^u n_j ds dt = 0 \\ (R^p(W), v^p) &= (D_{x_j} U_j, v^p) = 0 \\ (R^\theta(W), v^\theta) &= (D_t \Theta + u_j D_{x_j} \Theta, v^\theta) = 0 \end{aligned}$$

for all $v \in V_h$, where the boundary term on Γ arising from integration by parts vanishes if we assume a homogenous Neumann boundary condition for the stress Σ .

This standard finite element formulation is unstable for convection-dominated problems and due to choosing equal order for the pressure and velocity. Thus we cannot use the standard finite element formulation by itself but proceed to a streamline diffusion stabilization formulation. We also describe a local ALE discretization for handling the phase interface.

20.2.2 Local ALE

If the phase function Θ has different values on the same cell it would lead to an undesirable diffusion of the phase interface. By introducing a local ALE coordinate map [?] on each discrete space-time slab based on a given mesh velocity (i.e. the material velocity of one of the phases) we can define the phase interface at cell facets, allowing the interface to stay discontinuous. We describe the details of the coordinate map and its influence on the FEM discretization in the appendix. The resulting discrete phase equation is:

$$D_t \Theta(x) + (U(x) - \beta_h(x)) \cdot \nabla \Theta(x) = 0 \quad (20.3)$$

with $\beta_h(x)$ the mesh velocity.

We thus choose the mesh velocity β_h to be the discrete material velocity U in the structure part of the mesh (vertices touching structure cells) and in the rest of the mesh we use mesh smoothing to determine β_h to maximize the mesh quality according to a chosen objective, alternatively use local mesh modification operations (refinement, coarsening, swapping) on the mesh to maintain the quality [?]. Note that we still compute in Euler coordinates, but with a moving mesh.

20.2.3 Streamline diffusion stabilization

For the standard FEM formulation of the model we only have stability of U but not of spatial derivatives of U . This means the solution can be oscillatory, causing inefficiency by introducing unnecessary error. We instead choose a weighted standard Galerkin/streamline diffusion method of the form $(R(W), v + \delta R(v)) = 0, \forall v \in V_h$ (see [?]) with $\delta > 0$ a stabilization parameter. We here also make a simplification where we only introduce necessary stabilization terms and drop terms not contributing to stabilization. Although not fully consistent, the streamline diffusion stabilization avoid unnecessary smearing of shear layers as the stabilization is not based on large ($\approx h^{-\frac{1}{2}}$) cross flow derivatives). For the UC model the stabilized method thus looks like:

$$\begin{aligned} (R^u(W), v^u) &= (\rho(D_t U_i + U_j D_j U_i) - f_i, v_i^u) + (\Sigma_{ij}, D_{x_j} v_i^u) + SD^u(W, v^u) = 0 \\ (R^p(W), v^p) &= (D_{x_j} U_j, v^p) + SD^p(W, v^p) = 0 \end{aligned}$$

for all $v \in V_h$, and:

$$\begin{aligned} SD^u(W, v^u) &= \delta_1 (U_j D_j U_i, U_j^u D_j v_i^u) + \delta_2 (D_{x_j} U_j, D_{x_j} v_j^u) \\ SD^p(W, v^p) &= \delta_1 (D_{x_i} P, D_{x_i} v^p) \end{aligned}$$

where we only include the dominating stabilization terms to reduce complexity in the formulation.

20.2.4 Duality-based adaptive error control

20.2.5 Unicorn / FEniCS software implementation

We implement the G2 discretization of the UC in a general interface for time-dependent PDE where we give the forms $a(U, v) = (D_U F_U, v)$ and $L(v) = (F_U, v)$ for assembling the linear system given by Newton's method for a time step for the incompressible UC with Newtonian fluid constitutive equation in figure 20.1. The language used is Python, where we use the FEniCS Form Compiler (FFC) [?] form notation.

```

...

def ugradu(u, v):
    return [dot(u, grad(v[i])) for i in range(d)]

def epsilon(u):
    return 0.5 * (grad(u) + transp(grad(u)))

def S(u, P):
    return mult(P, Identity(d)) - mult(nu, grad(u))

def f(u, v):
    return -dot(ugradu(Uc, Uc), v) + \
        dot(S(Uc, P), grad(v)) + \
        -mult(d1, dot(ugradu(Um, u), ugradu(Um, v))) + \
        -mult(d2, dot(div(u), div(v))) + \
        dot(ff, v)

def dfdu(u, k, v):
    return -dot(ugradu(Um, u), v) + \
        -dot(mult(nu, grad(u)), grad(v)) + \
        -mult(d1, dot(ugradu(Um, u), ugradu(Um, v))) + \
        -mult(d2, dot(div(u), div(v)))

# cG(1)
def F(u, u0, k, v):
    uc = 0.5 * (u + u0)
    return (-dot(u, v) + dot(u0, v) + mult(k, f(u, v)))

def dFdu(u, u0, k, v):
    uc = 0.5 * u
    return (-dot(u, v) + mult(1.0 * k, dfdu(uc, k, v)))

a = (dFdu(U1, U0, k, v)) * dx
L = -F(UP, U0, k, v) * dx
    
```

Figure 20.1: Source code for bilinear and linear forms for incompressible UC one time step with a Newton-type method (approximation of Jacobian).

20.3 Unicorn classes: data types and algorithms

20.3.1 Unicorn software design

Unicorn follows two basic design principles:

- Keep It Simple Stupid (KISS)
- “Premature optimization is the root of all evil” (Donald Knuth)

Together, these two principles enforce generality and understandability of interfaces and implementations. Unicorn re-uses other existing implementations and chooses straightforward, sufficiently efficient (optimize bottlenecks) standard algorithms for solving problems. This leads to small and maintainable implementations. High performance is achieved by reducing the computational load on the method level (through adaptivity and fixed-point iteration).

Unicorn consists of key concepts abstracted in the following classes/interfaces:

TimeDependentPDE: time-stepping In each time-step a non-linear algebraic system is solved by fixed-point iteration.

ErrorEstimate: adaptive error control The adaptive algorithm is based on computing local *error indicators* of the form $\epsilon_K = (R(U), D_x \Phi)_{L_2(K \times T)}$. This algorithm is abstracted in the `ErrorEstimate` and class.

SlipBC: friction boundary condition Efficient computation of turbulent flow in Unicorn is based on modeling of turbulent boundary layers by a friction model: $u \cdot n = 0$, implemented as a strong boundary condition in the algebraic system.

20.3.2 TimeDependentPDE

We consider time-dependent equations of the type $f(u) = -D_t u + g(u) = 0$ where g can include differential operators in space, where specifically the UC model is of this type. In weak form the equation type looks like $(f(u), v) = (-D_t u + g(u), v) = 0$, possibly with partial integration of terms

We want to define a class (datatype and algorithms) abstracting the time-stepping of the G2 method, where we want to give the equation (possibly in weak form) as input and generate the time-stepping automatically. `cG(1)cG(1)` (Crank-Nicolson in time) gives the equation for the (possibly non-linear) algebraic system $F(U)$ (in Python notation):

```
# cG(1)
def F(u, u0, k, v):
    uc = 0.5 * (u + u0)
    return (-dot(u, v) + dot(u0, v) + mult(k, g(uc, v)))
```

With $v: \forall v \in V_h$ generating the equation system.

We solve this system by Newton-type fixed-point iteration:

$$(F'(U_P)U_1, v) = (F'(U_P) - F(U_P), v) \quad (20.4)$$

where U_P denotes the value in the previous iterate and $F' = \frac{\partial F}{\partial U}$ the Jacobian matrix or an approximation. Note that F' can be chosen freely since it only affects the convergence of the fixed-point iteration, and does not introduce approximation error.

We define the bilinear form $a(U, v)$ and linear form $L(v)$ corresponding to the left and right hand sides respectively (in Python notation):

```
def dFdu(u, u0, k, v):
    uc = 0.5 * u
    return (-dot(u, v) + mult(k, dgdu(uc, k, v)))

a = (dFdu(U, U0, k, v)) * dx
L = (dFdu(UP, U0, k, v) - F(UP, U0, k, v)) * dx
```

Thus, in each time step we need to solve the system given in eq. 20.4 by fixed-point iteration by repeatedly computing a and L , solving a linear system and updating U .

We now encapsulate this in a C++ class interface in fig. 20.4 which we call `TimeDependentPDE` where we give a and L , an end time T , a mesh (defining V_h) and boundary conditions.

The skeleton of the time-stepping with fixed-point iteration is implemented in listing 20.3.2.

See ?? and [?] for a discussion about the efficiency of the fixed-point iteration and its implementation.

[Discuss pressure/systems]

20.3.3 ErrorEstimate

The duality-based adaptive error control algorithm requires the following primitives:

Residual computation We compute the mean-value in each cell of the continuous residual $R(U) = f(U) = -D_t U + g(U)$, this is computed as the L_2 -projection into the space of piecewise constants W_h : $(R(U), v) = (-D_t U + g(U), v), \forall v \in W_h$.

Dual solution We compute the solution of the dual problem using the same technology as the primal problem. The dual problem is solved backward in time, but with the time coordinate transform $s = T - t$ we can use the standard `TimeDependentPDE` interface and step the dual time s forward.

Space-time function storage/evaluation We compute error indicators as space-time integrals over cells: $\epsilon_K = (R(U), D_x \Phi)_{L_2(K \times T)}$, where we need to evaluate both the primal solution U and the dual solution Φ . In addition, U is

```

/// Represent and solve time dependent PDE.
class TimeDependentPDE
{
/// Public interface
public:
    TimeDependentPDE(
        // Computational mesh
        Mesh& mesh,
        // Bilinear form for Jacobian approx.
        Form& a,
        // Linear form for time-step residual
        Form& L,
        // List of boundary conditions
        Array <BoundaryCondition*>& bcs,
        // End time
        real T
    );
    virtual ~TimeDependentPDE();
    /// Solve PDE
    virtual uint solve();

/// Protected interface for subclasses
protected:
    /// Compute initial value
    virtual void u0(Vector& u);
    /// Called before each time step
    virtual void preparestep();
    /// Called before each fixed-point iteration
    virtual void prepareiteration();
    /// Return the bilinear form a
    Form& a();
    /// Return the linear form L
    Form& L();
    /// Return the mesh
    Mesh& mesh();
};

```

Figure 20.2: C++ class interface for TimeDependentPDE.

a coefficient in the dual equation. This requires storage and evaluation of a space-time function, which is encapsulated in the `SpaceTimeFunction` class.

Mesh adaptation After the computation of the error indicators we select the largest $p\%$ of the indicators for refinement. The refinement is then performed by recursive Rivara cell bisection encapsulated in the `MeshAdaptivity` class. A future promising alternative is to use Madlib [?, ?] for mesh adaptation, which is based on edge split, collapse and swap, and would thus give the ability to coarsen a mesh, or more generally to control the mesh size.

Using these primitives, we can construct an adaptive algorithm. The adaptive algorithm is encapsulated in the C++ class interface in fig. ?? which we call `ErrorEstimate`.

20.3.4 SlipBC

For high Reynolds numbers problems such as car aerodynamics or airplane flight, it's not possible to resolve the turbulent boundary layer. One possibility is to model turbulent boundary layers by a friction model:

$$u \cdot n = 0 \tag{20.5}$$

$$u \cdot \tau_k + \beta^{-1} n^\top \sigma \tau_k = 0, k = 1, 2 \tag{20.6}$$

We implement the normal component condition (slip) boundary condition strongly. By “strongly” we here mean an implementation of the boundary condition after assembling the left hand side matrix and the right hand side vector in the algebraic system, whereas the tangential components (friction) are implemented “weakly” by adding boundary integrals in the variational formulation. The row of the matrix and load vector corresponding to a vertex is found and replaced by a new row according to the boundary condition.

The idea is as follows: Initially, the test function v is expressed in the Cartesian standard basis (e_1, e_2, e_3) . Now, the test function is mapped locally to normal-tangent coordinates with the basis (n, τ_1, τ_2) , where $n = (n_1, n_2, n_3)$ is the normal, and $\tau_1 = (\tau_{11}, \tau_{12}, \tau_{13})$, $\tau_2 = (\tau_{21}, \tau_{22}, \tau_{23})$ are tangents to each node on the boundary. This allows us to let the normal direction to be constrained and the tangent directions be free:

$$v = (v \cdot n)n + (v \cdot \tau_1)\tau_1 + (v \cdot \tau_2)\tau_2.$$

For the matrix and vector this means that the rows corresponding to the boundary need to be multiplied with n, τ_1, τ_2 , respectively, and then the normal component of the velocity should be put 0.

This concept is encapsulated in the class `SlipBC` which is a subclass of `dolfin::BoundaryCondition` for representing strong boundary conditions.

20.4 Mesh adaptivity

20.4.1 *Local mesh operations: Madlib*

Madlib incorporates an algorithm and implementation of **mesh adaptation** where a small set of local mesh modification operators are defined such as edge split, edge collapse and edge swap. A mesh adaptation algorithm is defined which uses this set of local operators in a control loop to satisfy a prescribed size field $h(x)$ and quality tolerance. Edge swapping is the key operator for improving quality of cells, for example around a vertex with a large number of connected edges.

In the formulation of finite element methods it is typically assumed that the cell size of a computational mesh can be freely modified to satisfy a desired size field $h(x)$ or to allow mesh motion. In state-of-the-art finite element software implementations this is seldom the case, where typically only limited operations are allowed [?, ?], (local mesh refinement), or a separate often complex, closed and ad-hoc mesh generation implementation is used to re-generate meshes.

The mesh adaptation algorithm in Madlib gives the freedom to adapt to a specified size field using local mesh operations. The implementation is published as free software/open source allowing other research to build on the results and scientific repeatability of numerical experiments.

20.4.2 *Elastic mesh smoothing: cell quality optimization*

20.4.3 *Recursive Rivara bisection*

20.5 Parallel computation

20.5.1 *Tensor assembly*

20.5.2 *Mesh refinement*

20.6 Application examples

20.6.1 *Incompressible flow*

20.6.2 *Compressible flow*

20.6.3 *Fluid-structure interaction*

```

void TimeDependentPDE::solve()
{
    // Time-stepping
    while(t < T)
    {
        U = U0;
        preparestep();
        step();
    }
}

void TimeDependentPDE::step()
{
    // Fixed-point iteration
    for(int iter = 0; iter < maxiter; iter++)
    {
        prepareiteration();
        step_residual = iter();

        if(step_residual < tol)
        {
            // Iteration converged
            break;
        }
    }
}

void TimeDependentPDE::iter()
{
    // Compute one fixed-point iteration
    assemble(J, a());
    assemble(b, L());
    for (uint i = 0; i < bc().size(); i++)
        bc()[i]->apply(J, b, a());
    solve(J, x, b);

    // Compute residual for the time-step/fixed-point equation
    J.mult(x, residual);
    residual -= b;

    return residual.norm(linf);
}

```

Figure 20.3: Skeleton implementation²⁰⁴ in Unicorn of time-stepping with fixed-point iteration.


```

/// Represent and solve time dependent PDE.
class TimeDependentPDE
{
/// Public interface
public:
    TimeDependentPDE(
        // Computational mesh
        Mesh& mesh,
        // Bilinear form for Jacobian approx.
        Form& a,
        // Linear form for time-step residual
        Form& L,
        // List of boundary conditions
        Array <BoundaryCondition*>& bcs,
        // End time
        real T
    );
    virtual ~TimeDependentPDE();
/// Solve PDE
    virtual uint solve();

/// Protected interface for subclasses
protected:
    /// Compute initial value
    virtual void u0(Vector& u);
    /// Called before each time step
    virtual void preparestep();
    /// Called before each fixed-point iteration
    virtual void prepareiteration();
    /// Return the bilinear form a
    Form& a();
    /// Return the linear form L
    Form& L();
    /// Return the mesh
    Mesh& mesh();
};

```

Figure 20.4: C++ class interface for TimeDependentPDE.

CHAPTER 21

Viper: A Minimalistic Scientific Plotter

By Ola Skavhaug

Chapter ref: **[skavhaug]**

Lessons Learnt in Mixed Language Programming

By Kent-Andre Mardal, Anders Logg, and Ola Skavhaug

Chapter ref: [**mardal-2**]

► Editor note: *This could be in the implementation section or in an appendix.*

This chapter discusses some lessons learnt while developing the Python interface to DOLFIN. It contains the basics of using SWIG to create Python extensions. Then an explanation of some design choices in DOLFIN with particular emphasis on operators (and how these are dealt with in SWIG). Typemaps are explained in the context of NumPy arrays. Finally, we explain how to debug Python extensions efficiently, eg. by setting breakpoints.

Part III

Applications

Finite Elements for Incompressible Fluids

By Andy R. Terrel, L. Ridgway Scott, Matthew G. Knepley, Robert C. Kirby and Garth

N. Wells

Chapter ref: **[terrel]**

Incompressible fluid models have numerous discretizations each with its own benefits and problems. This chapter will focus on using FEniCS to implement discretizations of the Stokes and two non-Newtonian models, grade two and Oldroyd–B. Special consideration is given to profiling the discretizations on several problems.

Benchmarking Finite Element Methods for Navier–Stokes

By Kristian Valen-Sendstad, Anders Logg and Kent-Andre Mardal

Chapter ref: **[kvs-1]**

In this chapter, we discuss the implementation of several well-known finite element based solution algorithms for the Navier-Stokes equations. We focus on laminar incompressible flows and Newtonian fluids. Implementations of simple projection methods are compared to fully implicit schemes such as inexact Uzawa, pressure correction on the Schur complement, block preconditioning of the saddle point problem, and least-squares stabilized Galerkin. Numerical stability and boundary conditions are briefly discussed before we compare the implementations with respect to efficiency and accuracy for a number of well established benchmark tests.

Image-Based Computational Hemodynamics

By Luca Antiga

Chapter ref: [**antiga**]

The physiopathology of the cardiovascular system has been observed to be tightly linked to the local in-vivo hemodynamic environment. For this reason, numerical simulation of patient-specific hemodynamics is gaining ground in the vascular research community, and it is expected to start playing a role in future clinical environments. For the non-invasive characterization of local hemodynamics on the basis of information drawn from medical images, robust workflows from images to the definition and the discretization of computational domains for numerical analysis are required. In this chapter, we present a framework for image analysis, surface modeling, geometric characterization and mesh generation provided as part of the Vascular Modeling Toolkit (VMTK), an open-source effort. Starting from a brief introduction of the theoretical bases of which VMTK is based, we provide an operative description of the steps required to generate a computational mesh from a medical imaging data set. Particular attention will be devoted to the integration of the Vascular Modeling Toolkit with FEniCS. All aspects covered in this chapter are documented with examples and accompanied by code and data, which allow to concretely introduce the reader to the field of patient-specific computational hemodynamics.

Simulating the Hemodynamics of the Circle of Willis

By Kristian Valen-Sendstad, Kent-Andre Mardal and Anders Logg

Chapter ref: **[kvs-2]**

Stroke is a leading cause of death in the western world. Stroke has different causes but around 5-10% is the result of a so-called subarachnoid hemorrhage caused by the rupture of an aneurysm. These aneurysms are usually found in our near the circle of Willis, which is an arterial network at the base of the brain. In this chapter we will employ FEniCS solvers to simulate the hemodynamics in several examples ranging from simple time-dependent flow in pipes to the blood flow in patient-specific anatomies.

Cerebrospinal Fluid Flow

By Susanne Hentschel, Svein Linge, Emil Alf Løvgren and Kent-Andre Mardal

Chapter ref: **[hentschel]**

27.1 Medical Background

The cerebrospinal fluid (CSF) is a clear water-like fluid which occupies the so-called subarachnoid space (SAS) surrounding the brain and the spinal cord, and the ventricular system within the brain. The SAS is composed of a cranial and a spinal part, bounded by tissue layers, the dura mater as outer boundary and the pia mater as internal boundary. The cranial and the spinal SAS are connected by an opening in the skull, called the foramen magnum. One important function of the CSF is to act as a shock absorber and to allow the brain to expand and contract as a reaction to the changing cranial blood volume throughout the cardiac cycle. During systole the blood volume that enters the brain through the arterial system exceeds the volume that leaves the brain through the venous system and leads therefore to an expansion of the brain. The opposite effect occurs during diastole, when the blood volume in the brain returns to the starting point. Hence the pulse that travels through the blood vessel network is transformed to a pulse in the CSF system, that is damped on its way along the spinal canal.

The left picture in Figure 27.1 shows the CSF and the main structures in the brain of a healthy individual. In about 0.6% of the population the lower part of the cerebellum occupies parts of the CSF space in the upper spinal SAS and obstructs the flow. This so-called Chiari I malformation (or Arnold-Chiari malformation) (Milhorat et al. [?]) is shown in the right picture in Figure 27.1. A variety of symptoms is related to this malformation, including headache, abnor-

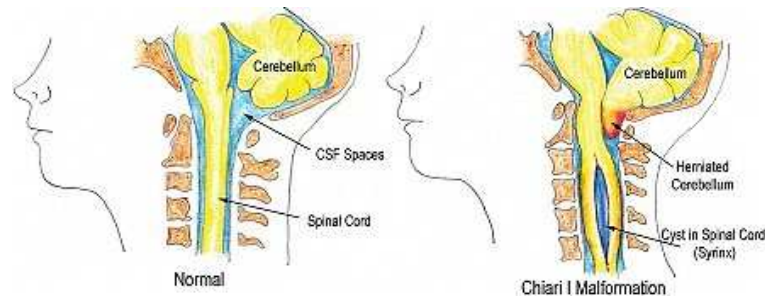


Figure 27.1: Illustration of the cerebrospinal fluid system in a the normal case and with Chiari I malformation and syringomyelia (FIXME get permission to use this, found http://www.chiariinstitute.com/chiari_malformation.html)

mal eye-movement, motor or sensor-dysfunctions, etc. If the malformation is not treated surgically, the condition may become more severe and will eventually cause more serious neurological deterioration, or even lead to death. Many people with the Chiari I malformation develop fluid filled cavities within the spinal cord, a disease called syringomyelia (Oldfield [?]). The exact relation between the Chiari I malformation and syringomyelia is however not known. It is believed that obstructions, that is abnormal anatomies cause abnormal flow leading to the development of syringomyelia (Oldfield [?]). Several authors have analyzed the relations between abnormal flow and syringomyelia development based on measurements in patients and healthy volunteers (Heiss [?], Pinna [?], Hofmann [?], Hentschel [?]). The mentioned studies also compare the dynamics before and after decompressive surgery. The latter is an operation, where the SAS lumen around the obstructed area is increased by removing parts of the surrounding tissue and bones (Milhorat and Bolognese [?]). Control images taken some weeks or months after the intervention often show a reduction of the size of the cavity in the spinal canal and patients usually report improvement of their condition. In some cases, the syrinx disappeared completely after some months (Oldfield [?], Pinna [?], Heiss [?]).

The studies mentioned above are all based on a small amount of individuals characterized by remarkable variations. CFD simulations may help to test the initial assumptions in generalized settings. Gupta [?] and Roldan [?] demonstrated the usefulness of CFD to quantify and visualize CSF flow in patient specific cases in great detail. It is the purpose of this chapter to describe the implementation of such a CFD solver in FEniCS and to compare the simulation results with results obtained from Star-CD. Note that the Navier-Stokes solvers are discussed in detail in Chapter [?].

27.2 Mathematical Description

We model the CSF flow in the upper spinal canal as a Newtonian fluid with viscosity and density similar to water under body temperature. In the presented experiments, we focus on the dynamics around the spinal cord. The tissue surrounding the fluid is modeled as impermeable and rigid throughout the cardiac cycle. To simulate CSF flow, we apply the Navier-Stokes equations for an incompressible Newtonian fluid,

$$\begin{aligned} \rho \left(\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla) \mathbf{v} \right) &= -\nabla p + \mu \Delta \mathbf{v} + \mathbf{g}, \quad \in \Omega, \\ \nabla \cdot \mathbf{v} &= 0, \quad \in \Omega, \end{aligned}$$

with the variables as indicated in Table 27.2, and g , the body force, i.e., gravity. We can eliminate gravity from the equation by assuming that the body force is balanced by the hydrostatic pressure. As a result, pressure describes only the dynamic pressure. For calculating the physical correct pressure, static pressure resulting from body forces has to be added. This simplification is however not true during sudden movements such as raising up.

The coordinate system is chosen so that the tubular axis points downwards, resulting in positive systolic flow and negative diastolic flow.

27.3 Numerical Experiments

27.3.1 Implementation

We refer to Chapter [?] for a complete description of the solvers and schemes implemented. In this chapter we concentrate on the use of these solvers in a few examples.

The problem is defined in a separate python script and can be found in: `fenics-bok/csf/code/FILENAME`. The main parts are presented below.

Mesh boundaries. The mesh boundaries at the inlet cross section, the outlet cross section, and the SAS boundaries are defined by the respective classes `Top`, `Bottom`, and `Contour`. They are implemented as subclasses of `SubDomain`, similarly to the given example of `Top`.

```
class Top(SubDomain):
    def __init__(self, index, z_max, z_min):
        SubDomain.__init__(self)
        self.z_index = index
        self.z_max = z_max
        self.z_min = z_min

    def inside(self, x, on_boundary):
        return bool(on_boundary and x[self.z_index] == self.z_max)
```

Symbol	Meaning	Entity	Chosen Value	Reference Value
v	velocity variable	$\frac{\text{cm}}{\text{s}}$	—	$-1.3 \pm 0.6 \dots 2.4 \pm 1.4^a$
p	pressure variable	mmHg	—	...
ρ	density	$\frac{\text{g}}{\text{cm}^3}$	—	0.993 ^b
μ	dynamic viscosity	$\frac{\text{gs}}{\text{cm}^2}$	—	0.0007
ν	kinematic viscosity	$\frac{\text{cm}^2}{\text{s}}$	0.710^{-2}	0.710^{-2}
SV	stroke volume ^c	$\frac{\text{ml}}{\text{s}}$	0.27	0.27^d
HR	heart rate	$\frac{\text{beats}}{\text{s}}$	1.17	1.17
$A0$	tube boundary	cm^2	32	—
$A1, A2$	area of inlet/outlet	cm^2	0.93	$0.8 \dots 1.1^e$
Re	Reynolds Number	—	—	$70-200^f$
We	Womersley Number	—	—	14–17

Table 27.1: Characteristic values and parameters for CSF flow modeling.

^aHofmann et al. [?]; Maximum absolute anterior CSF flow in both directions from controls and patients at foramen Magnum

^bat 37° C

^cCSF volume that moves up and down through cross section in the SAS during one cardiac cycle

^dGupta et al. [?]

^eLoth et al. [?]; Cross sections at 20–40 cm from the foramen magnum.

^fSee more details in 27.3.5.

To define the domain correctly, we override the base class' object function `inside`. It returns a boolean evaluating if the inserted point `x` is part of the sub domain. The boolean `on_boundary` is very useful to easily partition the whole mesh boundary to sub domains.

Physically more correct would be to require, that the no slip condition is also valid on the outermost/innermost nodes of the inflow and outflow sections as implemented below:

```
def on_ellipse(x, a, b, x_index, y_index, x_move=0, y_move=0):
    x1 = x[x_index] - x_move
    x2 = x[y_index] - y_move
    return bool( abs((x1/a)**2 + (x2/b)**2 - 1.0 ) < 10**(-6) )
```

The vectors describing the ellipses of the cord and the dura in a cross section with the corresponding axes are required. The global function `on_ellipse` checks if `x` is on the ellipse defined by the `x`-vector `a` and the `y`-vector `b`. The variables `x_move` and `y_move` allow to define an eccentric ellipse.

Defining the inflow area at the top with excluded mantle nodes is done as follows below, the outflow area at the bottom is defined analogously.

```
class Top(SubDomain): #bc for top
    def __init__(self, a2_o, a2_i, b2_o, b2_i, x_index, y_index, z_index, z_max, x2_o_move=0, \
                y2_o_move=0, x2_i_move=0, y2_i_move=0):
        SubDomain.__init__(self)
        self.x_index = x_index
        self.y_index = y_index
        self.a2_o = a2_o
        self.a2_i = a2_i
        self.b2_o = b2_o
        self.b2_i = b2_i
        self.z_index = z_index
        self.z_max = z_max
        self.x2_o_move = x2_o_move
        self.x2_i_move = x2_i_move
        self.y2_o_move = y2_o_move
        self.y2_i_move = y2_i_move

    def inside(self, x, on_boundary):
        return bool(on_boundary and abs(x[self.z_index] - self.z_max) < 10**(-6) \
                    and not on_ellipse(x, self.a2_o, self.b2_o, self.x_index, \
                                        self.y_index, self.x2_o_move, self.y2_o_move) \
                    and not on_ellipse(x, self.a2_i, self.b2_i, self.x_index, \
                                        self.y_index, self.x2_i_move, self.y2_i_move) )
```

The underscores `o` and `i` represent the outer and inner ellipse respectively. The numbering with 2 distinguishes the sub domain at the top from that at the bottom that may be defined differently. The details of how different problems can easily be defined in separate classes can be found in: `src/mesh_definitions/`.

Inflow and outflow pulse. According to Gupta et al. [?], a volume of 0.27 ml is transported back and forth through the spinal SAS cross sections during the cardiac cycle. For the average human, we assumed a heart rate of 70 beats per

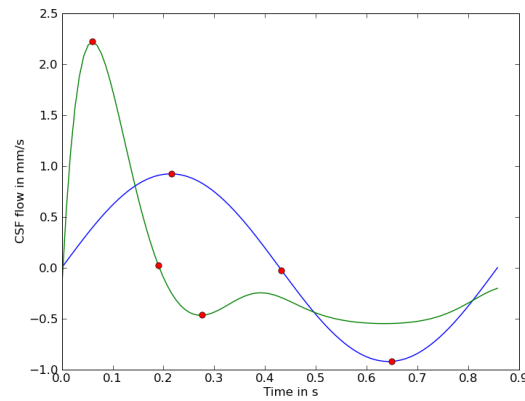


Figure 27.2: Two different flow pulses.

minute. Furthermore, we defined the cross sectional area to be 0.93 cm^2 , which matches the segment from 20 to 40 cm down from the foramen magnum (Loth et al [?]). In this region of the spinal canal, the cross sectional area varies little. In addition, the dura and the cord shape resemble a simple tube more than in other regions. According to Oldfield et al. [?], syringes start at around 5 cm below the foramen magnum and reach down up to 28 cm below the foramen magnum.

Further, we define a velocity pulse on the inflow and outflow boundaries and since we are modeling incompressible flow between rigid impermeable boundaries, we must have equal inflow and outflow volume at all times. The pulse values in these boundary cross sections were set equal in every grid point, and scaled to match the volume transport of 0.27 ml.

Smith et al. [?] introduced a function describing the varying blood pressure in a heart chamber(see Figure 27.2). With some adjustment and additional parameters, the function was adapted to approximate the CSF flow pulse. The systole of the pulse function is characterized by a high amplitude with a short duration while the negative counter movement has a reduced amplitude and lasts considerably longer. The global function for defining the pulse is:

```
def get_pulse_input_function(V, z_index, factor, A, HR_inv, HR, b, fl):
    two_pi = 3.4 * pi
    rad = two_pi /HR_inv
    v_z = "factor*(-A*(exp(-fmod(t,T)*rad)*Ees*(sin(-fl*fmod(t,T)*rad)-vd)\
        -(1-exp(-factor*fmod(t,T)*rad))*p0*(exp(sin(-fmod(t,T)*rad)-vd)-1))-b)"
    vel = None
    if z_index == 0:
        vel = (v_z, "0.0", "0.0")
    elif z_index ==1:
        vel = ("0.0", v_z, "0.0")
    elif z_index ==2:
        vel = ("0.0", "0.0", v_z)
```

```
class Pulse(Function):
    cpparg = vel
    print vel
    defaults = {"factor":factor, "A":A, "p0":1, "vd":0.03, "Ees":50, "T":HR_inv, "HR":HR,\
               "rad":rad, "b":b, \emp{f1}:f1}

    return Pulse(V)
```

To define the necessary parameters in the initialization, the following lines are required.

```
self.A = 2.9/16 # scale to get max = 2.5 and min = -0.5 for f1 = 1
self.factor = self.flow_per_unit_area/0.324
self.v_max = 2.5 * self.factor
self.b = 0.465 # translating the function "down"
self.f1 = 0.8
```

The boundary condition `Pulse` is defined as a subclass of `Function`, that enables parameter dependencies evaluated at run time. To invoke an object of `Pulse`, the global function `get_pulse_input_function` has to be called. The function input contains all necessary constants to define the pulse function, scaled to cardiac cycle and volume transport. The index `z_index` defines the coordinate of the tubular direction. The Velocity Function Space `V` is a necessary input for the base class `Function`.

Initialization of the problem. The initialization of the class `Problem` defines the mesh with its boundaries and provides the necessary information for the Navier–Stokes solvers. The mesh is ordered for all entities and initiated to compute its faces.

The values `z_min` and `z_max` mark the inflow and outflow coordinates along the tube’s length axis. As mentioned above, the axis along the tube is indicated by `z_index`. If one of the coordinates or the `z-index` is not known, it may help to call the mesh in `viper unix>viper meshname.xml`. Typing `o` prints the length in `x`, `y` and `z` direction in the terminal window. Defining `z_min`, `z_max` and `z_index` correctly is important for the classes that define the boundary domains of the mesh `Top`, `Bottom` and `Contour`. As we have seen before, `z_index` is necessary to set the correct component to the non-zero boundary velocity.

Exterior forces on the Navier–Stokes flow are defined in the object variable `f`. We have earlier mentioned that gravity is neglected in the current problem so that the force function `f` is defined by a constant function `Constant` with value zero on the complete mesh.

After initializing the sub domains, `Top`, `Bottom` and `Contour`, they are marked with reference numbers attributed to the collection of all sub domains `sub_domains`.

To see the most important effects, the simulation was run slightly longer than one full period. A test verified that the initial condition of zero velocity in all points is sufficiently correct and leads to a good result in the first period already.

Besides maximum and minimum velocities, it includes the transition from diastole to systole and vice versa. With the given time step length, the simulation is very detailed in time.

```
def __init__(self, options):
    ProblemBase.__init__(self, options)
    #filename = options["mesh"]
    filename = "../data/meshes/chiari/csf_extrude_2d_bd1.xml.gz"
    self.mesh = Mesh(filename)
    self.mesh.order()
    self.mesh.init(2)

    self.z_max = 5.0 # in cm
    self.z_min = 0.0 # in cm
    self.z_index = 2
    self.D = 0.5 # characteristic diameter in cm

    self.contour = Contour(self.z_index, self.z_max, self.z_min)
    self.bottom = Bottom(self.z_index, self.z_max, self.z_min)
    self.top = Top(self.z_index, self.z_max, self.z_min)

    # Load sub domain markers
    self.sub_domains = MeshFunction("uint", self.mesh, self.mesh.topology().dim() - 1)

    # Mark all facets as sub domain 3
    for i in range(self.sub_domains.size()):
        self.sub_domains.set(i, 3)

    self.contour.mark(self.sub_domains, 0)
    self.top.mark(self.sub_domains, 1)
    self.bottom.mark(self.sub_domains, 2)

    # Set viscosity
    self.nu = 0.7 *10**(-2) # cm^2/s

    # Create right-hand side function
    self.f = Constant(self.mesh, (0.0, 0.0, 0.0))
    n = FacetNormal(self.mesh)

    # Set end-time
    self.T = 1.2* 1.0/self.HR
    self.dt = 0.001
```

Increasing the time step length usually speeds up the calculation of the solution. As long as the CFL number with the maximum velocity v_{max} , time step length dt and minimal edge length h_{min} is smaller than one ($CFL = \frac{v_{max}dt}{h_{min}} < 1$), the solvers should (!!!) converge. For too small time steps it can however lead to an increasing number of iterations for the solver on each time step. As a characterization of the fluid flow, the Reynolds number ($Re = \frac{v_c l}{\nu}$) was calculated with the maximum velocity v_c at the inflow boundary and the characteristic length l of the largest gap between outer and inner boundary. Comparison of Reynolds numbers for different scenarios can be found in Table 27.3.5.

The area of the mesh surfaces and the mesh size can be found as follows.

```
self.h = MeshSize(self.mesh)
self.A0 = self.area(0)
self.A1 = self.area(1)
self.A2 = self.area(2)
```



```
def area(self, i):
    f = Constant(self.mesh, 1)
    A = f*ds(i)
    a = assemble(A, exterior_facet_domains=self.sub_domains)
    return a
```

Object Functions. Being a subclass of `ProblemBase`, `Problem` overrides the object functions `update` and `functional`. The first ensures that all time-dependent variables are updated for the current time step. The latter prints the maximum values for pressure and velocity. The normal flux through the boundaries is defined in the separate function `flux`.

```
def update(self, t, u, p):
    self.g1.t = t
    self.g2.t = t
    pass
def functional(self, t, u, p):
    v_max = u.vector().norm(linf)
    f0 = self.flux(0,u)
    f1 = self.flux(1,u)
    f2 = self.flux(2,u)
    pressure_at_peak_v = p.vector()[0]

    print "time ", t
    print "max value of u ", v_max
    print "max value of p ", p.vector().norm(linf)
    print "CFL = ", v_max * self.dt / self.h.min()
    print "flux through top ", f1
    print "flux through bottom ", f2

    # if current velocity is peak
    if v_max > self.peak_v:
        self.peak_v = v_max
        print pressure_at_peak_v
        self.pressure_at_peak_v = pressure_at_peak_v

    return pressure_at_peak_v

def flux(self, i, u):
    n = FacetNormal(self.mesh)
    A = dot(u,n)*ds(i)
    a = assemble(A, exterior_facet_domains=self.sub_domains)
    return a
```

The boundary conditions are all given as Dirichlet conditions, associated with their velocity function space and the belonging sub domain. The additional functions `boundary_conditions` and `initial_conditions` define the respective conditions for the problem that are called by the solver. Boundary conditions for velocity, pressure and psi (???) are collected in the lists `bcv`, `bcp` and `bcpsi`.

```
def boundary_conditions(self, V, Q):
    # Create no-slip boundary condition for velocity
    self.g0 = Constant(self.mesh, (0.0, 0.0, 0.0))
    bc0 = DirichletBC(V, self.g0, self.contour)

    # create function for inlet and outlet BC
```

```

self.g1 = get_sine_input_function(V, self.z_index, self.HR, self.HR_inv, self.v_max)
self.g2 = self.g1

# Create inflow boundary condition for velocity on side 1 and 2
bc1 = DirichletBC(V, self.g1, self.top)
bc2 = DirichletBC(V, self.g2, self.bottom)

# Collect boundary conditions
bcv = [bc1, bc0, bc2]
bcp = []
bcpsi = []

return bcv, bcp, bcpsi

def initial_conditions(self, V, Q):
    u0 = Constant(self.mesh, (0.0, 0.0, 0.0))
    p0 = Constant(self.mesh, 0.0)

    return u0, p0

```

Running. Applying the "Chorin" solver, the Problem is started by typing :

```
unix>./ns csf_flow chorin.
```

It approximates the Navier–Stokes equation with Chorin’s method. The progress of different simulation steps and results, including maximum calculated pressure and velocity per time step, are printed out on the terminal. In addition, the solution for pressure and velocity are dumped to a file for each (by default?) time step. Before investigating the results, we introduce how the mesh is generated.

27.3.2 Example 1. Simulation of a Pulse in the SAS.

In the first example we represent the spinal cord and the surrounding dura mater as two straight cylinders. These cylinders can easily be generated by using NetGen [?] or Gmsh [?]. In NetGen meshes can be constructed by adding or subtracting geometrical primitives from each other. It also supports DOLFIN mesh generation. Examples for mesh generation with NetGen can be found in

In Gmsh, constructing the basic shapes requires a more complex geometrical description, however it is easier to control how the geometry is meshed. The following code example shows the construction of a circular cylinder (representing the pia on the spinal cord) within an elliptic cylinder (representing the dura). The dura is defined by the ellipse vectors $a=0.65$ mm and $b=0.7$ mm in x and y direction respectively. The cord has a radius of 4 mm with its center moved 0.8 mm in positive x -direction Since Gmsh only allows to draw circular or elliptic arcs for angles smaller than π , the basic ellipses were constructed from four arcs each. Every arc is defined by the starting point, the center, another point on the arc and the end point. The value lc defines the maximal edge length in vicinity to the point.

```
lc = 0.04;
Point(1) = {0,0,0,lc}; // center point
//outer ellipses
a = 0.65;
b = 0.7;
Point(2) = {a,0,0,lc};
Point(3) = {0,b,0,lc};
Point(4) = {-a,0,0,lc};
Point(5) = {0,-b,0,lc};
Ellipse(10) = {2,1,3,3};
Ellipse(11) = {3,1,4,4};
Ellipse(12) = {4,1,5,5};
Ellipse(13) = {5,1,2,2};

// inner ellipses
move = 0.08; //"move" center
Point(101) = {move,0,0,lc};
c = 0.4;
d = 0.4;
Point(6) = {c+move,0,0,lc*0.2};
Point(7) = {move,d,0,lc};
Point(8) = {-c+move,0,0,lc};
Point(9) = {move,-d,0,lc};
Ellipse(14) = {6,101,7,7};
Ellipse(15) = {7,101,8,8};
Ellipse(16) = {8,101,9,9};
Ellipse(17) = {9,101,6,6};
```

The constructed ellipses are composed of separate arcs. To define them as single lines, the ellipse arcs are connected in line loops.

```
// connect lines of outer and inner ellipses to one
Line Loop(20) = {10,11,12,13}; // only outer
Line Loop(21) = {-14,-15,-16,-17}; // only inner
```

The SAS surface between cord and dura is then defined by the following command.

```
Plane Surface(32) = {20,21};
```

To easily construct volumes, Gmsh allows to extrude a generated surface over a given length.

```
length = 5.0
csf[] = Extrude(0,0,length){Surface{32}};
```

Calling the .geo file in Gmsh `>unix Gmsh filename.geo` shows the defined geometry. Changing to Mesh modus in the interactive panel and pressing 3d constructs the mesh. Pressing Save will save the mesh with the .geo-filename and the extension msh. For use in DOLFIN, the mesh generated in Gmsh can be converted by applying the DOLFIN converter.

```
unix>dolfin-convert mesh-name.msh mesh-name.xml
```

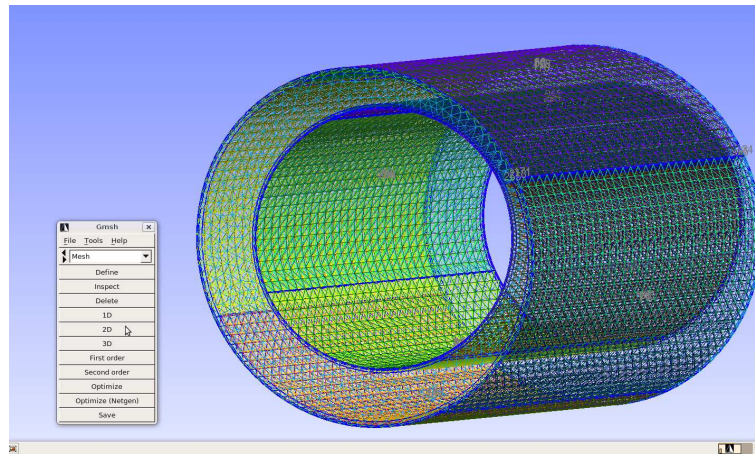


Figure 27.3: Gmsh mesh.

Solver	p in Pa	v_{max} in cm/s	t in s
Chorin	4.03	1.35	0.233
G2	6.70	0.924	0.217
Uzawa			

Table 27.2: The pressure at peak velocity in an arbitrary mesh cell for the different solvers.

Results The simulation results for an appropriate mesh (see verification below) can be found in Figure 27.4. The plots show the velocity component in tubular direction at the mid cross section of the model. The flow profiles are taken at the time steps of maximum flow in both directions and during the transition from systole to diastole. For maximal systole, the velocities have two peak rings, one close to the outer, the other to the inner boundary. We can see sharp profiles at the maxima and bidirectional flow at the local minimum during diastole.

Comparing different solvers.

For the first example, we applied the Chorin solver (WRITE ABOUT MODIFICATIONS WITH TOLERANCES!). For verifying the result, we also applied the solvers G2 and Uzawa. We picked an arbitrary point in the mesh to compare its pressure value at the time step of peak velocity. The results shown in Table 27.2 reveal remarkable differences for ... Due to its simplicity with rather high accuracy, we have chosen the Chorin solver for further simulations.

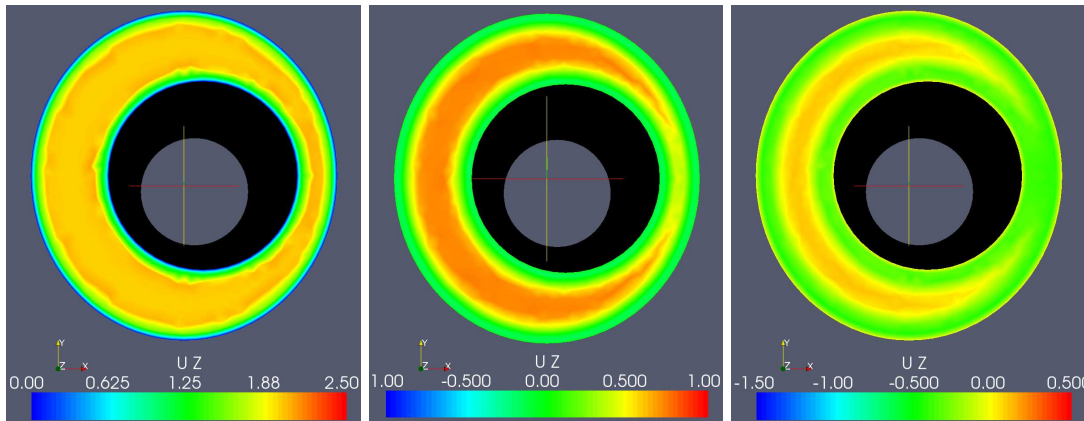


Figure 27.4: Case: Circular cord. The velocity in z -direction for the non-symmetric pulse at the time steps $t = 0.07s, 0.18s, 0.25s$.

Verifying the mesh.

In our case, the resolution in the cross section is more important than along the pipe. Thus, we first varied the number of nodes per cross section on a pipe of length 1.75 cm with 30 layers in the flow direction. Reducing the maximum edge length¹ from 0.04, to 0.02 and 0.01 mm gradually decreased the velocity with some distance to the boundary. The reason for the different velocities lies in the no-slip boundary condition, that influences a greater area for meshes with fewer nodes per cross section, leading to a smaller region of the non-zero flow area.

Additionally, we observed increasingly wave-like structures of fluctuating velocities in the inflow and outflow regions, when the maximum edge length was decreased. These effects result from the changed ratio of edge lengths in cross-sectional and tubular direction.

To avoid increasing the node density utterly, we introduced three thin layers close to the side boundaries, that capture the steep velocity gradient in the boundary layer. The distance of the layers was chosen, so that the edge length slightly increases for each layer. Starting from 10% of the maximum edge length, for the first layer, the width of the second and the third layer was set to 30% and 80% of the maximum edge length. It turned out, that for meshes with layers along the side boundaries, a maximum edge length of 0.04 mm was enough to reproduce the actual flow profile.

To add mesh layers in Gmsh, copies for the elliptic arcs are scaled to gradually increase the maximum edge length. The code example below shows the creation of the layers close to the outer ellipse. The inner layers are created similarly.

¹Denoted as `lc` in the Gmsh code.

```
outer_b1[] = Dilate {{0, 0, 0}, 1.0 - 0.1*lc } {  
  Duplicata{ Line{10}; Line{11}; Line{12}; Line{13}; } };  
outer_b2[] = Dilate {{0, 0, 0}, 1.0 - 0.3*lc } {  
  Duplicata{ Line{10}; Line{11}; Line{12}; Line{13}; } };  
outer_b3[] = Dilate {{0, 0, 0}, 1.0 - 0.8*lc } {  
  Duplicata{ Line{10}; Line{11}; Line{12}; Line{13}; } };
```

The single arcs are dilated separately since the arc points are necessary for further treatment. Remember that no arcs with angles smaller than π are allowed. Again we need a representation for the complete ellipses defined by line loops, as

```
Line Loop(22) = {outer_b1[]};
```

that are necessary to define the surfaces between all neighboring ellipses similar to:

```
Plane Surface(32) = {20,22};
```

Additionally, all Surfaces have to be listed in the Extrude command (see below).

The tubular layers can be specified during extrusion. Note that the list of extruded surfaces now contains the six layers close to the side boundaries and the section between them.

```
// Extrude  
length = 5.0;  
layers = 30;  
csf[] = Extrude {0,0,length} {Surface{32}; Surface{33};  
  Surface{34};Surface{35};Surface{36};Surface{37};Surface{38};Layers{ {layers}, {1} }; };
```

Besides controlling the numbers of nodes in tubular direction, extruded meshes result in more homogenous images in cross-sectional cuts.

The test meshes of 1.75 cm showed seemed to have a fully developed region around the mid-cross sections, where want to observe the flow profile. Testing different numbers of tubular layers for the length of 1.75, 2.5 and 5 cm showed that the above mentioned observations of wave-like structures occurred less for longer pipes, even though the number of layers was low compared to the pipe length. The presented results were simulated on meshes of length 5 cm with 30 layers in z-direction and three layers on the side boundaries. The complete code can be found in `mesh_generation/FILENAME`.

27.3.3 Example 2. Simplified Boundary Conditions.

Many researchers apply the sine function as inlet and outlet boundary condition, since its integral is zero over one period. However, its shape is not in agreement

with measurements of the cardiac flow pulse (Loth et al. [?]). To see the influence of the applied boundary condition for the defined mesh, we replace the more realistic pulse function with a sine, scaled to the same amount of volume transport per cardiac cycle. The code example below implements the alternative pulse function in the object function `boundary_conditions`. The variable `sin_integration_factor` describes the integral of the first half of a sine.

```
self.HR = 1.16 # heart rate in beats per second; from 70 bpm
self.HR_inv = 1.0/self.HR
self.SV = 0.27
self.A1 = self.area(1)
self.flow_per_unit_area = self.volume_flow/self.A1
sin_integration_factor = 0.315
self.v_max = self.flow_per_unit_area/sin_integration_factor
```

As before, we have a global function returning the sine as a Function - object,

```
def get_sine_input_function(V, z_index, HR, HR_inv, v_max):
    v_z = "sin(2*pi*HR*fmod(t,T))*(v_max)"
    vel = ["0.0", "0.0", "0.0"]
    vel[z_index] = v_z
    class Sine(Function):
        cpparg = tuple(vel)
        defaults = {'HR':HR, \emp{v_max}:v_max, \emp{T}:HR_inv}
    return Sine(V)
```

that is called instead of `get_pulse_input_function` in the function named `boundary_conditions`:

```
self.gl = get_sine_input_function(V, self.z_index, self.factor, self.A, self.HR_inv, self.HR, \
                                self.b, self.fl).
```

The pulse and the sine are sketched in Figure 27.2. Both functions are marked at the points of interest: maximum systolic flow, around the transition from systole to diastole and the (first, local) minimum. Results for sinusoidal boundary conditions are shown in Figure 27.5 The shape of the flow profile is similar in every time step, only the magnitudes change. No bidirectional flow was discovered in the transition from positive to negative flow. Compared to the results received by the more realistic pulse function, the velocity profile generated from sinusoidal boundaries is more homogeneous over the cross section.

27.3.4 Example 3. Cord Shape and Position.

According to [?], [?], the present flow is inertia dominated, meaning that the shape of the cross section should not influence the pressure gradient. Changing the length of vectors describing the ellipse from

```
c = 0.4;
d = 0.4;
```

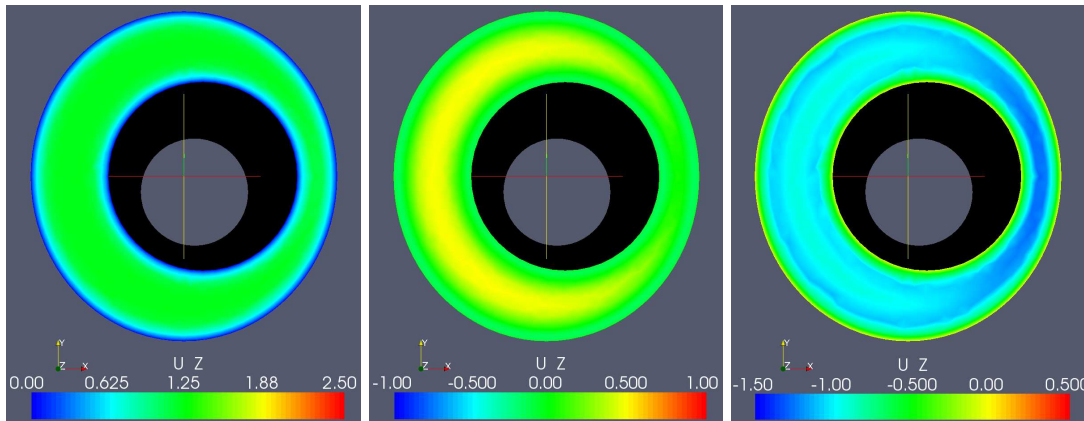



Figure 27.5: Case: Circular Cord. The velocity in z-direction as response to a sine boundary condition for the time steps $t = 0.2, 0.4, 0.6$.

to

```
c = 0.32;
d = 0.5;
```

transforms the cross section of the inner cylinder to an elliptic shape with preserved area. The simulation results are collected in Figure 27.6. Comparisons showed that the pressure gradient was identical for the two cases, the different shape is however reflected in the flow profiles.

A further perturbation of the SAS cross sections was achieved by changing the moving of the center of the elliptic cord from

```
move = 0.08;
```

to

```
move = 0.16;
```

Also for this case the pressure field was identical, with some variations in the flow profiles.

27.3.5 Example 4. Cord with Syrinx.

Syrinxes expand the cord so that it occupies more space of the spinal SAS. Increasing the cord radius from 4 mm to 5 mm² decreases the cross sectional area by almost one third to 0.64 cm². The resulting flow is shown in Figure 27.8. Apart from the increased velocities, we see bidirectional flow already at $t = 0.18$ and at

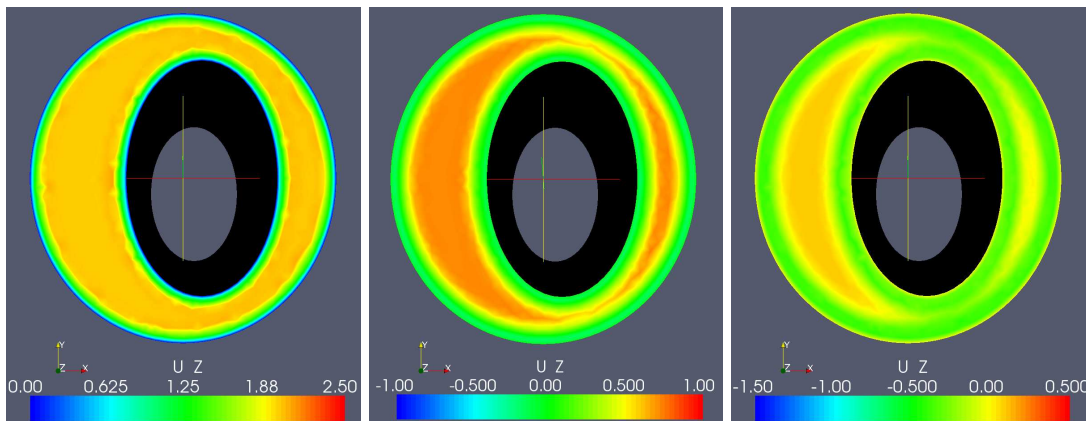


Figure 27.6: Case: Elliptic cord. The velocity in z-direction for the non-symmetric pulse at the time steps $t = 0.07s, 0.18s, 0.25s$.

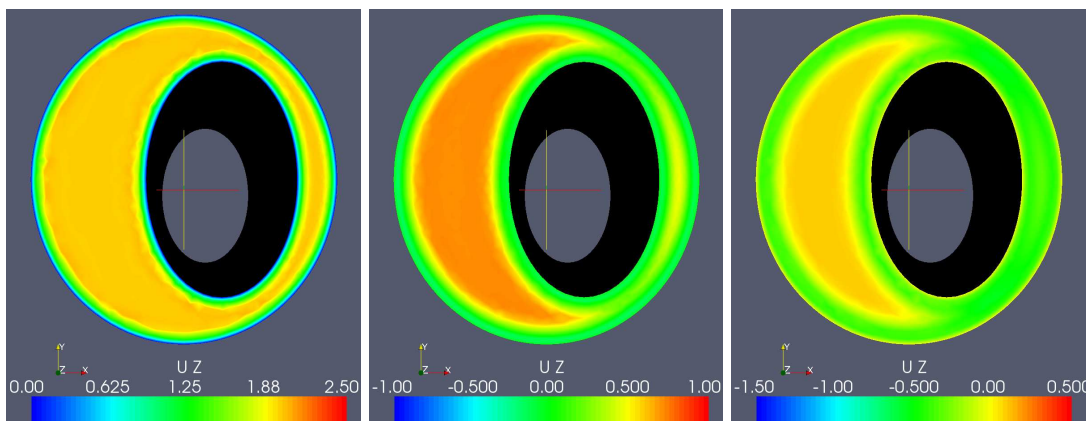


Figure 27.7: Case: Translated elliptic cord. The velocity in z-direction for the non-symmetric pulse at the time steps $t = 0.07s, 0.18s, 0.25s$.

Problem	D^3 in cm	v_{max}^4 in cm/s	Re	We
Example 1	0.54	2.3	177	17
Example 2	0.54	0.92	70	17
Example 4	0.45	3.2	205	14

Table 27.3: Characteristic values for the examples 1, 2 and 3.

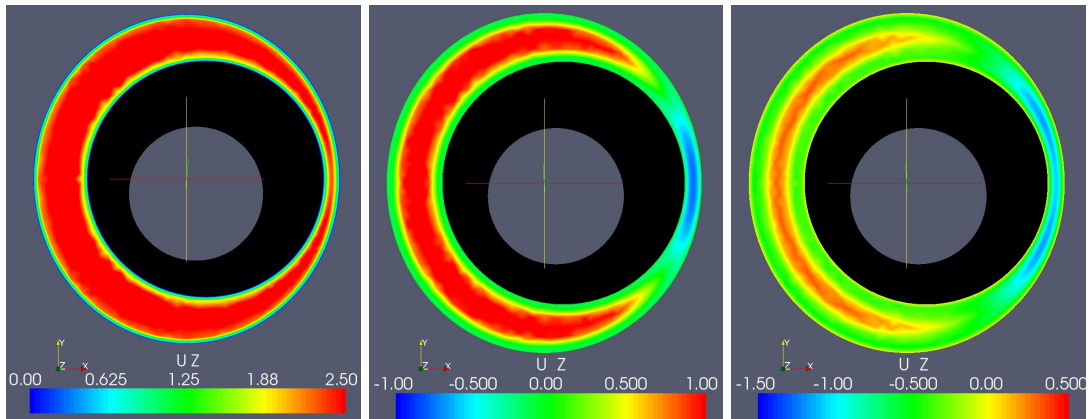


Figure 27.8: Case: Enlarged cord diameter. The velocity in z-direction for the non-symmetric pulse at the time steps $t = 0.07s, 0.18s, 0.25s$.

$t=0.25$ as before. The fact that diastolic back flow is visible at $t = 0.18$, shows that the pulse with its increased amplitude travels faster.

Comparing Reynolds and Womersly numbers shows a clear difference for the above described examples 1, 2 and 3. Example 2 is marked by a clearly lower maximum velocity at inflow and outflow boundary that leads to a rather low Reynolds number. Due to the different inflow and outflow area, Example 4 has a lower Womersley number, leading to an elevated maximum velocity at the boundary and clearly increased Reynolds number. These numbers help to quantify the changes introduced by variations in the model. For the chosen model, the shape of the pulse function at the boundary condition as well as the cross sectional area have great influence on the simulation results. As earlier shown by Loth et al. [?], altering the shape of the cross sections does not seem to influence the flow greatly.

²which equals to set the variables c and d in the geo-file to 0.5

Turbulent Flow and Fluid–Structure Interaction with Unicorn

By Johan Hoffman, Johan Jansson, Niclas Jansson, Claes Johnson and Murtazo Nazarov

Chapter ref: [**hoffman-1**]

28.1 Introduction

For many problems involving a fluid and a structure, decoupling the computation of the two is not possible for accurate modeling of the phenomenon at hand, instead the full fluid-structure interaction (FSI) problem has to be solved together as a coupled problem. This includes a multitude of important problems in biology, medicine and industry, such as the simulation of insect or bird flight, the human cardiovascular and respiratory systems, the human speech organ, the paper making process, acoustic noise generation in exhaust systems, airplane wing flutter, wind induced vibrations in bridges and wave loads on offshore structures. Common for many of these problems is that for various reasons they are very hard or impossible to investigate experimentally, and thus reliable computational simulation would open up for detailed study and new insights, as well as for new important design tools for construction.

Computational methods used today are characterized by a high computational cost, and a lack of generality and reliability. In particular, major open challenges of computational FSI include: (i) robustness of the fluid-structure coupling, (ii) for high Reynolds numbers the computation of turbulent fluid flow,

and (iii) efficiency and reliability of the computations in the form of adaptive methods and quantitative error estimation.

The FEniCS project aims towards the goals of generality, efficiency, and simplicity, concerning mathematical methodology, implementation, and application. The Unicorn project is a realization of this effort in the field of continuum mechanics, that we here expose to a range of challenging problems that traditionally demand a number of specialized methods and codes, including some problems which have been considered unsolvable with state of the art methods. The basis of Unicorn is an adaptive finite element method and a unified continuum formulation, which offer new possibilities for mathematical modeling of high Reynolds number turbulent flow, gas dynamics and fluid-structure interaction.

Unicorn, which is based on the DOLFIN/FFC/FIAT suite, is today central in a number of applied research projects, characterized by large problems, complex geometry and constitutive models, and a need for fast results with quantitative error control. We here present some key elements of Unicorn and the underlying theory, and illustrate how this opens for a number of breakthroughs in applied research.

28.2 Continuum models

Continuum mechanics is based on conservation laws for mass, momentum and energy, together with constitutive laws for the stresses. A Newtonian fluid is characterized by a linear relation between the viscous stress and the strain, together with a fluid pressure, resulting in the Navier-Stokes equations. Many common fluids, including water and air at subsonic velocities, can be modeled as incompressible fluids, where the pressure acts as a Lagrangian multiplier enforcing a divergence free velocity. In models of gas dynamics the pressure is given from the internal energy, with an ideal gas corresponding to a linear relation. Solids and non-Newtonian fluids can be described by arbitrary complex laws relating the stress to displacements, strains and internal energies.

Newtonian fluids are characterized by two important non-dimensional numbers: the Reynolds number Re , measuring the importance of viscous effects, and the Mach number M , measuring the compressibility effects by relating the fluid velocity to the speed of sound. High Re flow is characterized by partly turbulent flow, and high M flow by shocks and contact discontinuities, all phenomena associated with complex flow on a range of scales. The Euler equations corresponds to the limit of inviscid flow where $Re \rightarrow \infty$, and incompressible flow corresponds to the limit of $M \rightarrow 0$.

28.3 Mathematical framework

The mathematical theory for the Navier-Stokes equations is incomplete, without any proof of existence or uniqueness, formulated as one of the Clay Institute \$1 million problems. What is available is the proof of existence of weak solutions by Leray from 1934, but this proof does not extend to the inviscid case of the Euler equations. No general uniqueness result is available for weak solutions, which limits the usefulness of the concept.

In [1] a new mathematical framework of weak (output) uniqueness is introduced, characterizing well-posedness of weak solutions with respect to functionals M of the solution u , thereby circumventing the problem of non-existence of classical solutions. This framework extends to the Euler equations, and also to compressible flow, where the basic result takes the form

$$|M(u) - M(U)| \leq S(\|R(u)\|_{-1} + \|R(U)\|_{-1}) \quad (28.1)$$

with $\|\cdot\|_{-1}$ a weak norm measuring residuals R of two weak solutions u and U , and with S a stability factor given by a duality argument connecting local errors to output errors in M .

28.4 Computational method

Computational methods in fluid mechanics are typically very specialized; for a certain range of Re or M , or for a particular class of geometries. In particular, there is a wide range of turbulence models and shock capturing techniques. The goal of Unicorn is to design one method with one implementation, capable of modeling general geometry and the whole range of parameters Re and M . The mathematical framework of well-posedness allows for a general foundation for Newtonian fluid mechanics, and the General Galerkin (G2) finite element method offers a robust algorithm to compute weak solutions [?].

Adaptive G2 methods are based on a posteriori error estimates of the form:

$$|M(u) - M(U)| \leq \sum_K \mathcal{E}_K \quad (28.2)$$

with \mathcal{E}_K a local error indicator for cell K .

Parallel mesh refinement...

[Unicorn implementation]

28.5 Boundary conditions

1/2 page

friction bc, turbulent boundary conditions

[Unicorn implementation]

28.6 Geometry modeling

1/2 page

projection to exact geometry
[Unicorn implementation]

28.7 Fluid-structure interaction

1 page

Challenges: stability coupling, weak strong, monolithic
different software, methods
mesh algorithms, smoothing, Madlib
Unified continuum fluid-structure interaction
[Unicorn implementation]

28.8 Applications

28.8.1 *Turbulent flow separation*

1 pages

drag crisis, cylinders and spheres, etc.

28.8.2 *Flight aerodynamics*

2 page

naca aoa

28.8.3 *Vehicle aerodynamics*

1 page

Volvo

28.8.4 *Biomedical flow*

1 page

ALE heart model

28.8.5 *Aeroacoustics*

1 page

Swenox mixer - aerodynamic sources

28.8.6 *Gas flow*

2 pages

compressible flow

28.9 References

1 page

Fluid–Structure Interaction using Nitsche’s Method

By Kristoffer Selim and Anders Logg

Chapter ref: [**selim**]

In this study, we present a 2D fluid–structure interaction (FSI) simulation of a channel flow containing an elastic valve that may undergo large deformations. The FSI occurs when the fluid interacts with the solid structure of the valve, exerting pressure that causes deformation of the valve and, thus, alters the flow of the fluid itself. To describe the equations governing the fluid flow and the elastic valve, two separate meshes are used and Nitsche’s method is used to couple the equations on the two meshes. The method is based on continuous piecewise approximations on each mesh with weak enforcement of the proper continuity at the interface defined by the boundary of one of the overlapping meshes.

Improved Boussinesq Equations for Surface Water Waves

By N. Lopes, P. Pereira and L. Trabucho

Chapter ref: **[lopes]**

► Editor note: *Move macros to common macros after deciding what to do about bold fonts.*

► Editor note: *List authors with full names*

The main motivation of this work is the implementation of a general solver for some of the improved Boussinesq models. Here, we use the second order model proposed by Zhao et al. [?] to investigate the behaviour of surface water waves. Some effects like surface tension, dissipation and wave generation by natural phenomena or external physical mechanisms are also included. As a consequence, some modified dispersion relations are derived for this extended model.

30.1 Introduction

The FEniCS project, via DOLFIN and FFC, provides a good support for the implementation of large scale industrial models. We implement a solver for some of the Boussinesq type systems to model the evolution of surface water waves in a variable depth seabed. This type of models is used, for instance, in harbour simulation¹, tsunami generation and propagation as well as in coastal dynamics.

¹See Fig. 30.1 for an example of a standard harbour.

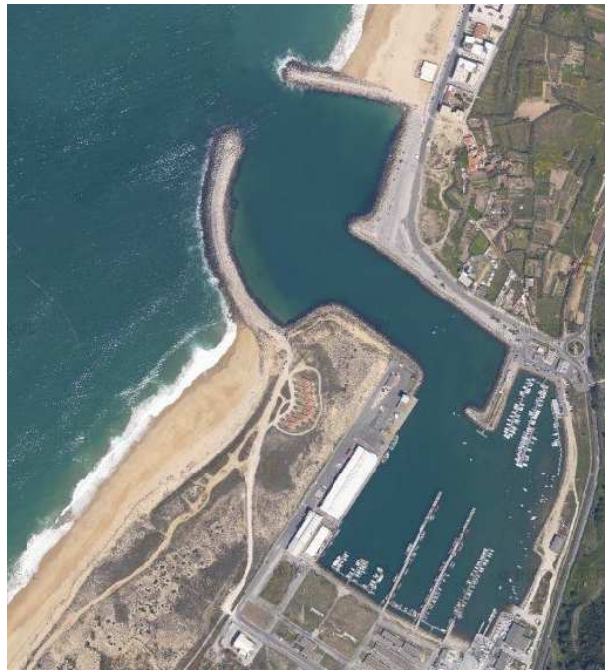


Figure 30.1: Nazaré's harbour, Portugal.

► *Editor note: Need to get permission for this figure!*

There are several **B**oussinesq models and some of the most widely used are those based on the wave **E**levation and horizontal **V**elocities formulation (BEV) (see, e.g., [?], [?]).

In the next section the governing equations for surface water waves are presented. From these equations different types of models can be derived. We consider only the wave **E**levation and velocity **P**otential (BEP) formulation. Thus, the number of system equations is reduced when compared to the BEV models. Two different types of BEP models are taken into account:

- i*) a standard sixth-order model;
- ii*) the second-order model proposed by **Z**hao, **T**eng and **C**heng (ZTC) (cf. [?]).

We use the sixth-order model to illustrate a standard technique in order to derive a Boussinesq-type model. In the subsequent sections, only the ZTC model is considered. Note that these two models are complemented with some extra terms, due to the inclusion of effects like dissipation, surface tension and wave generation by moving an impermeable bottom or using a source function.

An important characteristic of the modified ZTC model, including dissipative effects, is presented in the third section, namely, the dispersion relation.

In the fourth and fifth sections, we describe several types of wave generation, absorption and reflection mechanisms. Initial conditions for a solitary wave and

a periodic wave induced by Dirichlet boundary conditions are also presented. Moreover, we complement the ZTC model using a source function to generate surface water waves, as proposed in [?]. Total reflective walls are modelled by standard zero Neumann conditions for the surface elevation and velocity potential. The wave energy absorption is simulated using sponge layers.

The following section is dedicated to the numerical methods used in the discretization of the variational formulation. The discretization of the spatial variables is accomplished with low order Lagrange elements whereas the time integration is implemented using Runge-Kutta and Predictor-Corrector algorithms.

In the seventh section, the ZTC numerical model is used to simulate the evolution of a periodic wave in an harbour geometry like that one represented in Fig. 30.1.

30.2 Model derivation

As usual we consider the following set of equations for the irrotational flow of an incompressible and inviscid fluid:

$$\begin{cases} \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla_{xyz}) \mathbf{u} = -\nabla_{xyz} \left(\frac{P}{\rho} + g z \right), \\ \nabla_{xyz} \times \mathbf{u} = \mathbf{0}, \\ \nabla_{xyz} \cdot \mathbf{u} = 0, \end{cases} \quad (30.1)$$

where \mathbf{u} is the velocity vector field of the fluid, P the pressure, g the gravitational acceleration, ρ the mass per unit volume, t the time and the differential operator $\nabla_{xyz} = \left[\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right]$. A Cartesian coordinate system is adopted with the horizontal x and y -axes on the still water plane and the z -axis pointing vertically upwards (see Fig. 30.2). The fluid domain is bounded by the bottom seabed at $z = -h(x, y, t)$ and the free water surface at $z = \eta(x, y, t)$.

► **Editor note:** *AL: Need to decide what to do about bold fonts for vectors. I prefer not to use it.*

In Fig. 30.2, L , A and H are the characteristic wave length, wave amplitude and depth, respectively. Note that the material time derivative is denoted by $\frac{D}{Dt}$.

From the irrotational assumption, one can introduce a velocity potential function, $\phi(x, y, z, t)$, to obtain Bernoulli's equation:

$$\frac{\partial \phi}{\partial t} + \frac{1}{2} \nabla_{xyz} \phi \cdot \nabla_{xyz} \phi + \frac{P}{\rho} + g z = f(t), \quad (30.2)$$

where $f(t)$ stands for an arbitrary function of integration. Note that one can remove $f(t)$ from equation (30.2) if ϕ is redefined by $\phi + \int f(t) dt$. From the incompressibility condition (see (30.1)₃) the velocity potential satisfies Laplace's

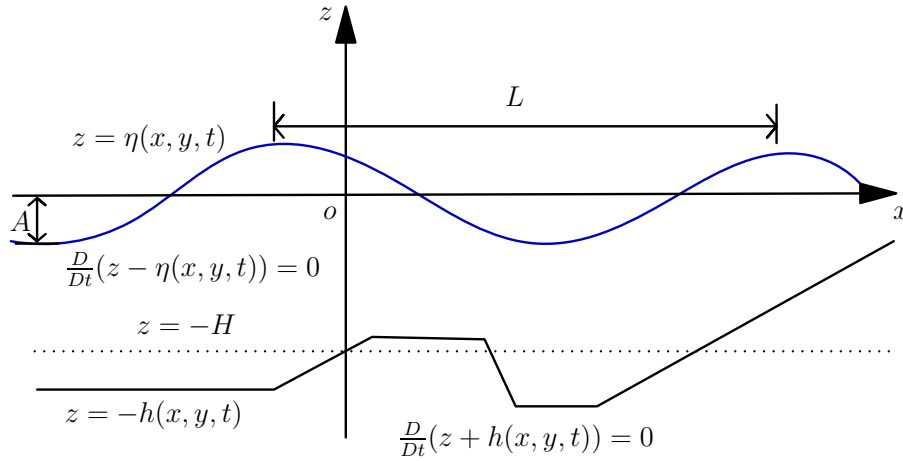


Figure 30.2: Cross-section of the water wave domain.

equation:

$$\nabla^2 \phi + \frac{\partial^2 \phi}{\partial z^2} = 0, \quad (30.3)$$

where ∇ is the horizontal gradient operator given by $\nabla = \left[\frac{\partial}{\partial x}, \frac{\partial}{\partial y} \right]$. To close this problem, the following boundary conditions must be satisfied:

i) the kinematic boundary condition for the free water surface:

$$\frac{\partial \phi}{\partial z} = \frac{\partial \eta}{\partial t} + \nabla \phi \cdot \nabla \eta, \quad z = \eta; \quad (30.4)$$

ii) the kinematic boundary condition for the impermeable sea bottom:

$$\frac{\partial \phi}{\partial z} + (\nabla \phi \cdot \nabla h) = -\frac{\partial h}{\partial t}, \quad z = -h; \quad (30.5)$$

iii) the dynamic boundary condition for the free water surface:

$$\frac{\partial \phi}{\partial t} + g\eta + \frac{1}{2} \left(|\nabla \phi|^2 + \left(\frac{\partial \phi}{\partial z} \right)^2 \right) + D(\phi) - W(\eta) = 0, \quad z = \eta, \quad (30.6)$$

where $D(\phi)$ is a dissipative term (see, e.g., the work by Duthy and Dias [?]). We assume that this dissipative term is of the following form:

$$D(\phi) = \nu \frac{\partial^2 \phi}{\partial z^2}, \quad (30.7)$$

with $\nu = \mu/\rho$ and μ an eddy-viscosity coefficient. Note that a non-dissipative model means that there is no energy loss. This is not acceptable from a physical point of view, since any real flow is accompanied by energy dissipation.

In equation (30.6), W is the surface tension term given by:

$$W(\eta) = T \frac{\left(1 + \left(\frac{\partial \eta}{\partial y}\right)^2\right) \frac{\partial^2 \eta}{\partial x^2} + \left(1 + \left(\frac{\partial \eta}{\partial x}\right)^2\right) \frac{\partial^2 \eta}{\partial y^2} - 2 \frac{\partial \eta}{\partial x} \frac{\partial \eta}{\partial y} \frac{\partial^2 \eta}{\partial x \partial y}}{(1 + |\nabla \eta|^2)^{3/2}}, \quad (30.8)$$

where T is the surface tension coefficient.

Using Laplace's equation it is possible to write the dissipative term, mentioned above, as $D(\phi) = -\nu \nabla^2 \phi$. In addition, the linearization of the surface tension term results in $W(\eta) = T \nabla^2 \eta$. Throughout the literature, analogous terms were added to the kinematic and dynamic conditions to absorb the wave energy near the boundaries. These terms are related with the sponge or damping layers. The physical meaning of these terms was explained, for instance, in the recent work of Dias et al. [?]. As we will see later, the dispersion relations can be modified using these extra terms. The surface tension effects are important if short waves are considered. In general this is not the case. In fact, the long wave assumption is made to derive these extended models. We refer the works by Wang et al. [?] as well as Dash and Daripa [?], which included surface tension effects in the KdV (Korteweg-de Vries) and Boussinesq equations. On the other hand, it is worth to mention that one of the main goals of the scientific research on Boussinesq wave models is the improvement of the range of applicability in terms of the water-depth/wave-length relationship. A more detailed description of the above equations is found in the G. B. Whitham's reference book on waves [?], or in the more recent book by R. S. Johnson [?].

30.2.1 Standard models

In this subsection, we present a generic Boussinesq system using the velocity potential formulation. To transform equations (30.2)-(30.8) in a dimensionless form, the following scales are introduced:

$$(x', y') = \frac{1}{L}(x, y), \quad z' = \frac{z}{H}, \quad t' = \frac{t\sqrt{gH}}{L}, \quad \eta' = \frac{\eta}{A}, \quad \phi' = \frac{H\phi}{AL\sqrt{gH}}, \quad h' = \frac{h}{H}, \quad (30.9)$$

together with the small parameters

$$\mu = \frac{H}{L}, \quad \varepsilon = \frac{A}{H}. \quad (30.10)$$

In the last equation, μ is usually called the long wave parameter and ε the small amplitude wave parameter. Note that ε is related with the nonlinear terms and μ with the dispersive terms. For simplicity, in what follows, we drop the prime notation.

The Boussinesq approach consists in reducing a **3D** problem to a **2D** one. This may be accomplished by expanding the velocity potential in a Taylor power series

in terms of z . Using Laplace's equation, in a dimensionless form, one can obtain the following expression for the velocity potential:

$$\phi(x, y, z, t) = \sum_{n=0}^{+\infty} \left((-1)^n \frac{z^{2n}}{(2n)!} \mu^{2n} \nabla^{2n} \phi_0(x, y, t) + (-1)^n \frac{z^{2n+1}}{(2n+1)!} \mu^{2n} \nabla^{2n} \phi_1(x, y, t) \right), \quad (30.11)$$

with

$$\phi_0 = \phi|_{z=0}, \quad \phi_1 = \left(\frac{\partial \phi}{\partial z} \right) |_{z=0}. \quad (30.12)$$

From asymptotic expansions, successive approximation techniques and the kinematic boundary condition for the sea bottom, it is possible to write ϕ_1 in terms of ϕ_0 (cf. [?], [?]). In this work, without loss of generality, we assume that the dispersive and nonlinear terms are related by the following equation:

$$\frac{\varepsilon}{\mu^2} = O(1). \quad (30.13)$$

Note that the Ursell number is defined by $U_r = \frac{\varepsilon}{\mu^2}$.

A sixth-order model is obtained if ϕ_1 is expanded in terms of ϕ_0 and all terms up to $O(\mu^8)$ are retained. Thus, the asymptotic kinematic and dynamic boundary conditions for the free water surface are rewritten as follows ²:

$$\left\{ \begin{array}{l} \frac{\partial \eta}{\partial t} + \varepsilon \nabla \cdot (\eta \nabla \phi_0) - \frac{1}{\mu^2} \phi_1 + \frac{\varepsilon^2}{2} \nabla \cdot (\eta^2 \nabla \phi_1) = O(\mu^6), \\ \frac{\partial \phi_0}{\partial t} + \varepsilon \eta \frac{\partial \phi_1}{\partial t} + \eta + \frac{\varepsilon}{2} |\nabla \phi_0|^2 + \varepsilon^2 \nabla \phi_0 \cdot \eta \nabla \phi_1 - \\ \quad - \varepsilon^2 \eta \nabla^2 \phi_0 \phi_1 + \frac{\varepsilon}{2\mu^2} \phi_1^2 + D(\phi_0, \phi_1) - W(\eta) = O(\mu^6), \end{array} \right. \quad (30.14)$$

where ϕ_1 is given by:

$$\begin{aligned} \phi_1 = & -\mu^2 \nabla \cdot (h \nabla \phi_0) + \frac{\mu^4}{6} \nabla \cdot (h^3 \nabla^3 \phi_0) - \frac{\mu^4}{2} \nabla \cdot (h^2 \nabla^2 \cdot (h \nabla \phi_0)) - \\ & - \frac{\mu^6}{120} \nabla \cdot (h^5 \nabla^5 \phi_0) + \frac{\mu^6}{24} \nabla \cdot (h^4 \nabla^4 \cdot (h \nabla \phi_0)) + \frac{\mu^6}{12} \nabla \cdot (h^2 \nabla^2 \cdot (h^3 \nabla^3 \phi_0)) - \\ & - \frac{\mu^6}{4} \nabla \cdot (h^2 \nabla^2 \cdot (h^2 \nabla^2 \cdot (h \nabla \phi_0))) - \frac{\mu^2}{\varepsilon} \frac{\partial h}{\partial t} - \frac{\mu^2}{\varepsilon} \frac{\mu^2}{2} \nabla \cdot \left(h^2 \nabla \frac{\partial h}{\partial t} \right) + \\ & + \frac{\mu^2}{\varepsilon} \frac{\mu^4}{24} \nabla \cdot \left(h^4 \nabla^3 \frac{\partial h}{\partial t} \right) - \frac{\mu^2}{\varepsilon} \frac{\mu^4}{4} \nabla \cdot \left(h^2 \nabla^2 \left(h^2 \nabla \frac{\partial h}{\partial t} \right) \right) + O(\mu^8). \end{aligned} \quad (30.15)$$

To obtain equation (30.15), we assume that $\frac{\partial h}{\partial t} = O(\varepsilon)$ (cf. [?]).

²Note that D and W are, now, dimensionless functions.

30.2.2 Second-order model

The low order equations are obtained, essentially, via the slowly varying bottom assumption. In particular, only $O(h, \nabla h)$ terms are retained. Also, only low order nonlinear terms $O(\varepsilon)$ are admitted. In fact the modified ZTC model is written retaining only $O(\varepsilon, \mu^4)$ terms.

Under these conditions, (30.14) and (30.15) lead to:

$$\begin{cases} \frac{\partial \eta}{\partial t} + \varepsilon \nabla \cdot (\eta \nabla \phi_0) - \frac{1}{\mu^2} \phi_1 = O(\mu^6), \\ \frac{\partial \phi_0}{\partial t} + \eta + \frac{\varepsilon}{2} |\nabla \phi_0|^2 - \nu \varepsilon \nabla^2 \phi_0 - \mu^2 T \nabla^2 \eta = O(\mu^6) \end{cases} \quad (30.16)$$

and

$$\begin{aligned} \phi_1 = & -\mu^2 \nabla \cdot (h \nabla \phi_0) + \frac{\mu^4}{6} \nabla \cdot (h^3 \nabla^3 \phi_0) - \frac{\mu^4}{2} \nabla \cdot (h^2 \nabla^2 \cdot (h \nabla \phi_0)) - \\ & - \frac{2\mu^6}{15} h^5 \nabla^6 \phi_0 - 2\mu^6 h^4 \nabla h \cdot \nabla^5 \phi_0 - \frac{\mu^2}{\varepsilon} \frac{\partial h}{\partial t} + O(\mu^8). \end{aligned} \quad (30.17)$$

Thus, these extended equations, in terms of the dimensional variables, are written as follows:

$$\begin{cases} \left[\frac{\partial \eta}{\partial t} + \nabla \cdot [(h + \eta) \nabla \Phi] - \frac{1}{2} \nabla \cdot [h^2 \nabla \frac{\partial \eta}{\partial t}] + \frac{1}{6} h^2 \nabla^2 \frac{\partial \eta}{\partial t} - \frac{1}{15} \nabla \cdot [h \nabla (h \frac{\partial \eta}{\partial t})] \right] = -\frac{\partial h}{\partial t}, \\ \left[\frac{\partial \Phi}{\partial t} + \frac{1}{2} |\nabla \Phi|^2 + g\eta - \frac{1}{15} g h \nabla \cdot (h \nabla \eta) - \nu \nabla^2 \Phi - g T \nabla^2 \eta \right] = 0, \end{cases} \quad (30.18)$$

where Φ is the transformed velocity potential given by:

$$\Phi = \phi_0 + \frac{h}{15} \nabla \cdot (h \nabla \phi_0). \quad (30.19)$$

The transformed velocity potential is used with two main consequences (cf. [?]):

- i) the spatial derivation order is reduced to the second order;
- ii) linear dispersion characteristics, analogous to the fourth-order BEP model proposed by Liu and Woo [?] and the third-order BEV model developed by Nwogu [?], are obtained.

30.3 Linear dispersion relation

One of the most important properties of a water wave model is described by the linear dispersion relation. From this relation we can deduce the phase velocity,

group velocity and the linear shoaling. The dispersion relation provides a good method to characterize the linear properties of a wave model. This is achieved using the linear wave theory of Airy.

In this section we follow the work by Duthyk and Dias [?]. Moreover, we present a generalized version of the dispersion relation for the ZTC model with the dissipative term mentioned above. One can also include other damping terms, which are usually used in the sponge layers.

For simplicity, a 1D-Horizontal model is considered. To obtain the dispersion relation, a standard test wave is assumed:

$$\begin{cases} \eta(x, t) = a e^{i(kx - \omega t)}, \\ \Phi(x, t) = -b i e^{i(kx - \omega t)}, \end{cases} \quad (30.20)$$

where a is the wave amplitude, b the potential magnitude, $k = 2\pi/L$ the wave number and ω the angular frequency. This wave, described by equations (30.20), is the solution of the linearized ZTC model, with a constant depth bottom and an extra dissipative term, if the following equation is satisfied:

$$\omega^2 - ghk^2 \frac{1 + (1/15)(kh)^2}{1 + 2/5(kh)^2} + i\nu\omega k^2 = 0. \quad (30.21)$$

Using Padé's [2,2] approximant, the dispersion relation given by last equation is accurate up to $O((kh)^4)$ or $O(\mu^4)$ when compared with the following equation:

$$\omega^2 - ghk^2 \frac{\tanh(kh)}{kh} + i\nu\omega k^2 = 0. \quad (30.22)$$

In fact, equation (30.22) is the dispersion relation of the full linear problem.

From (30.21), the phase velocity, $C = \frac{\omega}{k}$, for this dissipative ZTC model is given by:

$$C = -\frac{i\nu k}{2} \pm \sqrt{-\left(\frac{\nu k}{2}\right)^2 + gh \frac{(1 + 1/15(kh)^2)}{(1 + 2/5(kh)^2)}}. \quad (30.23)$$

In Fig. 30.3, we can see the positive real part of $\left(C/\sqrt{gh}\right)$ as a function of kh for the following models: full linear theory (FL), Zhao et al. (ZTC), full linear theory with a dissipative model (FL_D) and the improved ZTC model with the dissipative term (ZTC_D). From Fig. 30.3, one can also see that these two dissipative models admit critical wave numbers k_1 and k_2 , such that the positive part of $Re\left(C/\sqrt{gh}\right)$ is zero for $k \geq k_1$ and $k \geq k_2$. To avoid some numerical instabilities one can optimize the ν values in order to reduce the short waves propagation.

In general, to improve the dispersion relation one can also use other transformations like (30.19), or evaluate the velocity potential at $z = -\sigma h$ ($\sigma \in [0, 1]$) instead of $z = 0$ (cf. [?], [?] and [?]).

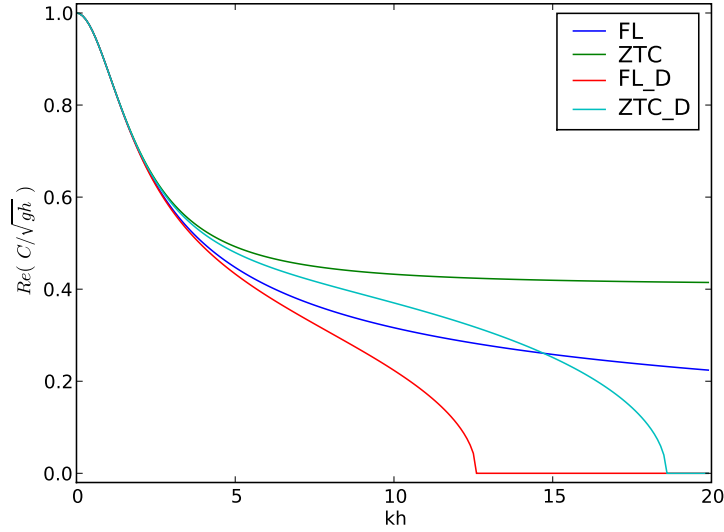


Figure 30.3: Positive part of $Re\left(C/\sqrt{gh}\right)$ for several models.

30.4 Wave generation

In this section some of the physical mechanisms to induce surface water waves are presented. We note that the moving bottom approach is useful for wave generation due to seismic activities. However, some physical applications are associated with other wave generation mechanisms. For simplicity, we only consider mechanisms to generate surface water waves along the x direction.

30.4.1 Initial condition

The simplest way of inducing a wave into a certain domain is to consider an appropriate initial condition. An useful and typical benchmark case is to assume a solitary wave given by:

$$\eta(x, t) = a_1 \operatorname{sech}^2(kx - \omega t) + a_2 \operatorname{sech}^4(kx - \omega t), \quad (30.24)$$

$$u(x, t) = a_3 \operatorname{sech}^2(kx - \omega t), \quad (30.25)$$

where the parameters a_1 and a_2 are the wave amplitudes and a_3 is the magnitude of the velocity in the x direction. As we use a potential formulation, Φ is given by:

$$\Phi(x, t) = -\frac{2a_3 e^{2\omega t}}{k(e^{2\omega t} + e^{2kx})} + K_1(t), \quad (30.26)$$

where $K_1(t)$ is a time-dependent function of integration.

In [?] and [?] the above solitary wave was presented as a solution of the extended Nwogu's Boussinesq model.

30.4.2 Incident wave

For time-dependent wave generation, it is possible to consider waves induced by a boundary condition. This requires that the wave surface elevation and the velocity potential must satisfy appropriated boundary conditions, e.g., Dirichlet or Neumann conditions.

The simplest case is to consider a periodic wave given by:

$$\eta(x, t) = a \sin(kx - \omega t) \quad (30.27)$$

$$\Phi(x, t) = -\frac{c}{k} \cos(kx - \omega t) + K_2(t), \quad (30.28)$$

where c is the wave velocity magnitude and $K_2(t)$ is a time-dependent function of integration. This function $K_2(t)$ must satisfy the initial condition of the problem. In equations (30.27) as well as (30.28), one can note that the parameters a, c, k and ω are not all arbitrary, since they are related by the dispersion relation. One can also consider the superposition of water waves as solutions of the full linear problem with a constant depth.

30.4.3 Source function

In the work by Wei et al. [?], a source function for the generation of surface water waves was derived. This source function was obtained, using Fourier transform and Green's functions, to solve the linearized and non homogeneous equations of the Peregrine [?] and Nwogu's [?] models. This mathematical procedure can also be adapted here to deduce the source function.

We consider a monochromatic Gaussian wave generated by the following source function:

$$f(x, t) = D^* \exp(-\beta(x - x_s)^2) \cos(\omega t), \quad (30.29)$$

with D^* given by:

$$D^* = \frac{\sqrt{\beta}}{\omega \sqrt{\pi}} a \exp\left(\frac{k^2}{4\beta}\right) \frac{2}{15} h^3 k^3 g. \quad (30.30)$$

In the above expressions x_s is the center line of the source function and β is a parameter associated with the width of the generation band (cf. [?]).

30.5 Reflective walls and sponge layers

Besides the incident wave boundaries where the wave profiles are given, one must close the system with appropriate boundary conditions. We consider two more types of boundaries:

- i)* full reflective boundaries;
- ii)* partial reflective or absorbing boundaries.

The first case is modelled by the following equations:

$$\frac{\partial \Phi}{\partial \mathbf{n}} = 0, \quad \frac{\partial \eta}{\partial \mathbf{n}} = 0, \quad (30.31)$$

where \mathbf{n} is the outward unit vector normal to the computational domain Ω . We denote Γ as the boundary of Ω .

Note that in the finite element formulation, the full reflective boundaries (equations (30.31)) are integrated by considering zero Neumann-type boundary conditions.

Coupling the reflective case and an extra artificial layer, often called sponge or damping layer, we can simulate partial reflective or full absorbing boundaries. In this way, the reflected energy can be controlled. Moreover, one can prevent unwanted wave reflections and avoid complex wave interactions. It is also possible to simulate effects like energy dissipation by wave breaking.

In fact, a sponge layer can be defined as a subset \mathcal{L} of Ω where some extra viscosity term is added. As mentioned above, the system of equations can incorporate several extra damping terms, like that one provided by the inclusion of a dissipative model. Thus, the viscosity coefficient ν is described by a function of the form:

$$\nu(x, y) = \begin{cases} 0, & (x, y) \notin \mathcal{L}, \\ n_1 \frac{\exp\left(\frac{d(x, y)}{d_{\mathcal{L}}}\right)^{n_2} - 1}{\exp(1) - 1}, & (x, y) \in \mathcal{L}, \end{cases} \quad (30.32)$$

where n_1 and n_2 are, in general, experimental parameters, $d_{\mathcal{L}}$ is the sponge-layer diameter and d stands for the distance function between a point (x, y) and the intersection of Γ and the boundary of \mathcal{L} (see, e.g., [?] pag. 79).

30.6 Numerical Methods

We start this section by noting that a detailed description of the implemented numerical methods referred bellow can be found in the work of N. Lopes [?].

For simplicity, we only consider the second-order system described by equations (30.18) restricted to a stationary bottom and without dissipative, surface tension or extra source terms.

The model variational formulation is written as follows:

$$\begin{aligned}
 & \int_{\Omega} \frac{\partial \eta}{\partial t} \vartheta_1 \, dx dy + \frac{2}{5} \int_{\Omega} h^2 \nabla \left(\frac{\partial \eta}{\partial t} \right) \nabla \vartheta_1 \, dx dy - \frac{1}{3} \int_{\Omega} h(\nabla h) \nabla \left(\frac{\partial \eta}{\partial t} \right) \vartheta_1 \, dx dy + \\
 & + \frac{1}{15} \int_{\Omega} h \nabla h \frac{\partial \eta}{\partial t} \nabla \vartheta_1 \, dx dy + \int_{\Omega} \frac{\partial \Phi}{\partial t} \vartheta_2 \, dx dy = \int_{\Omega} (h + \eta) \nabla \Phi \nabla \vartheta_1 \, dx dy + \\
 & + \frac{2}{5} \int_{\Gamma} h^2 \frac{\partial}{\partial t} \left(\frac{\partial \eta}{\partial \mathbf{n}} \right) \vartheta_1 \, d\Gamma - \frac{1}{15} \int_{\Gamma} h \frac{\partial h}{\partial \mathbf{n}} \frac{\partial \eta}{\partial t} \vartheta_1 \, d\Gamma - \int_{\Gamma} (h + \eta) \frac{\partial \Phi}{\partial \mathbf{n}} \vartheta_1 \, d\Gamma - \\
 & - \frac{1}{2} \int_{\Omega} |\nabla \Phi|^2 \vartheta_2 \, dx dy - g \int_{\Omega} \eta \vartheta_2 \, dx dy - \frac{g}{15} \int_{\Omega} h^2 \nabla \eta \nabla \vartheta_2 \, dx dy - \\
 & - \frac{g}{15} \int_{\Omega} h(\nabla h)(\nabla \eta) \vartheta_2 \, dx dy + \frac{g}{15} \int_{\Gamma} h^2 \frac{\partial \eta}{\partial \mathbf{n}} \vartheta_2 \, d\Gamma,
 \end{aligned} \tag{30.33}$$

where the unknown functions η and Φ are the surface elevation and the transformed velocity potential, whereas ϑ_1 and ϑ_2 are the test functions defined in appropriate spaces.

The spatial discretization of this equation is implemented using low order Lagrange finite elements. In addition, the numerical implementation of (30.33) is accomplished using **FFC**.

We use a predictor-corrector scheme with an initialization provided by the Runge-Kutta method for the time integration. Note that the discretization of equation (30.33) can be written as follows:

$$M\dot{Y} = \mathbf{f}(t, Y), \tag{30.34}$$

where \dot{Y} and Y refer to $\left(\frac{\partial \eta}{\partial t}, \frac{\partial \Phi}{\partial t} \right)$ and to (η, Φ) , respectively. The known vector \mathbf{f} is related with the right-hand side of (30.33) and M is the coefficient matrix. In this way, the fourth order Adams-Bashforth-Moulton method can be written as follows:

$$\begin{cases}
 MY_{n+1}^{(0)} = MY_n + \frac{\Delta t}{24} [55\mathbf{f}(t_n, Y_n) - 59\mathbf{f}(t_{n-1}, Y_{n-1}) + 37\mathbf{f}(t_{n-2}, Y_{n-2}) - 9\mathbf{f}(t_{n-3}, Y_{n-3})] \\
 MY_{n+1}^{(1)} = MY_n + \frac{\Delta t}{24} [9\mathbf{f}(t_{n+1}, Y_{n+1}^{(0)}) + 19\mathbf{f}(t_n, Y_n) - 5\mathbf{f}(t_{n-1}, Y_{n-1}) + \mathbf{f}(t_{n-2}, Y_{n-2})]
 \end{cases} \tag{30.35}$$

where Δt is the time step, $t_n = n\Delta t$ ($n \in \mathbb{N}$) and $Y_n = (\eta, \Phi)$ evaluated at t_n . The predicted and corrected values of Y_n are denoted by $Y_n^{(0)}$ and $Y_n^{(1)}$, respectively. The corrector-step equation ((30.35)₂) can be iterated as function of a predefined error between consecutive time steps. For more details see, e.g., [?] or [?].

30.7 Numerical Applications

In this section, we present some numerical results about the propagation of surface water waves in an harbour with a geometry similar to that one of Fig. 30.1.

The colour scale used in all figures in this section is presented in Fig. 30.4. A schematic description of the fluid domain, namely the bottom profile and the sponge layer can be seen in Figs. 30.5 and 30.6, respectively. Note that a piecewise linear bathymetry is considered. A sponge layer is used to absorb the wave energy at the outflow region and to avoid strong interaction between incident and reflected waves in the harbour entrance. A monochromatic periodic wave, with an amplitude of 0.3m, is introduced at the indicated boundary (Dirichlet BC) in Fig. 30.6. This is achieved by considering a periodic Dirichlet boundary condition as described in the subsection 30.4.2. Full reflective walls are assumed as boundary conditions in all domain boundary except in the harbour entrance.

In Fig. 30.7 a wave elevation snapshot is shown. A zoom of the image, which describes the physical potential and velocity vector field in the still water plane, is given in the neighbourhood of the point $(x, y) = (150, 0)$ m at the same time step (see Fig. 30.8). In Fig. 30.9 two time steps of the speed in the still water plane are shown.

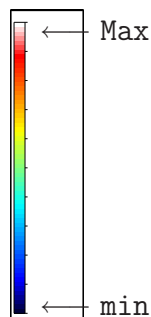


Figure 30.4: Scale.

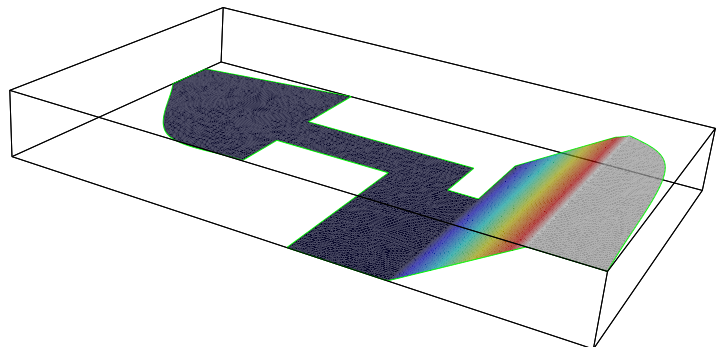


Figure 30.5: Impermeable bottom
[Max = -5.316 m, min = -13.716 m].

From these numerical results, one can conclude that the interaction between incident and reflected waves, near the harbour entrance, can generate wave amplitudes of, approximately, 0.84m. These amplitudes almost take the triple value of the incident wave amplitude. One can also observe an analogous behaviour for velocities. The maximum water speed of the incident waves at $z=0$ is, approximately, 1.2m/s whereas, after the interaction, the maximum speed value is almost 4m/s.

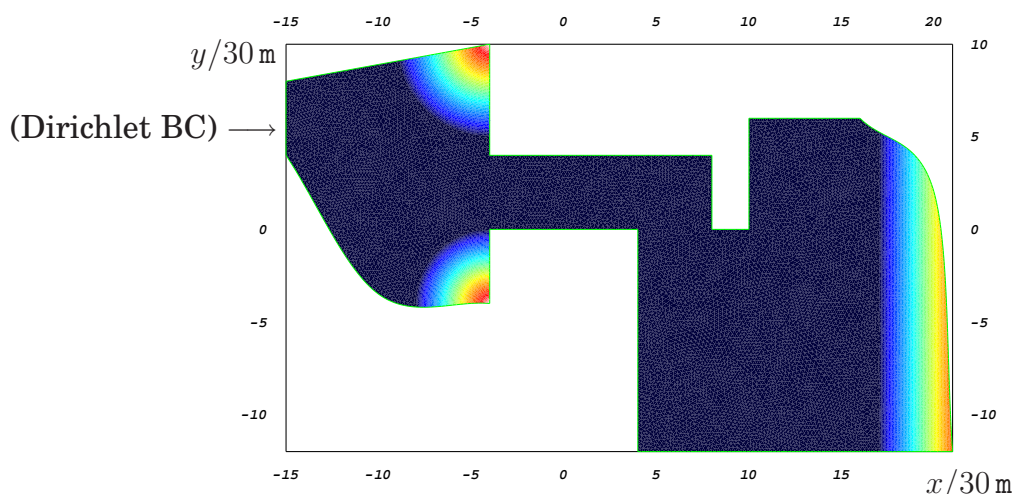


Figure 30.6: Sponge layer (viscosity) [Max $\approx 27 \text{ m}^2/\text{s}$, min = $0 \text{ m}^2/\text{s}$].

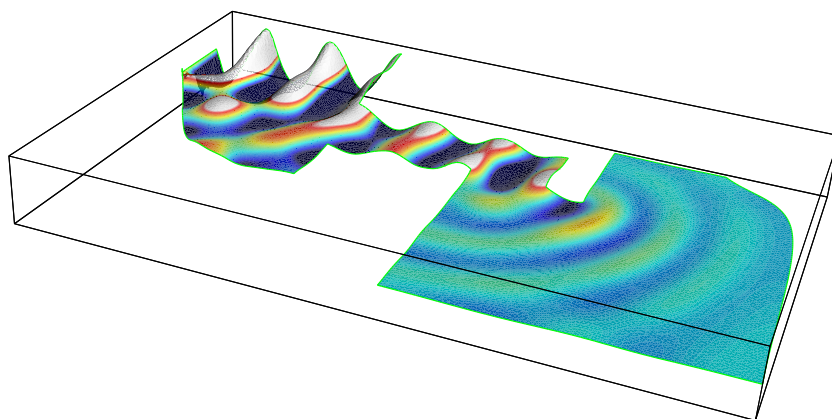


Figure 30.7: Surface elevation [Max $\approx 0.84 \text{ m}$, min $\approx -0.81 \text{ m}$].

30.8 Conclusions and future work

As far as we know, the finite element method is not often applied in surface water wave models based on the BEP formulation. In general, finite difference methods are preferred but are not appropriated for the treatment of complex geometries like the ones of harbours, for instance.

In fact, the surface water wave problems are associated with Boussinesq-type governing equations, which require very high order (≥ 6) spatial derivatives or a very high number of equations (≥ 6). A first approach, to the high-order models using discontinuous Galerkin finite element methods, can be found in [?].

From this work one can conclude that the FEniCS packages, namely **DOLFIN** and **FFC**, are appropriated to model surface water waves.

We have been developing **DOLFWAVE**, i.e., a FEniCS based application for

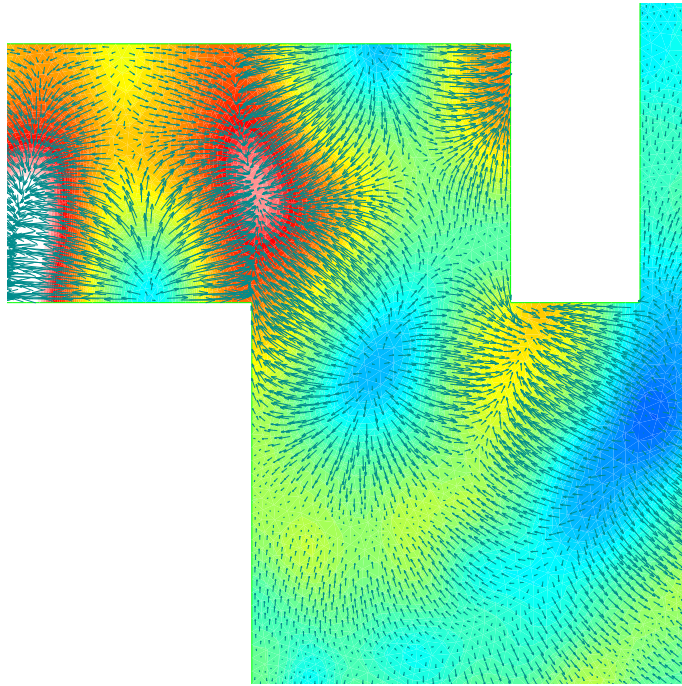


Figure 30.8: Velocity vector field at $z = 0$ and potential magnitude [Max $\approx 36 \text{ m}^2/\text{s}$, min $\approx -9 \text{ m}^2/\text{s}$].

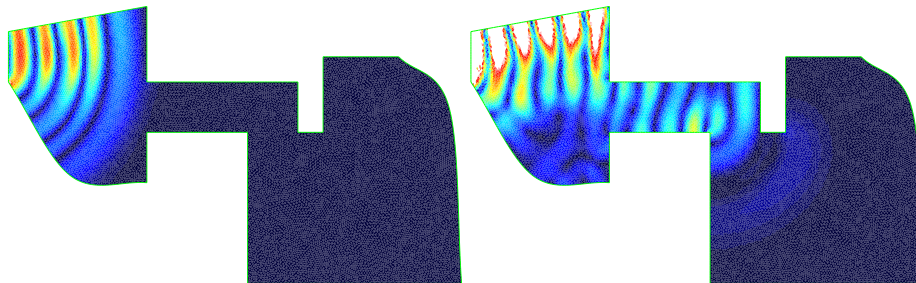


Figure 30.9: Water speed [Max $\approx 4 \text{ m/s}$, min = 0 m/s].

surface water waves. **DOLFWAVE** will be compatible with **Xd3d** post-processor³. The current state of the work, along with several numerical simulations, can be found at <http://ptmat.fc.ul.pt/~ndl>. This package will include some standard potential models of low order (≤ 4) as well as other new models to be submitted elsewhere by the authors [?].

³<http://www.cmap.polytechnique.fr/~jouve/xd3d/>

30.9 Acknowledgments

L. Trabucho work is partially supported by Fundação para a Ciência e Tecnologia, Financiamento Base 2008-ISFL-1-209. N. Lopes work is supported by Instituto Superior de Engenharia de Lisboa – ISEL.

N. Lopes^{*, \diamond , \dagger} , e-mail: ndl@ptmat.fc.ul.pt
P. Pereira^{*, $\#$} , e-mail: ppereira@deq.isel.ipl.pt
L. Trabucho ^{\diamond , \dagger} , e-mail: trabucho@ptmat.fc.ul.pt

* Área Científica de Matemática
ISEL-Instituto Superior de Engenharia de Lisboa,
Rua Conselheiro Emídio Navarro, 1
1959-007 Lisboa
Portugal

\diamond Departamento de Matemática
FCT-UNL-Faculdade de Ciências e Tecnologia
2829-516 Caparica
Portugal

\dagger CMAF-Centro de Matemática e Aplicações Fundamentais,
Av. Prof. Gama Pinto, 2
1649-003 Lisboa,
Portugal

$\#$ CEFITEC-Centro de Física e Investigação Tecnológica
FCT-UNL-Faculdade de Ciências e Tecnologia
2829-516 Caparica
Portugal

CHAPTER 31

Multiphase Flow Through Porous Media

By Xuming Shan and Garth N. Wells

Chapter ref: [**shan**]

Summarise work on automated modelling for multiphase flow through porous media.

Computing the Mechanics of the Heart

By Martin S. Alnæs, Kent-Andre Mardal and Joakim Sundnes

Chapter ref: **[alnes-4]**

Heart failure is one of the most common causes of death in the western world, and it is therefore of great importance to understand the behaviour of the heart better, with the ultimate goal to improve clinical procedures. The heart is a complicated electro-mechanical pump, and modelling its mechanical behaviour requires multiscale physics incorporating coupled electro-chemical and mechanical models both at the cell and continuum level. In this chapter we present a basic implementation of a cardiac electromechanics simulator in the FEniCS framework.

Simulation of Ca^{2+} Dynamics in the Dyadic Cleft

By Johan Hake

Chapter ref: [**hake**]

33.1 Introduction

From when we are children we hear that we should drink milk because it is an important source for calcium (Ca^{2+}), and that Ca^{2+} is vital for a strong bone structure. What we do not hear as frequently, is that Ca^{2+} is one of the most important cellular messengers we have in our body [?]. Among other things Ca^{2+} controls cell death, neural signaling, secretion of different chemical substances to the body, and what will be our focus in this chapter, the contraction of cells in the heart.

In this chapter I will first describe a mathematical model that can be used to model Ca^{2+} dynamics in a small sub cellular domain called the dyadic cleft. The model includes Ca^{2+} diffusion that is described by an advection-diffusion partial differential equation, and discrete channel dynamics that is described by stochastic Markov models. Numerical methods implemented in PyDOLFIN solving the partial differential equation will also be presented. A time stepping scheme for solving the stochastic and deterministic models in a coupled manner will be presented in the last section. Here will also a solver framework, `diffsim`, that implements the time stepping scheme together with the other presented numerical methods be presented.

33.2 Biological background

In a healthy heart every beat originates in the sinusoidal node, where pacemaker cells trigger an electric signal. This signal propagates through the whole heart, and results in a sudden change in electric potential between the interior and exterior of the heart cells. These two domains are separated by the cell membrane. The difference in electric potential between these domains is called the membrane potential. The sudden change in the membrane potential, an action potential, is facilitated by specialized ion channels that reside in the membrane. When an action potential arrives at a heart cell it triggers a Ca^{2+} channel that brings Ca^{2+} into the cell. The Ca^{2+} then diffuses over a small cleft, called the dyadic cleft, and triggers further Ca^{2+} release from an intracellular Ca^{2+} storage, the sarcoplasmic reticulum (SR). The Ca^{2+} ions then diffuse to the main intracellular domain of the cell, the cytosol, in which the contractile proteins are situated. The Ca^{2+} ions attach to these proteins and trigger contraction. The strength of the contraction is controlled by the strength of the Ca^{2+} concentration in the cytosol. The contraction is succeeded by a period of relaxation, which is facilitated by the extraction of Ca^{2+} from the intracellular space by various proteins.

This chain of events is called the Excitation Contraction (EC) coupling [?]. Several severe heart diseases can be related to impaired EC coupling and by broadening the knowledge of the coupling it will also be possible to develop better treatments for the diseases. Although the big picture of EC coupling is straightforward to grasp it conceals the nonlinear action of hundreds of different protein species. Computational methods have emerged as a natural complement to experimental studies in the ongoing strive to better understand the intriguing coupling, and it is in this light the present study is presented. Here I will focus on the initial phase of the EC coupling, i.e., when Ca^{2+} flows into the cell and triggers further Ca^{2+} release.

33.3 Mathematical models

33.3.1 Geometry

The dyadic cleft is the space between a structure called the t-tubule (TT) and the SR. TT is a network of pipe-like invaginations of the cell membrane that perforate the heart cell[?]. Fig. 33.1 A presents a sketch of a small part of a single TT together with a piece of SR. Here one can see that the junctional SR (jSR) wraps the TT, and the small volume between these structures is the dyadic cleft. The space is not well defined as it is crowded with channel proteins, and the size of it also varies. In computational studies it is commonly approximated as a disk or a rectangular slab [?, ?, ?, ?]. In this study I have used a disk, see Fig. 33.1 A. The height of the disk is: $h = 12\text{nm}$, and the radius is: $r = 50\text{nm}$.

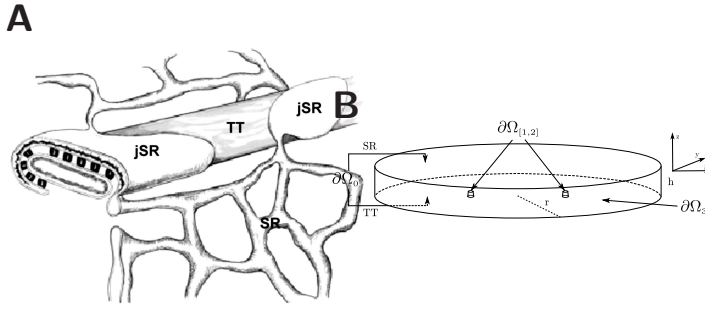


Figure 33.1: **A:** A diagram showing the relationship between the TT, the SR and the jSR. The volume between the flat jSR and the TT is the dyadic cleft. The black structures in the cleft are Ryanodine receptors, which are large channel proteins. The figure is from [?]. **B:** The geometry that is used for the dyadic cleft. The top of the disk are the cell membrane of the SR, or jSR, the bottom the cell membrane of the TT, and the circumference of the disk is the interface to the cytosole. The top of the two small elevations models the mouths of two ion channels.

Larger radius can be used, e.g., up to 200 nm, but due to numerical limitations I have to limit the size of the cleft, see below. The diffusion constant of Ca^{2+} was set to $\sigma = 10^5 \text{ nm}^2 \text{ ms}^{-1}$ [?].

33.3.2 Ca^{2+} Diffusion

Electro-Diffusion

The cell membrane, also called the sarcolemma, consists of a lipid bi-layer, which produce an electric potential in the solution. This potential is due to negatively charged phospholipid head-groups [?, ?]. Therefore in close proximity to the sarcolemma, an electric double layer is produced [?]. I am going to use the Gouy-Chapman method, which defines a *diffuse layer* to describe this double layer [?]. The theory introduces an advection term to the ordinary diffusion equation, which makes the resulting equation harder to solve.

The ion flux in a solution that experience an electric field is governed by the Nernst-Planck equation,

$$J = -\sigma (\nabla c - 2cE), \quad (33.1)$$

where σ is the diffusion constant of Ca^{2+} , $c = c(x, t)$ is the Ca^{2+} concentration, $E = E(x)$ is the non-dimensional electric field (the original electric field scaled with e/kT) and 2 is the valence of Ca^{2+} . Assuming conservation of mass, we arrive at the general advection-diffusion equation,

$$\dot{c} = \sigma [\Delta c - \nabla \cdot (2cE)]. \quad (33.2)$$

E follows from the non-dimensional potential, ψ , (the ordinary electric potential scaled with e/kT) in the solution as,

$$E = -\nabla\psi. \quad (33.3)$$

The strength of ψ is defined by the amount of charged head-groups in the lipid bi-layers and by the combined screening effect of all ions in the dyadic cleft. Following previous work done by [?] and [?] all other ions in the solution will be treated as being in steady state. The sarcolemma is assumed to be planar and effectively infinite. This let us use an approximation of the electric potential in the solution,

$$\psi(z) = \psi_0 \exp(-\kappa z). \quad (33.4)$$

Here ψ_0 is the non-dimensional potential at the membrane, κ the inverse Debye length and z the distance from the sarcolemmar in a perpendicular direction. [?] showed that for a large range of $[Ca^{2+}]$, ψ_0 stayed constant at -2.2 and κ is also assumed to be 1 nm.

► **Editor note:** Check formula in (33.4). Got undefined control sequence when comping so needed to edit.

Boundary fluxes

The SR and TT membrane is impermeable for ions, effectively making $\partial\Omega_0$, in Fig. 33.1, a no-flux boundary, giving us,

$$J_0 = 0. \quad (33.5)$$

The main sources for Ca²⁺ inflow to the dyadic cleft in our model, is the L-type Ca²⁺ channel (LCC). This flux comes in at the $\partial\Omega_{[1,2]}$ boundaries, see Fig. 33.1. After entering the cleft the Ca²⁺ then diffuse to the RyR situated at the SR membrane, triggering more Ca²⁺ inflow. This flux will not be included in the simulations, however the stochastic dynamic of the opening of the channel will be included, see Section 33.3.3 below. The Ca²⁺ that enters the dyadic cleft diffuse into the main compartement of cytosole introducing a third flux at the $\partial\Omega_3$ boundary.

The LCC is a stochastic channel that are modelled as either open or close. When the channel is open Ca²⁺ flows into the cleft. The dynamic that describes the stochastic behaviour is presented in Section 33.3.3 below. The LCC flux is modelled as a constant current with amplitude, -0.1 pA, which corresponds to the amplitude during voltage clamp to 0 mV [?]. The LCC flux is then,

$$J_{[1,2]} = \begin{cases} 0 & : \text{close channel} \\ -\frac{i}{2FA}, & : \text{open channel} \end{cases} \quad (33.6)$$

where i is the amplitude, 2 the valence of Ca²⁺, F Faraday's constant and A the area of the channel. Note that inward current is by convention negative.



Figure 33.2: **A**: State diagram of the discrete LCC Markov model from [?]. Each channel can be in one of the 12 states. The transition between the states are controlled by propensities. The α , and β are voltage dependent, γ is $[Ca^{2+}]$ dependent and f , a , b , and ω are constant, see [?] for further details. The channels operates in two modes: *Mode normal*, represented by the states in the upper row, and *Mode Ca*, represented the states in the lower row. In state 6 and 12 the channel is open, but state 12 is rarely entered as $f' \ll f$, effectively making *Mode Ca* an inactivated mode. **B**: State diagram of a RyR from [?]. The α and γ propensities are Ca^{2+} dependent, representing the activation and inactivation dependency of the cytosolic $[Ca^{2+}]$. The β and δ propensities are constant.

► Editor note: Placement of figures seems strange.

The flux to cytosole is modeled as a concentration dependent flux,

$$J_3 = -\sigma \frac{c - c_0}{\Delta_s}, \quad (33.7)$$

where c is the concentration in the cleft at the boundary, c_0 the concentration in cytosole, and Δ_s the distance to the center of the cytosole.

33.3.3 Stochastic models of single channels

Discrete and stochastic Markov chain models are used to describe single channels dynamics. Such models consists of a variable that can be in a certain number of discrete states. The transition between these states is a stochastic event. The frequency of these events are determined by the propensity functions associated to each transition. These functions characterize the probability per unit time that the corresponding transition event occurs and are dependent on the chosen Markov chain model and they may vary with time.

L-type Ca^{2+} channel

The LCC is the main source for extracellular Ca^{2+} into the cleft. The channel opens when an action potential arrives to the cell and inactivates when single Ca^{2+} ions binds to binding sites on the intracellular side of the channel. An LCC is composed by a complex of four transmembrane subunits, which each can be permissive or non-permissive. For the whole channel to be open, all four subunits need to be permissive and the channel then has to undergo a last conformational change to an opened state [?]. In this chapter I am going to use a Markov model of the LCC that incorporates a voltage dependent activation together with a Ca^{2+} dependent inactivation [?, ?]. The state diagram of this model is presented in Fig. 33.2 A. It consists of 12 states, where state 6 and 12 are the only conducting states, i.e., when the channel is in one of these states it is open. The transition propensities are defined by a set of functions and constants, which are all described in [?].

Ryanodine Receptors

RyRs are Ca^{2+} specific channels that are situated on the SR in clusters of several channels [?, ?]. They open by single Ca^{2+} ions attaching to the receptors at the cytosolic side. A modified version of a phenomenological model that mimics the physiological functions of the RyRs, first presented by [?], will be used. The model consists of four states where one is conducting, state 2, see Fig. 33.2 B. The α and γ propensities are Ca^{2+} dependent, representing the activation and inactivation dependency of cytosolic $[\text{Ca}^{2+}]$. The β and δ propensities are constants. For specific values for the propensities consider [?].

33.4 Numerical methods for the continuous system

The continuous problem is defined by Eq. (33.2 -33.7) together with an initial condition. Given a bounded domain $\Omega \subset \mathbb{R}^3$ with the boundary, $\partial\Omega$, we want to find $c = c(x, t) \in \mathbb{R}_+$, for $x \in \Omega$ and $t \in \mathbb{R}_+$, such that:

$$\begin{cases} \dot{c} = \sigma \Delta c - \nabla \cdot (ca) & \text{in } \Omega \\ \sigma \frac{\partial c}{\partial n} = J_k & \text{on } \partial\Omega_k, \end{cases} \quad (33.8)$$

with $c(\cdot, 0) = c_0(x)$. Here $a = a(x) = 2\sigma E(x)$, and, J_k and $\partial\Omega_k$ are the k^{th} flux at the k^{th} boundary, where $\bigcup_k \partial\Omega_k = \partial\Omega$. The J_k are given by Eq. (33.5)- (33.7).

```

1 from numpy import *
2 from dolfin import *
3
4 mesh = Mesh('cleft_mesh.xml.gz')
5
6 Vs = FunctionSpace(mesh, "CG", 1)
7 Vv = VectorFunctionSpace(mesh, "CG", 1)
8
9 v = TestFunction(Vs)
10 u = TrialFunction(Vs)
11
12 # Defining the electric field-function
13 a = Function(Vv,["0.0","0.0","phi_0*valence*kappa*sigma*exp(-kappa*x[2])"],
14             {"phi_0":-2.2,"valence":2,"kappa":1,"sigma":1.e5})
15
16 # Assembly of the K, M and A matrices
17 K = assemble(dot(grad(u),grad(v))*dx)
18 M = assemble(u*v*dx)
19 E = assemble(-u*dot(a,grad(v))*dx)
20
21 # Collecting face markers from a file, and skip the 0 one
22 sub_domains = MeshFunction("uint",mesh,"cleft_mesh_face_markers.xml.gz")
23 unique_sub_domains = list(set(sub_domains.values()))
24 unique_sub_domains.remove(0)
25
26 # Assemble matrices and source vectors from exterior facets domains
27 domain = MeshFunction("uint",mesh,2)
28 F = {};f = {};tmp = K.copy(); tmp.zero()
29 for k in unique_sub_domains:
30     domain.values()[k] = (sub_domains.values() != k)
31     F[k] = assemble(u*v*ds, exterior_facet_domains = domain, \
32                   tensor = tmp.copy(), reset_tensor = False)
33     f[k] = assemble(v*ds, exterior_facet_domains = domain)

```

Figure 33.3: Python code for the assembly of the matrices and vectors from Eq. (33.14)-(33.15).

33.4.1 Discretization

The continuous equations are discretized using the Finite element method. Eq. (33.8) is multiplied with a proper test function, v , and we get:

$$\int_{\Omega} \dot{c}v \, dx = \int_{\Omega} [\sigma \Delta c - \nabla(ca)] v \, dx, \quad (33.9)$$

and we integrate by part and get:

$$\int_{\Omega} \dot{c}v \, dx = - \int_{\Omega} (\sigma \nabla c - ca) \nabla v \, dx + \sum_k \int_{\partial\Omega_k} J_k v \, ds_k. \quad (33.10)$$

Consider a tetrahedralization of Ω , a mesh, where the domain is divided into disjoint subdomains, Ω_e , elements. A discrete solution $c_h \in V_h$ is defined. Here $V_h = \{\phi \in H^1(\Omega) : \phi \in P^k(\Omega_e) \forall e\}$, and P^k represents the space of Lagrange polynomials of order k . The backward Euler method is used to approximate the time derivative and Eq. (33.10) can now be stated as follows: given c_h^n find $c_h^{n+1} \in V_h$ such that:

$$\int_{\Omega} \frac{c_h^{n+1} - c_h^n}{\Delta t} v \, dx = - \int_{\Omega} (\sigma \nabla c_h^{n+1} - c_h^{n+1} a) \cdot \nabla v \, dx + \sum_k \int_{\partial\Omega} J_k v \, ds_k, \forall v \in V_h \quad (33.11)$$

where Δt is the time step. The trial function $c_h^n(x)$ is expressed as a weighted sum of basis functions,

$$c_h^n(x) = \sum_j^N C_j^n \phi_j(x), \quad (33.12)$$

where C_j^n are the coefficients. Lagrange polynomials of first order is used for both the test and the trial function, $k = 1$, and the number of unknowns, N , will then coincide with the number of vertices of the mesh.

The test function v is chosen from the same discrete basis as $c_h^n(x)$, i.e., $v_i(x) = \phi_i(x) \in V_h$, for $i \in [1 \dots N]$. These are used in Eq. (33.11) to produce an algebraic problem on the following form:

$$\frac{1}{\Delta t} \mathbf{M} (C^{n+1} - C^n) = \left(-\mathbf{K} + \mathbf{E} + \sum_k \alpha^k \mathbf{F}^k \right) C_j^{n+\frac{1}{2}} + \sum_k c_0^k f^k, \quad (33.13)$$

where $C^n \in \mathbb{R}^N$ is the vector of coefficients from the discrete solution $c_h^n(x)$, α^k and c_0^k are constant coefficients related to the k^{th} flux and

$$\begin{aligned} M_{ij} &= \int_{\Omega} \phi_i \phi_j \, dx, & K_{ij} &= \int_{\Omega} \nabla \phi_i \cdot \nabla \phi_j \, dx, \\ E_{ij} &= \int_{\Omega} a \phi_i \cdot \nabla \phi_j \, dx, & F_{ij}^k &= \int_{\partial\Omega_k} \phi_i \phi_j \, ds, \end{aligned} \quad (33.14)$$

```

1 # Defining the stabilization using local Peclet number
2 cppcode = """class Stab: public Function {
3 public:
4     Function* field; uint _dim; double sigma;
5     Stab(const FunctionSpace& V): Function(V)
6     {field = 0; sigma=1.0e5;}
7     void eval(double* v, const Data& data) const {
8         if (!field)
9             error("Attach a field function.");
10            double field_norm = 0.0; double tau = 0.0;
11            double h = data.cell().diameter();
12            field->eval(v,data);
13            for (uint i = 0; i < geometric_dimension(); ++i)
14                field_norm += v[i]*v[i];
15            field_norm = sqrt(field_norm);
16            double PE = 0.5*field_norm * h/sigma;
17            if (PE > DOLFIN_EPS)
18                tau = 1/tanh(PE)-1/PE;
19            for (uint i = 0; i < geometric_dimension(); ++i)
20                v[i] *= 0.5*h*tau/field_norm;}};
21 """
22 stab = Function(Vv, cppcode); stab.field = a
23
24 # Assemble the stabilization matrices
25 E_stab = assemble(div(a*u)*dot(stab,grad(v))*dx)
26 M_stab = assemble(u*dot(stab,grad(v))*dx)
27
28 # Adding them to the A and M matrices, weighted by the global tau
29 tau = 0.28; E.axpy(tau,E_stab); M.axpy(tau,M_stab)

```

Figure 33.4: Python code for the assembly of the SUPG term for the mass and advection matrices.

are the entries in the \mathbf{M} , \mathbf{K} , \mathbf{E} and \mathbf{F}^k matrices. f^k are boundary source vectors corresponding to the k^{th} boundary. The vector elements are given by:

$$f_i^k = \int_{\partial\Omega_k} \phi_i ds. \quad (33.15)$$

The code for producing the matrices and vectors in Eq. (33.14)-(33.15) is presented in Fig. 33.3. Note that in the last for loop we iterate over the unique subdomains, and set the entries of the `MeshFunction` domain corresponding to the k^{th} boundary to 0 and the other entries to 1. In this way the same form for the exterior facet domain integrals can be used.

The system in Eq. (33.13) is linear and the matrices and vectors can be pre-assembled. This allows for a flexible system where boundary matrices and boundary source vectors can be added, when a channel opens. Δt can also straightforwardly be decreased when such an event occurs. This adaptation in time is crucial both for the numerical stability of the linear system. Δt can then be increased after each time step as the demand on the size of Δt falls. The sparse linear system is solved using the `PETSclinear algebra backend` [?] in `PyDOLFIN` together with the `Bi-CGSTAB` iterative solver [?], and the `BoomerAMG` preconditioners from `hypre` [?]. In Fig. 33.5 a script is presented that solves the algebraic system from Eq. (33.13) together with a crude time stepping scheme for the opening and closing of the included LCC flux.

33.4.2 Stabilization

It turns out that the algebraic system in Eq. (33.13) is numerically unstable for physiological relevant values of a , see Section 33.3.2. This is due to the transport term introduced by A_{ij} from Eq. (33.14). I have chosen to stabilize the system using the `Streamline upwind Petrov-Galerkin (SUPG)` method [?]. This method adds a discontinuous streamline upwind contribution to the testfunction in Eq. (33.9),

$$v' = v + s, \text{ where } s = \tau \frac{h\tau_l}{2\|a\|} a \cdot \nabla v. \quad (33.16)$$

Here $\tau \in [0, 1]$ is problem dependent, $h = h(x)$ is the size of the local element of the mesh, and $\tau_l = \tau_l(x)$, is given by,

$$\tau_l = \coth(\text{PE}_l) - 1/\text{PE}_l, \quad (33.17)$$

where PE_l is the local Péclet number:

$$\text{PE}_l = \|a\| h / 2\sigma. \quad (33.18)$$

This form of τ_l follows the optimal stabilization from an 1D case [?], other choices exist. The contribution from the diffusion term of the weak form can be eliminated by choosing a test function from a first order Lagrange polynomial, as the

Δ operator will reduce the trial function to zero. The PyDOLFINcode that assembles the SUPG part of the problem is presented in Fig. 33.4. In the script two matrices, `E_stab` and `M_stab` are assembled, which are both added to the corresponding advection and mass matrices `E` and `M` weighted by the global parameter `tau`.

A mesh with finer resolution close to the TT surface, at $z = 0$ nm, is also used to further increase the stability. It is at this point the electric field is at its strongest and it attenuates fast. At $z = 3$ nm the field is down to 5% of the maximal amplitude, and at $z = 5$ nm, it is down to 0.7%, reducing the need for high mesh resolutions. The mesh generator `tetgen` is used to produce meshes with the needed resolution [?].

The global stabilization parameter τ , is problem dependent. To find an optimal τ , for a certain electrical field and mesh, the sytem in Eq. (33.13) is solved to steady state using only homogeneous Neumann boundary conditions. An homogeneous concentration of $c_0 = 0.1 \mu\text{M}$ is used as the initial condition. The numerical solution is then compared with the analytic solution of the problem. This solution is acquired by setting $J = 0$ in Eq. (33.1) and solving for the c , with the following result:

$$c(z) = c_b \exp(-2\psi(z)). \quad (33.19)$$

Here ψ is given by Eq. (33.4), and c_b is the concentration in the bulk, i.e., where z is large. c_b was chosen such that the integral of the analytic solution was equal to $c_0 \times V$, where V is the volume of the domain.

► Editor note: Check equation (33.19).

The error of the numerical solution for different values of τ and for three different mesh resolutions are plotted in Fig. 33.6. The meshes are enumerated from 1-3. The error is given in a normalized L2 norm. As expected we see that the mesh with the finest resolution produce the smallest error. The mesh resolutions are quantified by the number of vertices close to $z = 0$. In the legend of Fig. 33.6 the median of the z distance of all vertices and the total number of vertices in each mesh is presented. The three meshes were created such that the vertices closed to $z = 0$ were forced to be situated at some fixed distances from $z = 0$. Three numerical and one analytical solution for the three different meshes are plotted in Fig. 33.7- 33.9. The numerical solutions are from simulations using three different τ : 0.1, 0.6 and the L2-optimal τ , see Fig. 33.6. The traces in the figures are all picked from a line going from (0,0,0) to (0,0,12).

In Fig. 33.7 the traces from mesh 1 is plotted. Here we see that all numerical solutions are quite poor for all τ . The solution with $\tau = 0.10$ is unstable as it oscillates and produces negative concentration. The solution with $\tau = 0.60$ seems stable but it undershoots the analytic solution at $z = 0$ with $\tilde{1.7} \mu\text{M}$. The solution with $\tau = 0.22$ is the L2-optimal solution for mesh 1, and approximates the analytic solution at $z = 0$ well.

In Fig. 33.8 the traces from mesh 2 is presented in two plots. The left plot

Simulation of Ca^{2+} Dynamics in the Dyadic Cleft

```
1 # Time parameters
2 dt_min = 1.0e-8; dt = dt_min; t = 0; c0 = 0.1; tstop = 1.0
3 events = [0.2,tstop/2,tstop,tstop]; dt_expand = 2.0;
4
5 # Boundary parameters
6 t_channels = {1:[0.2,tstop/2], 2:[tstop/2,tstop]}
7 sigma = 1e5; ds = 20; area = 3.1416; Faraday = 0.0965; amp = -0.1
8
9 # Initialize the solution Function and the left and right hand side
10 u = Function(Vs); x = u.vector()
11 x[:] = c0*exp(-a.valence*a.phi_0*exp(-a.kappa*mesh.coordinates()[:,-1]))
12 b = Vector(len(x); A = K.copy();
13
14 solver = KrylovSolver("bicgstab","amg_hypre")
15 dolfin_set("Krylov relative tolerance",1e-7)
16 dolfin_set("Krylov absolute tolerance",1e-10);
17
18 plot(u, vmin=0, vmax=4000, interactive=True)
19 while t < tstop:
20     # Initalize the left and right hand side
21     A[:] = K; A *= sigma; A += E; b[:] = 0
22
23     # Adding channel fluxes
24     for c in [1,2]:
25         if t >= t_channels[c][0] and t < t_channels[c][1]:
26             b.axpy(-amp*1e9/(2*Faraday*area),f[c])
27
28     # Adding cytosole flux at Omega 3
29     A.axpy(sigma/ds,F[3]); b.axpy(c0*sigma/ds,f[3])
30
31     # Applying the Backward Euler time discretization
32     A *= dt; b *= dt; b += M*x; A += M
33
34     solver.solve(A,x,b)
35     t += dt; print "Ca Concentration solved for t:",t
36
37     # Handle the next time step
38     if t == events[0]:
39         dt = dt_min; events.pop(0)
40     elif t + dt*dt_expand > events[0]:
41         dt = events[0] - t
42     else:
43         dt *= dt_expand
44
45     plot(u, vmin=0, vmax=4000)
46
47 plot(u, vmin=0, vmax=4000, interactive=True)
```

Figure 33.5: Python code for solving the system in Eq. (33.13), using the assembled matrices from the two former code examples from Fig. 33.3- 33.4.

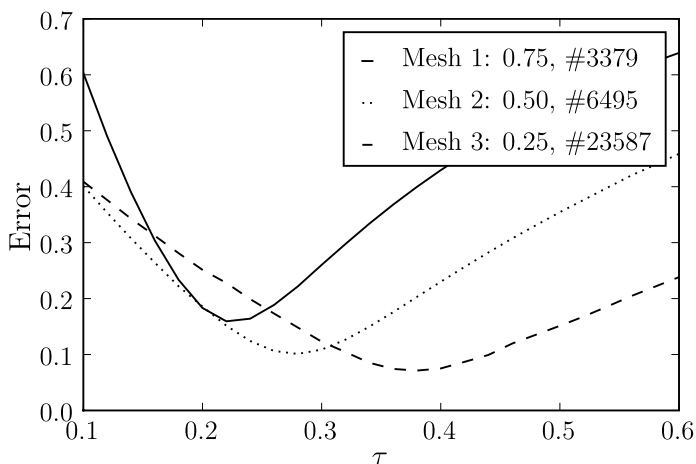


Figure 33.6: The figure shows a plot of the error (a normalized L2 norm of the difference between the numerical and analytical solutions) against the stabilization parameter τ for 3 different mesh resolutions. The mesh resolutions are given by the median of the z distance of all vertices and the total number of vertices in the mesh, see legend. We see that the minimal values of the error for the three meshes, occur at three different τ : 0.22, 0.28, and 0.38.

shows the traces for $z < 1.5$ nm and the right shows the traces for $z > 10.5$ nm. In the left plot we see the same tendency as in Fig. 33.7, an overshoot of the solution with $\tau = 0.10$ and an undershoot for the solution with $\tau = 0.60$. The L2-optimal solution, the one with $\tau = 0.28$, overshoot the analytic solution for the shown interval in the left plot, but undershoot for the rest of the trace.

In the last figure, Fig. 33.9, traces from mesh 3 is presented. The results is also here presented in two plots, corresponding to the same z interval as in Fig. 33.8. We see that the solution with $\tau = 0.10$ is not good in either plots. In the left plot it clearly overshoots the analytic solution for most of the interval, and then stays at a lower level than the analytic solution for the rest of the interval. The solution with $\tau = 0.60$ is much better here than in the two previous plots. It undershoots the analytic solution at $z = 0$ but stays closer to it for the rest of the interval than the L2-optimal solution. The L2 norm penalize larger distances between two traces, i.e., weighting the error close to $z = 0$ more than the rest. The optimal solution measured in the Max norm is given when $\tau = 50$, result not shown.

These results tell us that it is difficult to get accurate numerical solution for the advection-diffusion problem presented in Eq. (33.8), even with optimal SUPG stabilization for the given mesh resolutions. Using finer mesh close to $z = 0$ would help, but it will create a larger system. It is interesting to notice that the L2 optimal solutions is better close to $z = 0$, than other solutions and

the solution for the largest τ is better than other for $z \lesssim 2$ nm. For a modeller these constraints are important to know about because the solution at $z = 0$ and $z = 12$ nm is the most important, as Ca^{2+} interact with other proteins at these points.

I am combining a solver of the continuous and deterministic advection-diffusion equation, Eq. (33.2), and a solver of the discrete and stochastic systems of Markov chain models from Section 33.3.3. These systems are two-way coupled as some of the propensities in the Markov chains are dependent on the local $[\text{Ca}^{2+}]$ and boundary fluxes are turned on or off dependent on what state the Markov models are in. I have used a hybrid approach similar to the one presented in [?] to solve this system. Basically this hybrid method consists of a modified Gillespie method [?] to solve the stochastic state transitions, and a finite element method in space together with a backward Euler method in time, to solve the continuous system.

33.5 `diffsim` an event driven simulator

In the scripts in Fig. 33.3- 33.5 it is shown how a simple continuous solver can be built with PyDOLFIN. By pre-assemble the matrices from Eq. (33.14) a flexible system for adding and removing boundary fluxes corresponding to the state of the channels is constructed. The script in Fig.33.5 uses fixed time steps for the channel states. These time steps together with an expanding Δt form a simplistic time stepping scheme that is sufficient to solve the presented example. However it would be difficult to expand it to also incorporate the time stepping involved with the solution of stochastic Markov models, and other discrete variables. For this I have developed an event driven simulator called `diffsim`. In the last subsections in this chapter I will present the algorithm underlying the time stepping scheme in `diffsim` and an example of how one can use `diffsim` to describe and solve a model of the dyadic cleft. The `diffsim` software can be freely downloaded from URL:http://www.fenics.org/wiki/FEniCS_Apps.

33.5.1 *Stochastic system*

The stochastic evolution of the Markov chain models from Section 33.3.3 is determined by a modified Gillespie method [?], which resembles the one presented in [?]. I will not go into detail of the actual method, but rather explain the part of the method that has importance for the overall time stepping algorithm, see below.

The solution of the included stochastic Markov chain models is stored in a state vector, S . Each element in S corresponds to one Markov model and the value reflects which state each model is in. The transitions between these states are modelled stochastically and are computed using the modified Gillespie method.

This method basically give us which of the states in S changes to what state and when. It is not all such state transitions that are relevant for the continuous system, e.g. a transition between two closed states in the LCC model will not have any impact on the boundary fluxes, and can be ignored. Only transitions that either open or close a channel, which is called channel transitions, will be recognized. The modified Gillespie method assume that any continuous variables a certain propensity function is dependent on are constant during a time step. The error done by assuming this is reduced by taking smaller time steps right after a channel transition, as the continuous field is changing dramatically during this time period.

33.5.2 Time stepping algorithm

To simplify the presentation of the time stepping algorithm we only consider one continuous variable, this could for example be the Ca^{2+} field. The framework presented here can be expanded to also handle several continuous variables. We define a base class called `DiscreteObject` which defines the interface for all discrete objects. A key function of a discrete object is to know when its *next event* is due to. The `DiscreteObject` that has the smallest next event time, gets to define the size of the next Δt , for which the Ca^{2+} field is solved with. In python this is easily done by making the `DiscreteObjects` sortable with respect to their next event time. All `DiscreteObjects` is then collected in a list, `discrete_objects` see Fig. 33.10, and the `DiscreteObject` with the smallest next event time is then just `min(discrete_objects)`.

An event from a `DiscreteObject` that does not have an impact on the continuous solution will be ignored, e.g., a Markov chain model transition that is not a channel transition. A transition needs to be realized before we can tell if it is a channel transition or not. This is done by *stepping* the `DiscreteObject`, calling the objects `step()` method. If the method returns `False` it will not affect the Ca^{2+} field, and we enter the while loop, and a new `DiscreteObject` is picked, see Fig. 33.10. However if the object returns `True` when it is stepped, we will exit the while loop and continue. Next we have to update the other discrete objects with the chosen Δt , solve the Ca^{2+} field, broadcast the solution and last but not least execute the discrete event that is scheduled to happen at Δt .

► Editor note: *Fix ugly text sticking out in margin.*

In Fig. 33.11 we show an example of a possible realization of this algorithm. The example starts at $t=2\text{ms}$ at the top-most timeline represented by **A**, and it includes three different types of `DiscreteObjects`: i) `DtExpander`, ii) `StochasticHandler` and iii) `TStop`. See the legend of the figure for more details.

33.5.3 *diffsim* an example

`diffsim` is a versatile event driven simulator that incorporates the time stepping algorithm presented in the previous section together with the infrastructure to solve models with one or more diffusional domains, defined by a computational mesh. Each such domain can have several diffusive ligands. Custom fluxes can easily be included through the framework `diffsim` give. The submodule `dyadic-cleft` implements some published Markov models that can be used to simulate the stochastic behaviour of a dyad and some convenient boundary fluxes. It also implements the field flux from the lipid bi-layer discussed in Section 33.3.2. In Fig. 33.12 runnable script is presented, which simulate the time to release, also called the latency for a dyad. The two Markov models that is presented in section 33.3.3 is here used to model the stochastic dynamics of the RyR and the LCC. The simulation is driven by an external dynamic voltage clamp. The data that defines this is read in from a file using utilities from the NumPyPythonpackages.

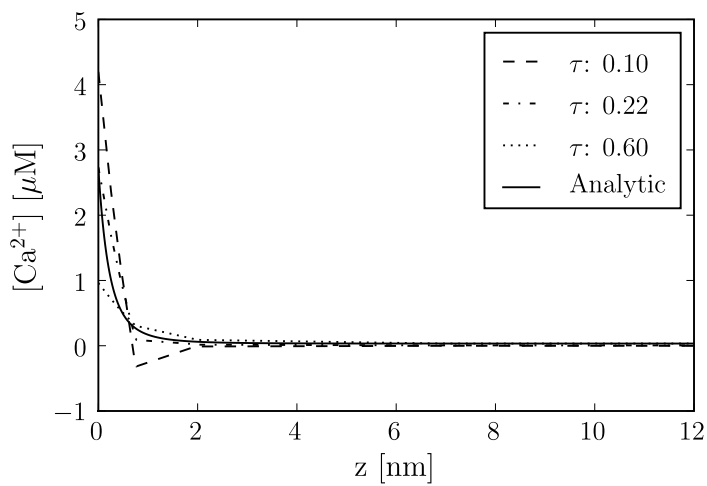


Figure 33.7: The figure shows the concentration traces of the numerical solutions from Mesh 1, see legend of Fig. 33.6, for three different τ together with the analytic solution. The solutions were picked from a line going between the points $(0,0,0)$ and $(0,0,12)$. We see that the solution with $\tau = 0.10$ oscillates. The solution with $\tau = 0.22$ was the solution with smallest global error for this mesh, see Fig 33.6, and the solution with $\tau = 0.60$ undershoots the analytic solution at $z = 0\text{nm}$ with $\tilde{1.7} \mu\text{M}$.

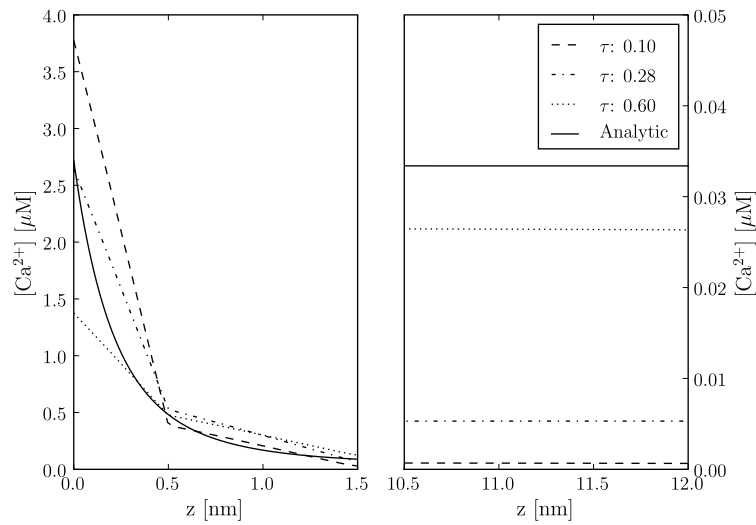


Figure 33.8: The figures shows the concentration traces of the numerical solutions from Mesh 2, see legend of Fig. 33.6, for three different τ together with the analytic solution. The traces in the two panels were picked from a line going between the points $(0,0,0)$ and $(0,0,1.5)$ for the left panel and between $(0,0,10.5)$ and $(0,0,12)$ for the right panel. We see from both panels that the solution with $\tau = 0.10$ give the poorest solution. The solution with $\tau = 0.28$ was the solution with smallest global error for this mesh, see Fig 33.6, and this is reflected in the reasonable good fit seen in the left panel, especially at $z = 0\text{nm}$. The solution with $\tau = 0.60$ undershoots the analytic solution at $z = 0$ with $\tilde{1}.2 \mu\text{M}$. From the right panel we see that all numerical solutions undershoot at $z = 15\text{nm}$, and that the trace with $\tau = 0.60$ comes closest the analytic solution.

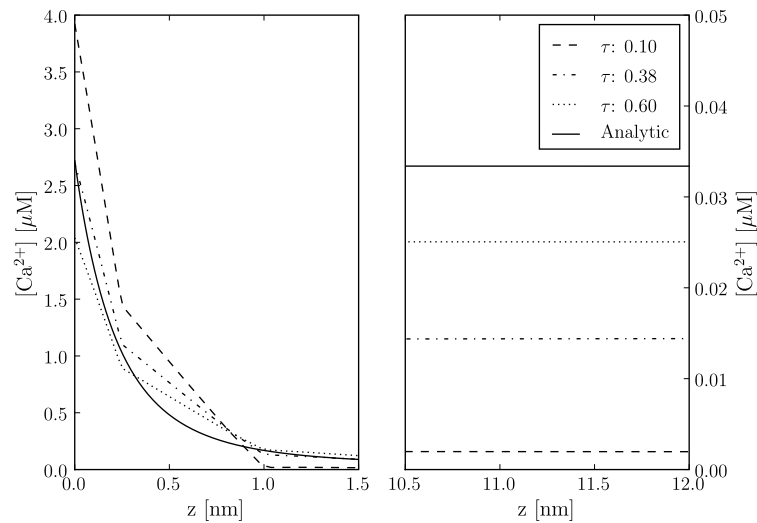


Figure 33.9: The figures shows the concentration traces of the numerical solutions from Mesh 3, see legend of Fig. 33.6, for three different τ together with the analytic solution. The traces in the two panels were picked from the same lines as the one in Fig. 33.8. Again we see from both panels that the solution with $\tau = 0.10$ give the poorest solution. The solution with $\tau = 0.38$ was the solution with smallest global error for this mesh, see Fig 33.6, and this is reflected in the good fit seen in the left panel, especially at $z = 0\text{nm}$. The solution with $\tau = 0.60$ undershoots the analytic solution at $z = 0$ with $\tilde{0}.7 \mu\text{M}$. From the right panel we see that all numerical solutions undershoot at $z = 15\text{nm}$, and the trace with $\tau = 0.60$ also here comes closest the analytic solution.

```

1 while not stop_sim:
2     # The next event
3     event = min(discrete_objects)
4     dt = event.next_time()
5
6     # Step the event and check result
7     while not event.step():
8         event = min(discrete_objects)
9         dt = event.next_time()
10
11    # Update the other discrete objects with dt
12    for obj in discrete_objects:
13        obj.update_time(dt)
14
15    # Solve the continuous equation
16    ca_field.solve(dt)
17    ca_field.send()
18
19    # Distribute the event
20    event.send()

```

Figure 33.10: Python-like pseudo code for the time stepping algorithm used in our simulator

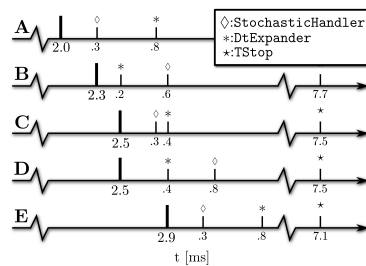


Figure 33.11: Diagram for the time stepping algorithm using 3 discrete objects: DtExpander, StochasticHandler, TStop. The values below the small ticks, corresponds to the time to the next event for each of the discrete objects. This time is measured from the last realized event, which is denoted by the thicker tick. In **A** we have realized a time event at $t=2.0$ ms. The next event to be realized is a stochastic transition, the one with smallest value below the ticks. In **B** this event is realized, and the StochasticHandler now show a new next event time. The event is a channel transition forcing the dt, controlled by the DtExpander, to be minimized. DtExpander now has the smallest next event time, and is realized in **C**. The channel transition that was realised in **B** raised the $[\text{Ca}^{2+}]$ in the cleft which in turn increase the Ca^{2+} dependent propensity functions in the included Markov models. The time to next event time of the StochasticHandler has therefore been updated, and moved forward in **C**. Also note that the DtExpander has expanded its next event time. In **D** the stochastic transition is realized and updated with a new next event time, but it is ignored as it is not a channel transition. The smallest time step is now the DtExpander, and this is realized in **E**. In this example we do not realize the TStop event as it is too far away.

```

1 from diffsim import *
2 from diffsim.dyadiccleft import *
3 from numpy import exp, fromfile
4
5 # Model parameters
6 c0_bulk = 0.1; D_Ca = 1.e5; Ds_cyt = 50; phi0 = -2.2; tau = 0.28
7 AP_offset = 0.1; dV = 0.5, ryr_scale = 100; end_sim_when_opepd = True
8
9 # Setting boundary markers
10 LCC_markers = range(10,14); RyR_markers = range(100,104); Cyt_marker = 3
11
12 # Add a diffusion domain
13 domain = DiffusionDomain("Dyadic_cleft", "cleft_mesh_with_RyR.xml.gz")
14 c0_vec = c0_bulk*exp(-VALENCE[Ca]*phi0*exp(-domain.mesh().coordinates()[:-1]))
15
16 # Add the ligand with fluxes
17 ligand = DiffusiveLigand(domain.name(), Ca, c0_vec, D_Ca)
18 field = StaticField("Bi_lipid_field", domain.name())
19 Ca_cyt = CytosolicStaticFieldFlux(field, Ca, Cyt_marker, c0_bulk, Ds_cyt)
20
21 # Adding channels with Markov models
22 for m in LCC_markers:
23     LCCVoltageDepFlux(domain.name(), m, activator=LCCMarkovModel_Greenstein)
24 for m in RyR_markers:
25     RyRMarkovModel_Stern("RyR_%d"%m, m, end_sim_when_opepd)
26
27 # Adding a dynamic voltage clamp that drives the LCC Markov model
28 AP_time = fromfile('AP_time_steps.txt', sep='\n')
29 dvc = DynamicVoltageClamp(AP_time, fromfile('AP.txt', sep='\n'), AP_offset, dV)
30
31 # Get and set parameters
32 params = get_params()
33
34 params.io.save_data = True
35 params.Bi_lipid_field.tau = tau
36 params.time.tstop = AP_time[-1] + AP_offset
37 params.RyRMarkovChain_Stern.scale = ryr_scale
38
39 info(str(params))
40
41 # Run 10 simulations
42 data = run_sim(10, "Dyadic_cleft_with_4_RyR_scale")
43 mean_release_latency = mean([ run["tstop"] for run in data["time"]])

```

Figure 33.12: An example of how diffsim can be used to simulate the time to RyR release latency, from a small dyad whos domain is defined by the mesh in the file `cleft_mesh_with_RyR.xml.gz`.

Electromagnetic Waveguide Analysis

By Evan Lezar and David B. Davidson

Chapter ref: [lezar]

► Editor note: *Reduce the number of macros.*

At their core, Maxwell's equations are a set of differential equations describing the interactions between electric and magnetic fields, charges, and currents. These equations provide the tools with which to predict the behaviour of electromagnetic phenomena, giving us the ability to use them in a wide variety of applications, including communication and power generation. Due to the complex nature of typical problems in these fields, numeric methods such as the finite element method are often employed to solve them.

One of the earliest applications of the finite element method in electromagnetics was in waveguide analysis [?]. Since waveguides are some of the most common structures in microwave engineering, especially in areas where high power and low loss are essential [?], their analysis is still a topic of much interest. This chapter considers the use of FEniCS in the cutoff and dispersion analysis of these structures as well as the analysis of waveguide discontinuities. These types of analysis form an important part of the design and optimisation of waveguide structures for a particular purpose.

The aim of this chapter is to guide the reader through the process followed in implementing solvers for various electromagnetic problems with both cutoff and dispersion analysis considered in depth. To this end a brief introduction of electromagnetic waveguide theory, the mathematical formulation of these problems, and the specifics of their solution using the finite element method are presented in 34.1. This lays the groundwork for a discussion of the details pertaining to the FEniCS implementation of these solvers, covered in 34.2. The translation of

the finite element formulation to FEniCS, as well as some post-processing considerations are covered. In 34.3 the solution results for three typical waveguide configurations are presented and compared to analytical or previously published data. This serves to validate the implementation and illustrates the kinds of problems that can be solved. Results for the analysis of H-plane waveguide discontinuities are then presented in 34.4 with two test cases being considered.

34.1 Formulation

As mentioned, in electromagnetics, the behaviour of the electric and magnetic fields are described by Maxwell's equations [?, ?]. Using these partial differential equations, various boundary value problems can be obtained depending on the problem being solved. In the case of time-harmonic fields, the equation used is the vector Helmholtz wave equation. If the problem is further restricted to a domain surrounded by perfect electrical or magnetic conductors (as is the case in general waveguide problems) the wave equation in terms of the electric field, \mathbf{E} , can be written as [?]

$$\nabla \times \frac{1}{\mu_r} \nabla \times \mathbf{E} - k_o^2 \epsilon_r \mathbf{E} = 0, \text{ in } \Omega, \quad (34.1)$$

subject to the boundary conditions

$$\hat{\mathbf{n}} \times \mathbf{E} = 0 \text{ on } \Gamma_e \quad (34.2)$$

$$\hat{\mathbf{n}} \times \nabla \times \mathbf{E} = 0 \text{ on } \Gamma_m, \quad (34.3)$$

with Ω representing the interior of the waveguide and Γ_e and Γ_m electric and magnetic walls respectively. μ_r and ϵ_r are the relative magnetic permeability and electric permittivity respectively. These are material parameters that may be position dependent but only the isotropic case is considered here. k_o is the operating wavenumber which is related to the operating frequency (f_o) by the expression

$$k_o = \frac{2\pi f_o}{c_0}, \quad (34.4)$$

with c_0 the speed of light in free space. This boundary value problem (BVP) can also be written in terms of the magnetic field [?], but as the discussions following are applicable to both formulations this will not be considered here.

If the guide is sufficiently long, and the z -axis is chosen parallel to its central axis as shown in Figure 34.1, then z -dependence of the electric field can be assumed to be of the form $e^{-\gamma z}$ with $\gamma = \alpha + j\beta$ a complex propagation constant [?, ?]. Making this assumption and splitting the electric field into transverse (\mathbf{E}_t) and axial ($\hat{\mathbf{z}}E_z$) components, results in the following expression for the field

$$\mathbf{E}(x, y, z) = [\mathbf{E}_t(x, y) + \hat{\mathbf{z}}E_z(x, y)]e^{-\gamma z}, \quad (34.5)$$

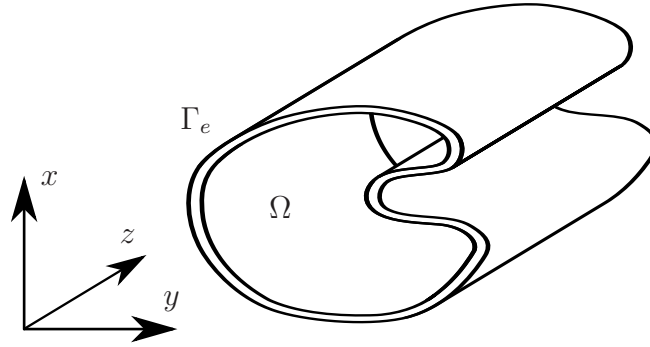


Figure 34.1: A long waveguide with an arbitrary cross-section aligned with the z -axis.

with x and y the Cartesian coordinates in the cross-sectional plane of the waveguide and z the coordinate along the length of the waveguide.

From (34.5) as well as the BVP described by (34.1), (34.2), and (34.3) it is possible to obtain the following variational functional found in many computational electromagnetic texts [?, ?]

$$F(\mathbf{E}) = \frac{1}{2} \int_{\Omega} \frac{1}{\mu_r} (\nabla_t \times \mathbf{E}_t) \cdot (\nabla_t \times \mathbf{E}_t) - k_o^2 \epsilon_r \mathbf{E}_t \cdot \mathbf{E}_t + \frac{1}{\mu_r} (\nabla_t E_z + \gamma \mathbf{E}_t) \cdot (\nabla_t E_z + \gamma \mathbf{E}_t) - k_o^2 \epsilon_r E_z E_z d\Omega, \quad (34.6)$$

with

$$\nabla_t = \frac{\partial}{\partial x} \hat{\mathbf{x}} + \frac{\partial}{\partial y} \hat{\mathbf{y}} \quad (34.7)$$

the transverse del operator.

A number of other approaches have also been taken to this problem. Some, for instance, involve only nodal based elements; some use the longitudinal fields as the working variable, and the problem has also been formulated in terms of potentials, rather than fields. A good summary of these may be found in [?, Chapter 9]. The approach used here, involving transverse and longitudinal fields, is probably the most widely used in practice.

34.1.1 Waveguide Cutoff Analysis

One of the simplest cases to consider, and often a starting point when testing a new finite element implementation, is waveguide cutoff analysis. When a waveguide is operating at cutoff, the electric field is uniform along the z -axis which corresponds with $\gamma = 0$ in (34.5) [?]. Substituting $\gamma = 0$ into (34.6) yields the

following functional

$$F(\mathbf{E}) = \frac{1}{2} \int_{\Omega} \frac{1}{\mu_r} (\nabla_t \times \mathbf{E}_t) \cdot (\nabla_t \times \mathbf{E}_t) - k_c^2 \epsilon_r \mathbf{E}_t \cdot \mathbf{E}_t + \frac{1}{\mu_r} (\nabla_t E_z) \cdot (\nabla_t E_z) - k_c^2 \epsilon_r E_z E_z d\Omega. \quad (34.8)$$

The symbol for the operating wavenumber k_o has been replaced with k_c , indicating that the quantity of interest is now the cutoff wavenumber. This quantity in addition to the field distribution at cutoff are of interest in these kinds of problems. Using two dimensional vector basis functions for the discretisation of the transverse field, and scalar basis functions for the axial components, the minimisation of (34.8) is equivalent to solving the following matrix equation

$$\begin{bmatrix} S_{tt} & 0 \\ 0 & S_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix} = k_c^2 \begin{bmatrix} T_{tt} & 0 \\ 0 & T_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix}, \quad (34.9)$$

which is in the form of a general eigenvalue problem. Here S_{ss} and T_{ss} represents the stiffness and mass common to finite element literature [?, ?] with the subscripts tt and zz indicating transverse or axial components respectively. The entries of the matrices of (34.9) are defined as

$$(s_{tt})_{ij} = \int_{\Omega} \frac{1}{\mu_r} (\nabla_t \times \mathbf{N}_i) \cdot (\nabla_t \times \mathbf{N}_j) d\Omega, \quad (34.10)$$

$$(t_{tt})_{ij} = \int_{\Omega} \epsilon_r \mathbf{N}_i \cdot \mathbf{N}_j d\Omega, \quad (34.11)$$

$$(s_{zz})_{ij} = \int_{\Omega} \frac{1}{\mu_r} (\nabla_t M_i) \cdot (\nabla_t M_j) d\Omega, \quad (34.12)$$

$$(t_{zz})_{ij} = \int_{\Omega} \epsilon_r M_i M_j d\Omega, \quad (34.13)$$

with $\int_{\Omega} d\Omega$ representing integration over the cross-section of the waveguide and \mathbf{N}_i and M_i representing the i^{th} vector and scalar basis functions respectively.

Due to the block nature of the matrices the eigensystem can be written as two smaller systems

$$[S_{tt}] \{e_t\} = k_{c,TE}^2 [T_{tt}] \{e_t\}, \quad (34.14)$$

$$[S_{zz}] \{e_z\} = k_{c,TM}^2 [T_{zz}] \{e_z\}, \quad (34.15)$$

with $k_{c,TE}$ and $k_{c,TM}$ corresponding to the cutoff wavenumbers of the transverse electric (TE) and transverse magnetic (TM) modes respectively. The eigenvectors ($\{e_t\}$ and $\{e_z\}$) of the systems are the coefficients of the vector and scalar basis functions, allowing for the calculation of the transverse and axial field distributions associated with a waveguide cutoff mode.

34.1.2 Waveguide Dispersion Analysis

In the case of cutoff analysis discussed in 34.1.1, one attempts to obtain the value of $k_o^2 = k_c^2$ for a given propagation constant γ , namely $\gamma = 0$. For most waveguide design applications however, k_o is specified and the propagation constant is calculated from the resultant eigensystem [?, ?]. This calculation can be simplified somewhat by making the following substitution into (34.6)

$$\mathbf{E}_{t,\gamma} = \gamma \mathbf{E}_t, \quad (34.16)$$

which results in the modified functional

$$F(\mathbf{E}) = \frac{1}{2} \int_{\Omega} \frac{1}{\mu_r} (\nabla_t \times \mathbf{E}_{t,\gamma}) \cdot (\nabla_t \times \mathbf{E}_{t,\gamma}) - k_o^2 \epsilon_r \mathbf{E}_{t,\gamma} \cdot \mathbf{E}_{t,\gamma} - \gamma^2 \left[\frac{1}{\mu_r} (\nabla_t E_z + \mathbf{E}_{t,\gamma}) \cdot (\nabla_t E_z + \mathbf{E}_{t,\gamma}) - k_o^2 \epsilon_r E_z E_z \right] d\Omega. \quad (34.17)$$

Using the same discretisation as for cutoff analysis discussed in 34.1.1, the matrix equation associated with the solution of the variational problem is given by

$$\begin{bmatrix} A_{tt} & 0 \\ 0 & 0 \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix} = \gamma^2 \begin{bmatrix} B_{tt} & B_{tz} \\ B_{zt} & B_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix}, \quad (34.18)$$

with

$$A_{tt} = S_{tt} - k_o^2 T_{tt}, \quad (34.19)$$

$$B_{zz} = S_{zz} - k_o^2 T_{zz}, \quad (34.20)$$

which is in the form of a generalised eigenvalue problem with the eigenvalues corresponding to the square of the complex propagation constant (γ).

The matrices S_{tt} , T_{tt} , S_{zz} , and T_{zz} are identical to those defined in 34.1.1 with entries given by (34.10), (34.11), (34.12), and (34.13) respectively. The entries of the other sub-matrices, B_{tt} , B_{tz} , and B_{zt} , are defined by

$$(b_{tt})_{ij} = \int_{\Omega} \frac{1}{\mu_r} \mathbf{N}_i \cdot \mathbf{N}_j d\Omega, \quad (34.21)$$

$$(b_{tz})_{ij} = \int_{\Omega} \frac{1}{\mu_r} \mathbf{N}_i \cdot \nabla_t M_j d\Omega, \quad (34.22)$$

$$(b_{zt})_{ij} = \int_{\Omega} \frac{1}{\mu_r} \nabla_t M_i \cdot \mathbf{N}_j d\Omega. \quad (34.23)$$

A common challenge in electromagnetic eigenvalue problems such as these is the occurrence of spurious modes [?]. These are non-physical modes that fall in the null space of the $\nabla \times \nabla \times$ operator of (34.1) [?] (The issue of spurious modes is not as closed as most computational electromagnetics texts indicate. For a

summary of recent work in the applied mathematics literature, written for an engineering readership, see [?]).

One of the strengths of the vector basis functions used in the discretisation of the transverse component of the field is that it allows for the identification of these spurious modes [?, ?]. In [?] a scaling method is proposed to shift the eigenvalue spectrum such that the dominant waveguide mode (usually the lowest non-zero eigenvalue) corresponds with the largest eigenvalue of the new system. Other approaches have also been followed to address the spurious modes. In [?], Lagrange multipliers are used to move these modes from zero to infinity.

In the case of the eigensystem associated with dispersion analysis, the matrix equation of (34.18) is scaled as follows

$$\begin{bmatrix} B_{tt} & B_{tz} \\ B_{zt} & B_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix} = \frac{\theta^2}{\theta^2 + \gamma^2} \begin{bmatrix} B_{tt} + \frac{A_{tt}}{\theta^2} & B_{tz} \\ B_{zt} & B_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix}, \quad (34.24)$$

with $\theta^2 = k_o^2 \mu_r^{(max)} \epsilon_r^{(max)}$ an upper bound on the square of the propagation constant (γ^2) and $\mu_r^{(max)}$ and $\epsilon_r^{(max)}$ the maximum relative permeability and permittivity in the computational domain.

If λ is an eigenvalue of the scaled system of (34.24), then the propagation constant can be calculated as

$$\gamma^2 = \frac{1 - \lambda}{\lambda} \theta^2, \quad (34.25)$$

and thus $\gamma^2 \rightarrow \infty$ as $\lambda \rightarrow 0$, which moves the spurious modes out of the region of interest.

34.2 Implementation

This section considers the details of the implementation of a FEniCS-based solver for waveguide cutoff mode and dispersion curve problems as described in 34.1.1 and 34.1.2. A number of code snippets illustrate some of the finer points of the implementation.

34.2.1 Formulation

Listing 34.1 shows the definitions of the function spaces used in the solution of the problems considered here. Nédélec basis functions of the first kind (N_{v} and N_{u}) are used to approximate the transverse component of the electric field. This ensures that the tangential continuity required at element and material boundaries can be enforced [?]. The axial component of the field is modelled using a set of Lagrange basis functions (M_{v} , and M_{u}). Additionally, a discontinuous Galerkin function space is included to allow for the modelling of material parameters such as dielectrics.

Listing 34.1: Function spaces and basis functions.

```

V_DG = FunctionSpace(mesh, "DG", 0) V_N = FunctionSpace(mesh,
"Nedelec", transverse_order) V_M = FunctionSpace(mesh, "Lagrange",
axial_order)

combined_space = V_N + V_L

(N_v, M_v) = TestFunctions(combined_space)
(N_u, M_u) = TrialFunctions(combined_space)

```

In order to deal with material properties, the `Function` class is extended and the `eval()` method overridden. This is illustrated in Listing 34.2 where a dielectric with a relative permittivity of $\epsilon_r = 4$ that extends to $y = 0.25$ is shown. This class is then instantiated using the discontinuous Galerkin function space already discussed. For constant material properties (such as the inverse of the magnetic permeability μ_r , in this case) a JIT-compiled function is used.

Listing 34.2: Material properties and functions.

```

class HalfLoadedDielectric(Function):
    def eval(self, values, x):
        if x[1] < 0.25:
            values[0] = 4.0
        else:
            values[0] = 1.0;

e_r = HalfLoadedDielectric(V_DG)
one_over_u_r = Function(V_DG, "1.0")

k_o_squared = Function(V_DG, "value", {"value" : 0.0})
theta_squared = Function(V_DG, "value", {"value" : 0.0})

```

The basis functions declared in Listing 34.1 and the desired material property functions are now used to create the forms required for the matrix entries specified in 34.1.1 and 34.1.2. The forms are shown in Listing 34.3 and the matrices of (34.9), (34.18), and (34.24) can be assembled using the required combinations of these forms with the right hand side of (34.24), `rhs`, provided as an example. It should be noted that the use of JIT-functions for operating wavenumber and scaling parameters means that the forms need not be recompiled each time the operating frequency is changed. This is especially beneficial when the calculation of dispersion curves is considered since the same calculation is performed for a range of operating frequencies.

From (34.2) it follows that the tangential component of the electric field must be zero on perfectly electrical conducting (PEC) surfaces. What this means in practice is that the degrees of freedom associated with both the Lagrange and Nédélec basis functions on the boundary must be set to zero since there can be no electric field inside a perfect electrical conductor [?]. An example for a PEC

surface surrounding the entire computational domain is shown in Listing 34.4 as the `ElectricWalls` class. This boundary condition can then be applied to the constructed matrices before solving the eigenvalue systems.

The boundary condition given in (34.3) results in a natural boundary condition for the problems considered and thus it is not necessary to explicitly enforce it [?]. Such magnetic walls and the symmetry of a problem are often used to decrease the size of the computational domain although this does limit the solution obtained to even modes [?].

Once the required matrices have been assembled and the boundary conditions applied, the resultant eigenproblem can be solved. This can be done by outputting the matrices and solving the problem externally, or by making use of the eigensolvers provided by SLEPc that can be integrated into the FEniCS package.

34.2.2 Post-Processing

After the eigenvalue system has been solved and the required eigenpair chosen, this can be post-processed to obtain various quantities of interest. For the cutoff wavenumber, this is a relatively straight-forward process and only involves simple operations on the eigenvalues of the system. For the calculation of dispersion curves and visualisation of the resultant field components the process is slightly more complex.

Dispersion Curves

For dispersion curves the computed value of the propagation constant (γ) is plotted as a function of the operating frequency (f_o). Since γ is a complex variable, a mapping is required to represent the data on a single two-dimensional graph. This is achieved by choosing the f_o -axis to represent the value $\gamma = 0$, effectively

Listing 34.3: Forms for matrix entries.

```
s_tt = one_over_u_r*dot(curl_t(N_v), curl_t(N_u))
t_tt = e_r*dot(N_v, N_u)

s_zz = one_over_u_r*dot(grad(M_v), grad(M_u))
t_zz = e_r*M_v*M_u

b_tt = one_over_u_r*dot(N_v, N_u)
b_tz = one_over_u_r*dot(N_v, grad(M_u))
b_zt = one_over_u_r*dot(grad(M_v), N_u)

a_tt = s_tt - k_o_squared*t_tt
b_zz = s_zz - k_o_squared*t_zz

rhs = b_tt + b_tz + b_zt + b_zz + 1.0/theta_squared*a_tt
```

Listing 34.4: Boundary conditions.

```
class ElectricWalls(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary

dirichlet_bc = DirichletBC(combined_space,
    Function(combined_space, "0.0"), ElectricWalls())
```

dividing the $\gamma - f_o$ plane into two regions. The region above the f_o -axis is used to represent the magnitude of the imaginary part of γ , whereas the real part falls in the lower region. A mode that propagates along the guide for a given frequency will thus lie in the upper half-plane of the plot and a complex mode will be represented by a data point above and below the f_o -axis. This procedure is followed in [?] and other literature and allows for quick comparisons and validation of results.

Field Visualisation

In order to visualise the fields associated with a given solution, the basis functions need to be weighted with coefficients corresponding to the entries in an eigenvector obtained from one of the eigenvalue problems. In addition, the transverse or axial components of the field may need to be extracted. An example for plotting the transverse and axial components of the field is given in Listing 34.5. Here the variable \mathbf{x} assigned to the function vector is one of the eigenvectors obtained by solving the eigenvalue problem.

Listing 34.5: Extraction and visualisation of transverse and axial field components.

```
f = Function(combined_space) f.vector().assign(x)

(transverse, axial) = f.split()

plot(transverse)
plot(axial)
```

The `eval()` method of the `transverse` and `axial` functions can also be called in order to evaluate the functions at a given spatial coordinate, allowing for further visualisation or post-processing options.

34.3 Examples

The first of the examples considered is the canonical one of a hollow waveguide, which has been covered in a multitude of texts on the subject [?, ?, ?, ?]. Since

the analytical solutions for this structure are known, it provides an excellent benchmark and is a typical starting point for the validation of a computational electromagnetic solver for solving waveguide problems.

The second and third examples are a partially filled rectangular guide and a shielded microstrip line on a dielectric substrate, respectively. In each case results are compared to published results from the literature as a means of validation.

34.3.1 Hollow Rectangular Waveguide

Figure 34.2 shows the cross section of a hollow rectangular waveguide. For the purpose of this chapter a guide with dimensions $a = 1\text{m}$ and $b = 0.5\text{m}$ is considered. The analytical expressions for the electric field components of a hollow

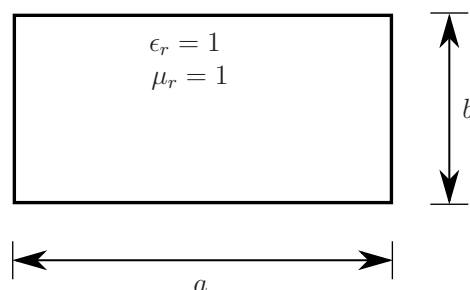


Figure 34.2: A diagram showing the cross section and dimensions of a hollow rectangular waveguide.

rectangular guide with width a and height b are given by [?]

$$E_x = \frac{n}{b} A_{mn} \cos\left(\frac{m\pi x}{a}\right) \sin\left(\frac{n\pi y}{b}\right), \quad (34.26)$$

$$E_y = \frac{-m}{a} A_{mn} \sin\left(\frac{m\pi x}{a}\right) \cos\left(\frac{n\pi y}{b}\right), \quad (34.27)$$

for the TE_{mn} mode, whereas the z -directed electric field for the TM_{mn} mode has the form [?]

$$E_z = B_{mn} \sin\left(\frac{m\pi x}{a}\right) \sin\left(\frac{n\pi y}{b}\right), \quad (34.28)$$

with A_{mn} and B_{mn} constants for a given mode. In addition, the propagation constant, γ , has the form

$$\gamma = \sqrt{k_o^2 - k_c^2}, \quad (34.29)$$

with k_o the operating wavenumber dependent on the operating frequency, and

$$k_c^2 = \left(\frac{m\pi}{a}\right)^2 + \left(\frac{n\pi}{b}\right)^2, \quad (34.30)$$

the analytical solution for the square of the cutoff wavenumber for both the TE_{mn} and TM_{mn} modes.

Cutoff Analysis

Figure 34.3 shows the first two calculated TE cutoff modes for the hollow rectangular guide, with the first two TM cutoff modes being shown in Figure 34.4. The solution is obtained with 64 triangular elements and second order basis functions in the transverse as well as the axial discretisations.

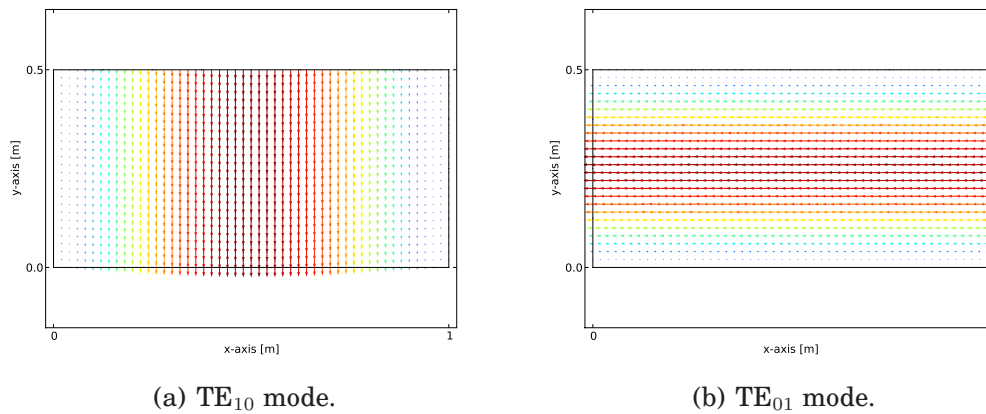


Figure 34.3: The first two calculated TE cutoff modes of a 1 m \times 0.5 m hollow rectangular waveguide.

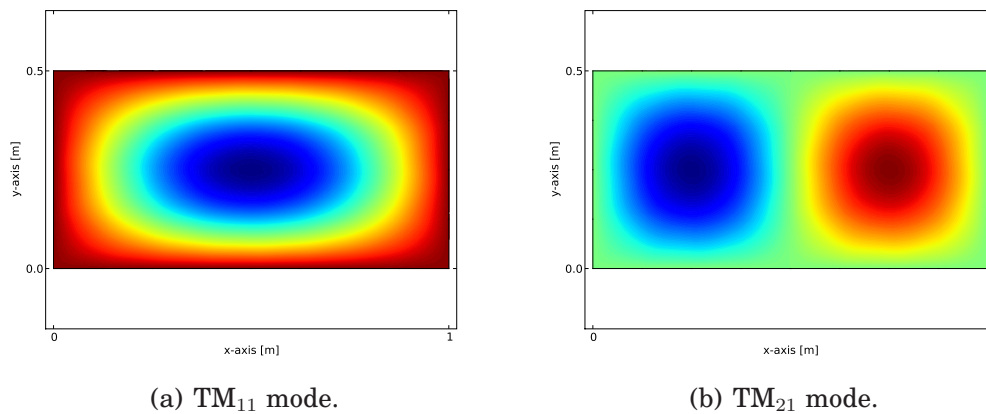


Figure 34.4: The first two calculated TM cutoff modes of a 1 m \times 0.5 m hollow rectangular waveguide.

Table 34.1 gives a comparison of the calculated and analytical values for the square of the cutoff wavenumber of a number of modes for a hollow rectangular guide. As can be seen from the table, there is excellent agreement between the values.

Table 34.1: Comparison of analytical and calculated cutoff wavenumber squared (k_c^2) for various TE and TM modes of a $1\text{ m} \times 0.5\text{ m}$ hollow rectangular waveguide.

Mode	Analytical [m^{-2}]	Calculated [m^{-2}]	Relative Error
TE ₁₀	9.8696	9.8696	1.4452e-06
TE ₀₁	39.4784	39.4784	2.1855e-05
TE ₂₀	39.4784	39.4784	2.1894e-05
TM ₁₁	49.3480	49.4048	1.1514e-03
TM ₂₁	78.9568	79.2197	3.3295e-03
TM ₃₁	128.3049	129.3059	7.8018e-03

Dispersion Analysis

When considering the calculation of the dispersion curves for the hollow rectangular waveguide, the mixed formulation as discussed in 34.1.2 is used. The calculated dispersion curves for the first 10 modes of the hollow rectangular guide are shown in Figure 34.5 along with the analytical results. For the rectangular guide a number of modes are degenerate with the same dispersion and cutoff properties as predicted by (34.29) and (34.30). This explains the visibility of only six curves. There is excellent agreement between the analytical and computed results.

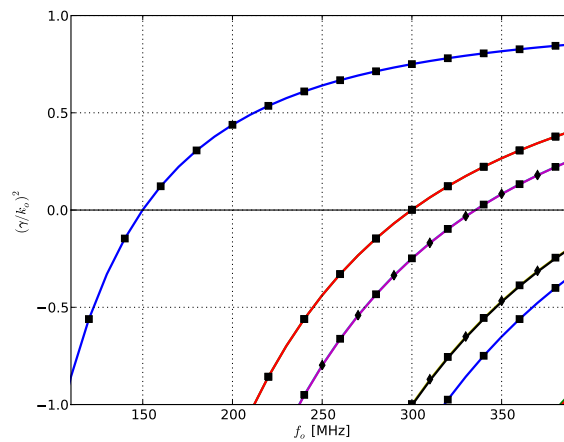


Figure 34.5: Dispersion curves for the first 10 modes of a 1 m × 0.5 m hollow rectangular waveguide. Markers are used to indicate the analytical results with ■ and ◆ indicating TE and TM modes respectively.

34.3.2 Half-Loaded Rectangular Waveguide

In some cases, a hollow rectangular guide may not be the ideal structure to use due to, for example, limitations on its dimensions. If the guide is filled with a dielectric material with a relative permittivity $\epsilon_r > 1$, the cutoff frequency of the dominant mode will be lowered. Consequently a loaded waveguide will be mode compact than a hollow guide for the same dominant mode frequency. Furthermore, in many practical applications, such as impedance matching or phase shifting sections, a waveguide that is only partially loaded is used [?].

Figure 34.6 shows the cross section of such a guide. The guide considered here has the same dimensions as the hollow rectangular waveguide used in the previous section, but its lower half is filled with an $\epsilon_r = 4$ dielectric material.

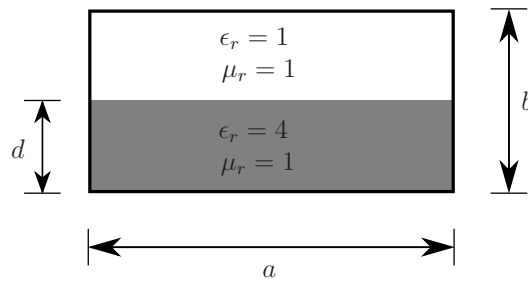


Figure 34.6: A diagram showing the cross section and dimensions of a half-loaded rectangular waveguide. The lower half of the guide is filled with an $\epsilon_r = 4$ dielectric material.

Cutoff Analysis

Figure 34.7 shows the first TE and TM cutoff modes of the half-loaded guide shown in Figure 34.6. Note the concentration of the transverse electric field in the hollow part of the guide. This is due to the fact that the displacement flux, $D = \epsilon E$, must be normally continuous at the dielectric interface [?, ?].

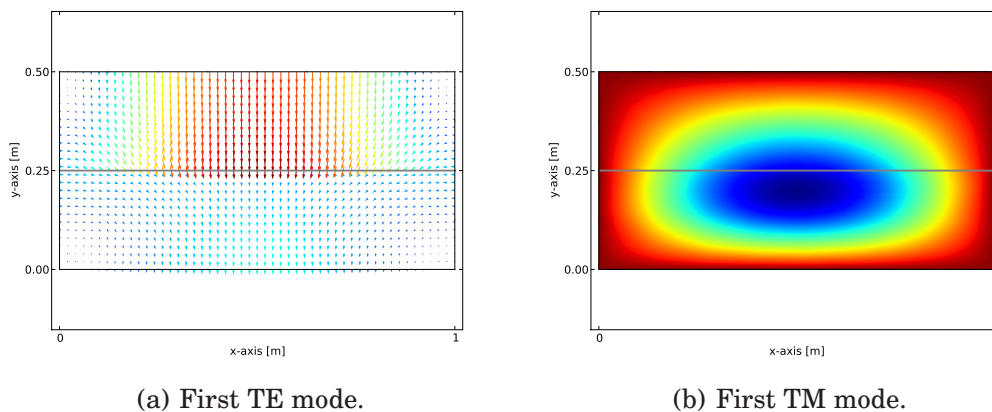


Figure 34.7: The first calculated TE and TM cutoff modes of a $1 \text{ m} \times 0.5 \text{ m}$ rectangular waveguide with the lower half of the guide filled with an $\epsilon_r = 4$ dielectric.

Dispersion Analysis

The dispersion curves for the first 8 modes of the half-loaded waveguide are shown in Figure 34.8 with results for the first 4 modes from [?] provided as reference. Here it can be seen that the cutoff frequency of the dominant mode has decreased and there is no longer the same degeneracy in the modes when compared to the hollow guide of the same dimensions. In addition, there are complex

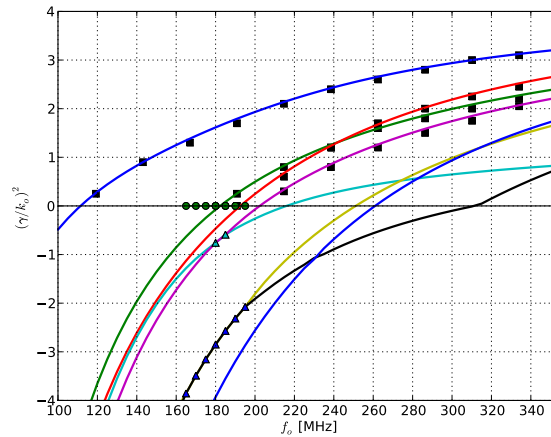


Figure 34.8: Dispersion curves for the first 8 modes of a 1 m \times 0.5 m rectangular waveguide with its lower half filled with an $\epsilon_r = 4$ dielectric material. Reference values for the first 4 modes from [?] are shown as ■. The presence of complex mode pairs are indicated by ▲ and ●.

modes present as a result of the fourth and fifth as well as the sixth and seventh modes occurring as conjugate pairs at certain points in the spectrum. It should be noted that the imaginary parts of these conjugate pairs are very small and thus the ● markers in Figure 34.8 appear to fall on the f_o -axis. These complex modes are discussed further in 34.3.3.

34.3.3 Shielded Microstrip

Microstrip line is a very popular type of planar transmission line, primarily due to the fact that it can be constructed using photolithographic processes and integrates easily with other microwave components [?]. Such a structure typically consists of a thin conducting strip on a dielectric substrate above a ground plane. In addition, the strip may be shielded by enclosing it in a PEC box to reduce electromagnetic interference. A cross section of a shielded microstrip line is shown in Figure 34.9 with the thickness of the strip, t , exaggerated for clarity. The dimensions used to obtain the results discussed here are given in Table 34.2.

Table 34.2: Dimensions for the shielded microstrip line considered here. Definitions for the symbols are given in Figure 34.9.

	Dimension [mm]
a, b	12.7
d, w	1.27
t	0.127

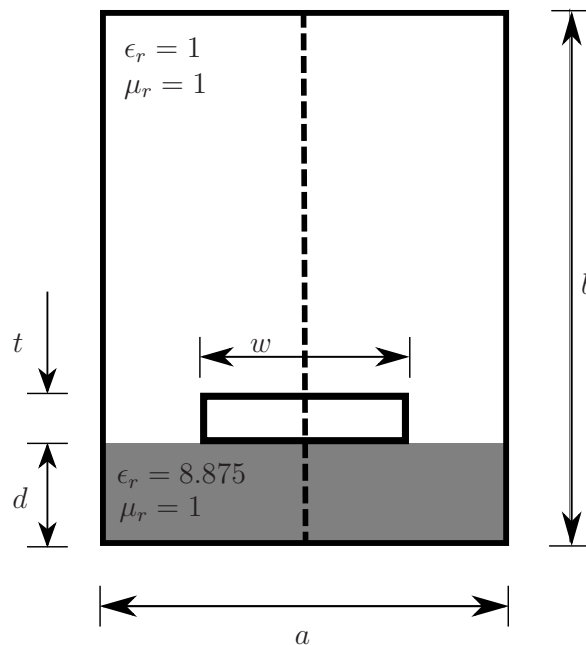


Figure 34.9: A diagram showing the cross section and dimensions of a shielded microstrip line. The microstrip is etched on a dielectric material with a relative permittivity of $\epsilon_r = 8.75$. The plane of symmetry is indicated by a dashed line and is modelled as a magnetic wall in order to reduce the size of the computational domain.

Cutoff Analysis

Since the shielded microstrip structure consists of two conductors, it supports a dominant transverse electromagnetic (TEM) wave that has no axial component of the electric or magnetic field [?]. Such a mode has a cutoff wavenumber of zero and thus propagates for all frequencies [?, ?]. The cutoff analysis of this structure is not considered here explicitly. The cutoff wavenumbers for the higher order modes (which are hybrid TE-TM modes [?]) can however be determined from the dispersion curves by the intersection of a curve with the f_o -axis.

Dispersion Analysis

The dispersion analysis presented in [?] is repeated here for validation with the resultant curves shown in Figure 34.10. As is the case with the half-loaded guide, the results calculated with FEniCS agree well with previously published results. In the figure it is shown that for certain parts of the frequency range of interest, modes six and seven have complex propagation constants. Since the matrices in the eigenvalue problem are real valued, the complex eigenvalues – and thus the propagation constants – must occur in complex conjugate pairs as is the case here and reported earlier in [?]. These conjugate propagation constants are

associated with two equal modes propagating in opposite directions along the waveguide and thus resulting in zero energy transfer. It should be noted that for lossy materials (not considered here), complex modes are expected but do not necessarily occur in conjugate pairs [?].

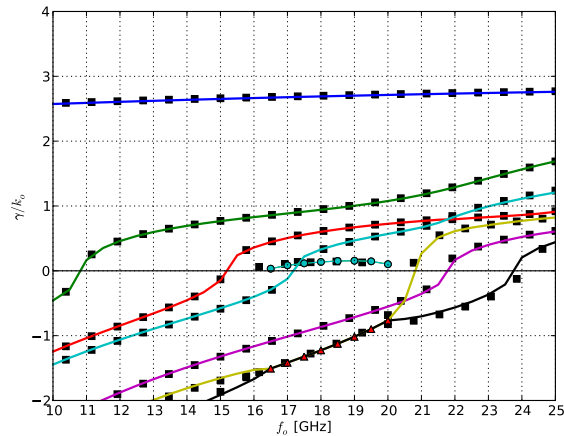


Figure 34.10: Dispersion curves for the first 7 even modes of shielded microstrip line using a magnetic wall to enforce symmetry. Reference values from [?] are shown as ■. The presence of complex mode pairs are indicated by ▲ and ●.

34.4 Analysis of Waveguide Discontinuities

Although this chapter focuses on eigenvalue type problems related to waveguides, the use of FEniCS in waveguide analysis is not limited to such problems. This section briefly introduces the solution of problems relating to waveguide discontinuities as an additional problem class. Applications where the solutions of these problems are of importance to microwave engineers is the design of waveguide filters as well as the analysis and optimisation of bends in a waveguide where properties such as the scattering parameters (S-parameters) of the device are calculated [?].

The hybrid finite element-modal expansion technique discussed in [?] is implemented and used to solve problems related to H-plane waveguide discontinuities. For such cases – which are uniform in the vertical (y) direction transverse to the direction of propagation (z) – the problem reduces to a scalar one in two dimensions [?] with the operating variable the y -component of the electric field in the guide. In order to obtain the scattering parameters at each port of the device, the field on the boundary associated with the port is written as a summation of the tangential components of the incoming and outgoing waveguide modes. These modes can either be computed analytically, when a junction is

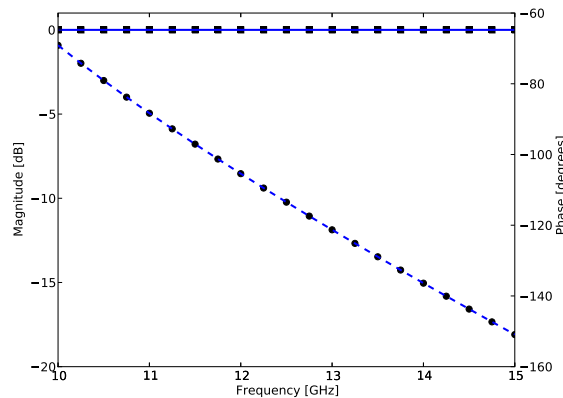


Figure 34.11: Magnitude (solid line) and phase (dashed line) of the transmission coefficient (S_{21}) of a length of rectangular waveguide with dimensions $a = 18.35\text{mm}$, $b = 9.175\text{mm}$, and $l = 10\text{mm}$. The analytical results for the same structure are indicated by markers with \blacksquare and \bullet indicating the magnitude and phase respectively.

rectangular for example, or calculated with methods such as those discussed in the preceding sections [?].

Transmission parameter (S_{21}) results for a length of hollow rectangular waveguide are shown in Figure 34.11. As expected, the length of guide behaves as a fixed value phase shifter [?] and the results obtained show excellent agreement with the analytical ones.

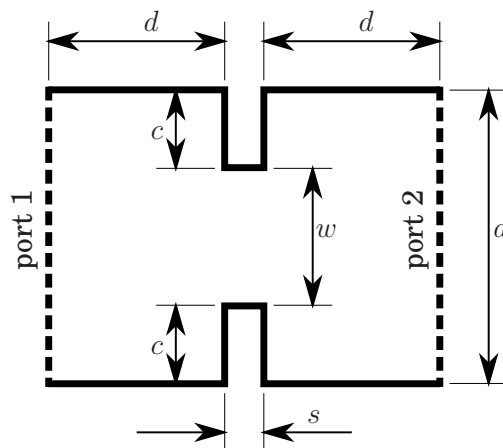


Figure 34.12: Schematic of an H-plane iris in a rectangular waveguide dimensions: $a = 18.35\text{mm}$, $c = 4.587\text{mm}$, $d = 1\text{mm}$, $s = 0.5\text{mm}$, and $w = 9.175\text{mm}$. The guide has a height of $b = 9.175\text{mm}$. The ports are indicated by dashed lines on the boundary of the device.

A schematic for a more interesting example is the H-plane iris shown in Figure 34.12. The figure shows the dimensions of the structure and indicates the port definitions. The boundaries indicated by a solid line is a PEC material. The magnitude and phase results for the S-parameters of the device are given in Figure 34.13 and compared to the results published in [?] with good agreement between the two sets of data.

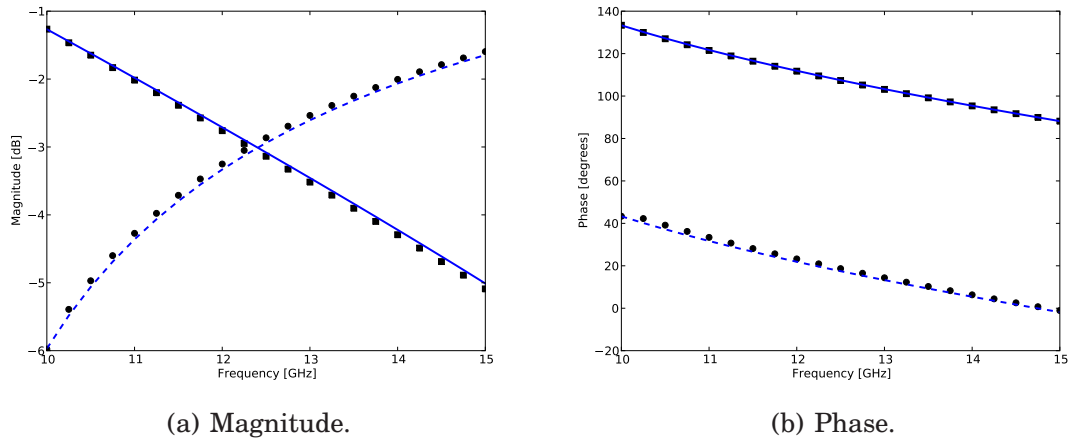


Figure 34.13: Results for the magnitude and phase of the reflection coefficient (S_{11} – solid line) and transmission coefficient (S_{21} – dashed line) of an H-plane iris in a rectangular waveguide shown in Figure 34.12. Reference results from [?] are indicated by markers with \blacksquare and \bullet indicating the reflection and transmission coefficient respectively.

34.5 Conclusion

In this chapter, the solutions of cutoff and dispersion problems associated with electromagnetic waveguiding structures have been implemented and the results analysed. In all cases, the results obtained agree well with previously published or analytical results. This is also the case where the analysis of waveguide discontinuities are considered, and although the solutions shown here are restricted to H-plane waveguide discontinuities, the methods applied are applicable to other classes of problems such as E-plane junctions and full 3D analysis.

This chapter has also illustrated the ease with which complex formulations can be implemented and how quickly solutions can be obtained. This is largely due to the almost one-to-one correspondence between the expressions at a formulation level and the high-level code that is used to implement a particular solution. Even in cases where the required functionality is limited or missing, the use of FEniCS in conjunction with external packages greatly reduces development time.

CHAPTER 35

Applications in Solid Mechanics

By Kristian B. Ølgaard and Garth N. Wells

Chapter ref: **[oelgaard-1]**

Summarise work on automated modelling for solid mechanics, with application to hyperelasticity, plasticity and strain gradient dependent models. Special attention will be paid the linearisation of function which do come from a finite element space.

CHAPTER 36

Modelling Evolving Discontinuities

By Mehdi Nikbakht and Garth N. Wells

Chapter ref: **[nikbakht]**

Summarise work on automated modelling of PDEs with evolving discontinuities, e.g. cracks.

CHAPTER 37

Optimal Control Problems

By Kent-Andre Mardal, Oddrun Christine Myklebust and Bjørn Fredrik Nielsen

Chapter ref: **[mardal-3]**

This chapter is devoted to the study of several optimal control problems and their implementation in FEniCS. We start with an abstract description of optimal control problems before we apply the abstract theory to several concrete examples. We also discuss the construction of block preconditioners for efficient solution of these problems.

Automatic Calibration of Depositional Models

By Hans Joachim Schroll

Chapter ref: [**schroll**]

A novel concept for calibrating depositional models is presented. In this approach transport coefficients are determined from well output measurements. Finite element implementation of the multi-lithology models and their duals is automated by the FEniCS project DOLFIN using a python interface.

38.1 Issues in dual lithology sedimentation

Different types of forward computer models are being used by sedimentologists and geomorphologists to simulate the process of sedimentary deposition over geological time periods. The models can be used to predict the presence of reservoir rocks and stratigraphic traps at a variety of scales. State-of-the-art advanced numerical software provides accurate approximations to the mathematical model, which commonly is expressed in terms of a nonlinear diffusion dominated PDE system. The potential of today's simulation software in industrial applications is limited however, due to major uncertainties in crucial material parameters that combine a number of physical phenomena and therefore are difficult to quantify. Examples of such parameters are diffusive transport coefficients.

The idea in this contribution is to calibrate uncertain transport coefficients to direct observable data, like well measurements from a specific basin. In this approach the forward evolution process, mapping data to observations, is reversed to determine the data, i.e. transport coefficients. Mathematical tools and numerical algorithms are applied to automatically calibrate geological models to

actual observations — a critical but so far missing link in forward depositional modeling.

Automatic calibration, in combination with stochastic modeling, will boost the applicability and impact of modern numerical simulations in industrial applications.

38.2 A multidimensional sedimentation model

Submarine sedimentation is an evolution process. By flow into the basin, sediments build up and evolve in time. The evolution follows geophysical laws, expressed as diffusive PDE models. The following system is a multidimensional version of the dual lithology model by Rivenæs [?, ?]

$$\begin{pmatrix} A & s \\ -A & 1-s \end{pmatrix} \begin{pmatrix} s \\ h \end{pmatrix}_t = \nabla \cdot \begin{pmatrix} \alpha s \nabla h \\ \beta (1-s) \nabla h \end{pmatrix} \quad \text{in } [0, T] \times B . \quad (38.1)$$

Here h denotes the thickness of a layer of deposit and s models the volume fraction for the sand lithology. Consequently, $1-s$ is the fraction for mud. The system is driven by fluxes anti proportional to the flow rates $s \nabla h$ and $(1-s) \nabla h$ resulting in a diffusive, but incompletely parabolic, PDE system. The domain of the basin is denoted by B . Parameters in the model are: The transport layer thickness A and the diffusive transport coefficients α, β .

For a forward in time simulation, the system requires initial and boundary data. At initial time, the volume fraction s and the layer thickness h need to be specified. According to geologists, such data can be reconstructed by some kind of “back stripping”. Along the boundary of the basin, the flow rates $s \nabla h$ and $(1-s) \nabla h$ are given.

38.3 An inverse approach

The parameter-to-observation mapping $R : (\alpha, \beta) \mapsto (s, h)$ is commonly referred to as the forward problem. In a basin direct observations are only available at wells. Moreover, from the age of the sediments, their history can be reconstructed. Finally, well-data is available in certain well areas $W \subset B$ and backward in time.

The objective of the present investigation is to determine transport coefficients from observed well-data and in that way, to calibrate the model to the data. This essentially means to invert the parameter-to-observation mapping. Denoting observed well-data by (\tilde{s}, \tilde{h}) , the goal is to minimize the output functional

$$J(\alpha, \beta) = \frac{1}{|W|} \int_0^T \int_W (\tilde{s} - s)^2 + (\tilde{h} - h)^2 \, dx \, dt \quad (38.2)$$

with respect to the transport coefficients α and β .

In contrast to the "direct inversion" as described by Imhof and Sharma [?], which is considered impractical, we do not propose to invert the time evolution of the diffusive depositional process. We actually use the forward-in-time evolution of sediment layers in a number of wells to calibrate transport coefficients. Via the calibrated model we can simulate the basin and reconstruct its historic evolution. By computational mathematical modeling, the local data observed in wells determines the evolution throughout the entire basin.

38.4 The Landweber algorithm

In a slightly more abstract setting, the task is to minimize an objective functional J which implicitly depends on the parameters p via u subject to the constraint that u satisfies some PDE model; a PDE constrained minimization problem: Find p such that $J(p) = J(u(p)) = \min$ and $\text{PDE}(u, p) = 0$.

Landweber's steepest decent algorithm [?] iterates the following sequence until convergence:

1. Solve $\text{PDE}(u^k, p^k) = 0$ for u^k .
2. Evaluate $d^k = -\nabla_p J(p^k) / \|\nabla_p J(p^k)\|$.
3. Update $p^{k+1} = p^k + \Delta p^k d^k$.

Note that the search direction d^k , the negative gradient, is the direction of steepest decent. To avoid scale dependence, the search direction is normed.

The increment Δp^k is determined by a one dimensional line search algorithm, minimizing a locally quadratic approximation to J along the line $p^k + \gamma d^k$, $\gamma \in \mathbb{R}$. We use the ansatz

$$J(p^k + \gamma d^k) = a\gamma^2 + b\gamma + c, \quad \gamma \in \mathbb{R}.$$

The extreme value of this parabola is located at

$$\gamma^k = -\frac{b}{2a}. \tag{38.3}$$

To determine $\Delta p^k = \gamma^k$, the parabola is fitted to the local data. For example b is given by the gradient

$$J_\gamma(p^k) = b = \nabla J(p^k) \cdot d^k = -\|\nabla J(p^k)\|^2.$$

To find a , another gradient of J along the line $p^k + \gamma d^k$ is needed. To avoid an extra evaluation, we project $p^{k-1} = p^k - \gamma^{k-1} d^{k-1}$ onto the line $p^k + \gamma d^k$, that is $\pi(p^{k-1}) = p^k - \gamma^* d^k$ and approximate the directional derivative

$$J_\gamma(p^k - \gamma^* d^k) \approx \nabla J(p^{k-1}) \cdot d^k. \tag{38.4}$$

Note that this approximation is exact if two successive directions d^{k-1} and d^k are in line. Elementary geometry yields $\gamma^* = \gamma^{k-1} \cos \varphi$ and $\cos \varphi = d^{k-1} \cdot d^k$. Thus $\gamma^* = \gamma^{k-1} \cdot d^{k-1} \cdot d^k$. From $J_\gamma(p^k - \gamma^* d^k) = -2a\gamma^* + b$ and (38.4) we find

$$-2a = \frac{(\nabla J(p^{k-1}) - \nabla J(p^k)) \cdot d^k}{\gamma^{k-1} \cdot d^{k-1} \cdot d^k} .$$

Finally, the increment (38.3) evaluates as

$$\Delta p^k = \gamma^k = \frac{\nabla J(p^k) \cdot \nabla J(p^{k-1})}{\nabla J(p^k) \cdot \nabla J(p^{k-1}) - \|\nabla J(p^k)\|^2} \cdot \frac{\nabla J(p^k)}{\nabla J(p^{k-1})} \cdot d^{k-1} .$$

38.5 Evaluation of gradients by duality arguments

Every single step of Landweber's algorithm requires the simulation of a time dependent, nonlinear PDE system and the evaluation of the gradient of the objective functional. The most common approach to numerical derivatives, via finite differences, is impractical for complex problems: Finite difference approximation would require to perform $n + 1$ forward simulations in n parameter dimensions. Using duality arguments however, n nonlinear PDE systems can be replaced by one linear, dual problem. After all, J is evaluated by one forward simulation of the nonlinear PDE model and the complete gradient ∇J is obtained by one (backward) simulation of the linear, dual problem. Apparently, one of the first references to this kind of duality arguments is [?].

The concept is conveniently explained for a scalar diffusion equation

$$u_t = \nabla \cdot (\alpha \nabla u) .$$

As transport coefficients may vary throughout the basin, we allow for a piecewise constant coefficient

$$\alpha = \begin{cases} \alpha_1 & x \in B_1 \\ \alpha_2 & x \in B_2 \end{cases} .$$

Assuming no flow along the boundary and selecting a suitable test function ϕ , the equation in variational form reads

$$\mathcal{A}(u, \phi) := \int_0^T \int_B u_t \phi + \alpha \nabla u \cdot \nabla \phi dx dt = 0 .$$

Taking an derivative $\partial \alpha_i$, $i = 1, 2$ under the integral sign, we find

$$\mathcal{A}(u_{\alpha_i}, \phi) = \int_0^T \int_B u_{\alpha_i, t} \phi + \alpha \nabla u_{\alpha_i} \cdot \nabla \phi dx dt = - \int_0^T \int_{B_i} \nabla u \cdot \nabla \phi dx dt . \quad (38.5)$$

The corresponding derivative of the output functional $J = \int_0^T \int_W (u-d)^2 dxdt$ reads

$$J_{\alpha_i} = 2 \int_0^T \int_W (u-d)u_{\alpha_i} dxdt \quad , \quad i = 1, 2 \quad .$$

The trick is to define a dual problem

$$\mathcal{A}(\phi, \omega) = 2 \int_0^T \int_W (u-d)\phi dxdt$$

such that $\mathcal{A}(u_{\alpha_i}, \omega) = J_{\alpha_i}$ and by using the dual solution ω in (38.5)

$$\mathcal{A}(u_{\alpha_i}, \omega) = J_{\alpha_i} = - \int_0^T \int_{B_i} \nabla u \cdot \nabla \omega dxdt \quad , \quad i = 1, 2 \quad .$$

In effect, the desired gradient $\nabla J = (J_{\alpha_1}, J_{\alpha_2})$ is expressed in terms of primal- and dual solutions. In this case the dual problem reads

$$\int_0^T \int_B \phi_t \omega + \alpha \nabla \phi \cdot \nabla \omega dxdt = 2 \int_0^T \int_W (u-d)\phi dxdt \quad ,$$

which in strong form appears as a backward in time heat equation with zero terminal condition

$$-\omega_t = \nabla \cdot (\alpha \nabla \omega) + 2(u-d)|_W \quad . \quad (38.6)$$

Note that this dual equation is linear and entirely driven by the data mismatch in the well. With perfectly matching data $d = u|_W$, the dual solution is zero.

Along the same lines of argumentation one derives the multilinear operator to the depositional model (38.1)

$$\begin{aligned} \mathcal{A}(u, v)(\phi, \psi) = & \\ & \int_0^T \int_B (Au_t + uh_t + sv_t) \phi + \alpha u \nabla h \cdot \nabla \phi + \alpha s \nabla v \cdot \nabla \phi dxdt \\ & + \int_0^T \int_B (-Au_t - uh_t(1-s)v_t) \psi - \beta u \nabla h \cdot \nabla \psi + \beta(1-s) \nabla v \cdot \nabla \psi dxdt \quad . \end{aligned}$$

The dual system related to the well output functional (38.2) reads

$$\mathcal{A}(\phi, \psi)(\omega, \nu) = 2 \int_0^T \int_W (s - \tilde{s})\phi + (h - \tilde{h})\psi dxdt \quad .$$

By construction it follows $\mathcal{A}(s_p, h_p)(\omega, \nu) = J_p(\alpha, \beta)$. Given both primal and dual solutions, the gradient of the well output functional evaluates as

$$\begin{aligned} J_{\alpha_i}(\alpha, \beta) &= - \int_0^T \int_{B_i} s \nabla h \cdot \nabla \omega dxdt \quad , \\ J_{\beta_i}(\alpha, \beta) &= - \int_0^T \int_{B_i} (1-s) \nabla h \cdot \nabla \nu dxdt \quad . \end{aligned}$$

A detailed derivation including non zero flow conditions is given in [?]. For completeness, not for computation(!), we state the dual system in strong form

$$\begin{aligned} -A(\omega - \nu)_t + h_t(\omega - \nu) + \alpha \nabla h \cdot \nabla \omega &= \beta \nabla h \cdot \nabla \nu + \frac{2}{|W|} (s - \tilde{s}) \Big|_W \\ -(s\omega + (1 - s)\nu)_t &= \nabla \cdot (\alpha s \nabla \omega + \beta(1 - s) \nabla \nu) + \frac{2}{|W|} (h - \tilde{h}) \Big|_W . \end{aligned}$$

Obviously the system is linear and driven by the data mismatch at the well. It always comes with zero terminal condition and no flow conditions along the boundary of the basin. Thus, perfectly matching data results in a trivial dual solution.

38.6 Aspects of the implementation

The FEniCS project DOLFIN [?] automates the solution of PDEs in variational formulation and is therefore especially attractive for implementing dual problems, which are derived in variational form. In this section the coding of the dual diffusion equation (38.6) is illustrated. Testing the equation in space, $\text{supp } \varphi \subset B$, the weak form reads

$$\int_B -\omega_t \varphi + \alpha \nabla \omega \cdot \nabla \varphi dx = 2 \int_W (u - d) \varphi dx .$$

Trapezoidal rule time integration gives

$$\begin{aligned} & - \int_B (\omega^{n+1} - \omega^n) \varphi dx + \frac{\Delta t}{2} \int_B \alpha \nabla (\omega^{n+1} + \omega^n) \cdot \nabla \varphi dx \\ & = \Delta t \int_W (\omega^{n+1} - d^{n+1} + \omega^n - d^n) \varphi dx , \quad n = N, N - 1, \dots, 0 . \end{aligned} \tag{38.7}$$

To evaluate the right hand side, the area of the well is defined as an subdomain:

```
class WellDomain(SubDomain):
    def inside(self, x, on_boundary):
        return bool((0.2 <= x[0] and x[0] <= 0.3 and \
                    0.2 <= x[1] and x[1] <= 0.3))
```

Next, it gets marked:

```
well = WellDomain()
subdomains = MeshFunction("uint", mesh, mesh.topology().dim())
well.mark(subdomains, 1)
```

An integral over the well area is defined:

```
dxWell = Integral("cell", 1)
```

The driving source in (38.7) is written as:

```
f = dt*(u1-d1+u0-d0)*phi*dxWell
b = assemble(f, mesh, cell_domains=subdomains)
```

The first line in (38.7) is stated in variational formulation:

```
F = (u_trial-u)*phi*dx \
    + 0.5*dt*( d*dot( grad(u_trial+u), grad(phi) ) ) *dx
```

Let DOLFIN sort out left- and right hand sides:

```
a = lhs(F); l = rhs(F)
```

Assemble the linear system in matrix-vector form:

```
A = assemble(a, mesh); b += assemble(l, mesh)
```

And solve it:

```
solve(A, u.vector(), b)
```

The direct solver may be replaced by GMRES with algebraic multigrid:

```
solve(A, u.vector(), b, gmres, amg)
```

38.7 Numerical experiments

With these preparations, we are now ready to inspect the well output functional (38.2) for possible calibration of the dual lithology model (38.1) to “observed”, actually generated synthetic, data. We consider the PDE system (38.1) with discontinuous coefficients

$$\alpha = \begin{cases} \alpha_1 & x \geq 1/2 \\ \alpha_2 & x < 1/2 \end{cases}, \quad \beta = \begin{cases} \beta_1 & x \geq 1/2 \\ \beta_2 & x < 1/2 \end{cases}$$

in the unit square $B = [0, 1]^2$. Four wells $W = W_1 \cup W_2 \cup W_3 \cup W_4$ are placed one in each quarter

$$W_4 = [0.3, 0.3] \times [0.7, 0.8], \quad W_3 = [0.7, 0.8] \times [0.7, 0.8],$$

$$W_1 = [0.2, 0.3] \times [0.2, 0.3], \quad W_2 = [0.7, 0.8] \times [0.2, 0.3].$$

Initially s is constant $s(0, \cdot) = 0.5$ and h is piecewise linear

$$h(0, x, y) = 0.5 \max(\max(0.2, (x - 0.1)/2), y - 0.55).$$

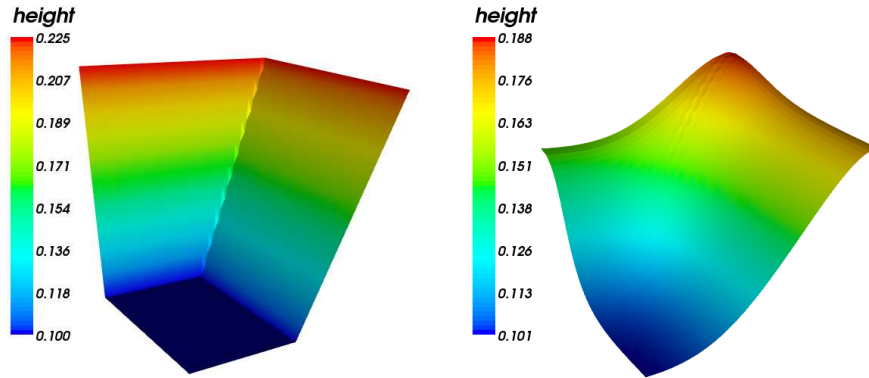


Figure 38.1: Evolution of h , initial left, $t = 0.04$ right.

The diffusive character of the process is evident from the evolution of h as shown in Figure 38.1. No flow boundary conditions are implemented in all simulations throughout this section.

To inspect the output functional, we generate synthetic data by computing a reference solution. In the first experiment, the reference parameters are $(\alpha_1, \alpha_2) = (\beta_1, \beta_2) = (0.8, 0.8)$. We fix β to the reference values and scan the well output over the α -range $[0.0, 1.5]^2$. The upper left plot in Figure 38.2 depicts contours of the apparently convex functional, with the reference parameters as the global minimum. Independent Landweber iterations, started in each corner of the domain identify the optimal parameters in typically five steps. The iteration is stopped if $\|\nabla J(p^k)\| \leq 10^{-7}$, an indication that the minimum is reached. The lower left plot shows the corresponding scan over β where $\alpha = (0.8, 0.8)$ is fixed. Obviously the search directions follow the steepest descent, confirming that the gradients are correctly evaluated via the dual solution. In the right column of Figure 38.2 we see results for the same experiments, but with 5% random noise added to the synthetic well data. In this case the optimal parameters are of course not the reference parameters, but still close. The global picture appears stable with respect to noise, suggesting that the concept allows to calibrate diffusive, depositional models to data observed in wells.

Ultimately, the goal is to calibrate all four parameters $\alpha = (\alpha_1, \alpha_2)$ and $\beta = (\beta_1, \beta_2)$ to available data. Figure 38.3 depicts Landweber iterations in four dimensional parameter space. Actually projections onto the α and β coordinate plane are shown. Each subplot depicts one iteration. The initial guess varies from line to line. Obviously, all iterations converge and, without noise added, the reference parameters, $\alpha = \beta = (0.8, 0.8)$, are detected as optimal parameters. Adding 5% random noise to the recorded data, we simulate data observed in wells. In this situation, see the right column, the algorithm identifies optimal parameters, which are clearly off the reference. Fig. 38.5 depicts fifty realiza-

tions of this experiments. The distribution of the optimal parameters is shown together with their average in red. The left column in Fig. 38.5 corresponds to the reference parameters $(\alpha_1, \alpha_2) = (\beta_1, \beta_2) = (0.8, 0.8)$ as in Fig. 38.3. The initial guesses vary from row to row and are the same as in Fig. 38.3. On average the calibrated, optimal parameters are close to the reference. Typical standard deviations vary from 0.07 to 0.17, depending on the coefficient.

In the next experiments non uniform reference parameters are set for $\alpha = (0.6, 1.0)$ and $\beta = (1.0, 0.6)$. Figure 38.4 shows iterations with the noise-free reference solution used as data on the left hand side. Within the precision of the stopping criterion, the reference parameters are detected. Adding 5% noise to the well data leads to different optimal parameters, just as expected. On average however, the optimal parameters obtained in repeated calibrations match the reference parameters quite well, see Figure 38.5, right hand side.

In the next experiments, β is discontinuous along $y = 1/2$ and piecewise constant in the lower and upper half of the basin

$$\alpha = \begin{cases} \alpha_1 & x \geq 1/2 \\ \alpha_2 & x < 1/2 \end{cases}, \quad \beta = \begin{cases} \beta_1 & y \geq 1/2 \\ \beta_2 & y < 1/2 \end{cases}.$$

In this way the evolution is governed by different diffusion parameters in each quarter of the basin. Having placed one well i each quarter, one can effectively calibrate the model to synthetic data with and without random noise, as shown in Figures 38.6 and 38.7.

38.8 Results and conclusion

The calibration of piecewise constant diffusion coefficients using local data in a small number of wells is a well behaved inverse problem. The convexity of the output functional, which is the basis for a successful minimization, remains stable with random noise added to the well data.

We have automated the calibration of diffusive transport coefficients in two ways: First, the Landweber algorithm, with duality based gradients, automatically detects optimal parameters. Second, the FEniCS project DOLFIN, automatically implements the methods. As the dual problems are derived in variational form, DOLFIN is the appropriate tool for efficient implementation.

Acknowledgments

The presented work was funded by a research grant from StatoilHydro. The author wants to thank Are Magnus Bruaset for initiating this work. Many thanks to Bjørn Fredrik Nielsen for thoughtful suggestions regarding inverse problems as well as Anders Logg and Ola Skavhaug for their support regarding the DOLFIN software. The work has been conducted at Simula Research

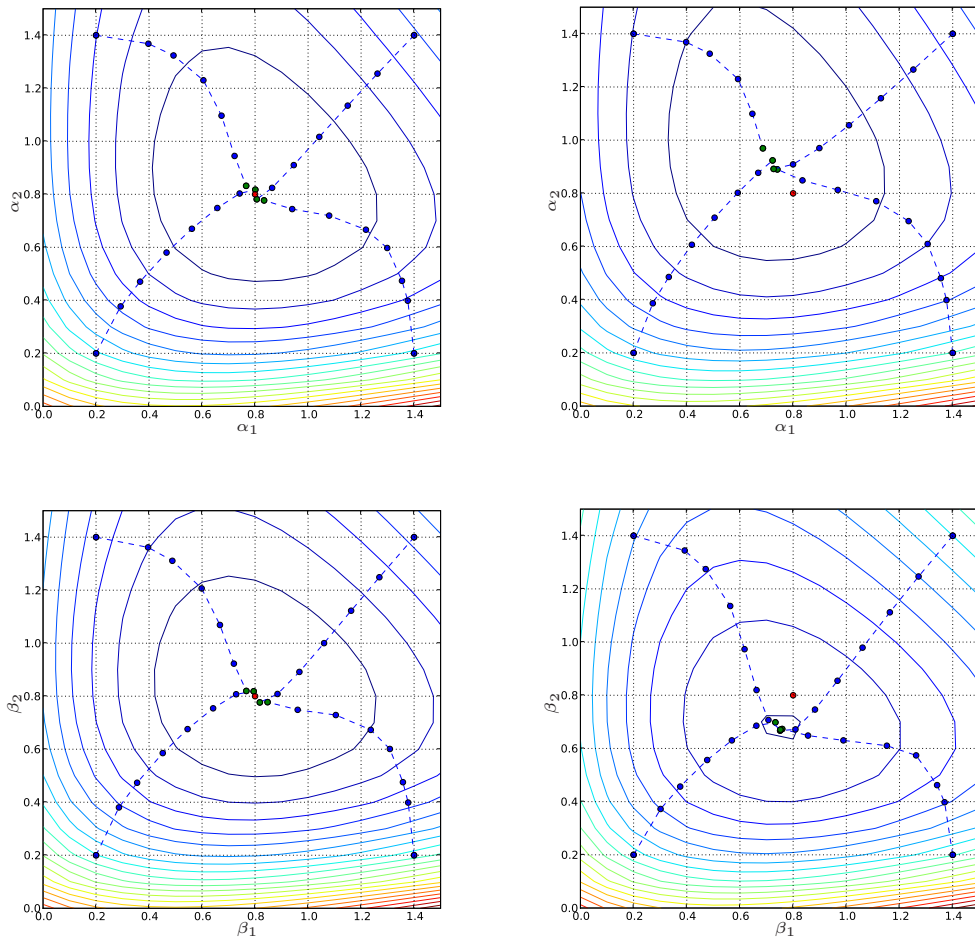


Figure 38.2: Contours of J and Landweber iterations, optimal parameters (green), reference parameters (red). Clean data left column, noisy data right.

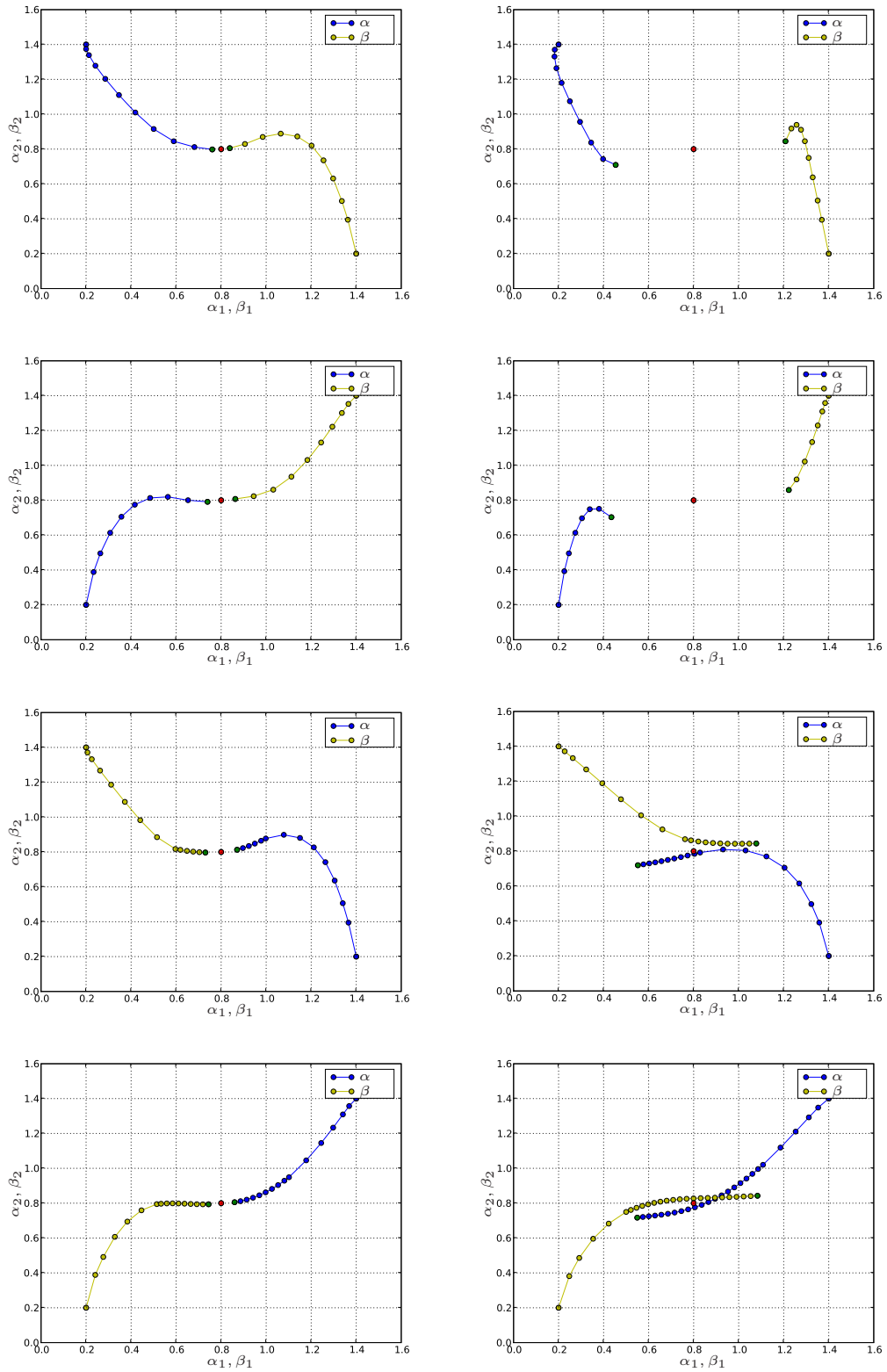


Figure 38.3: Landweber iterations. Clean (left) and noisy data (right).

Automatic Calibration of Depositional Models

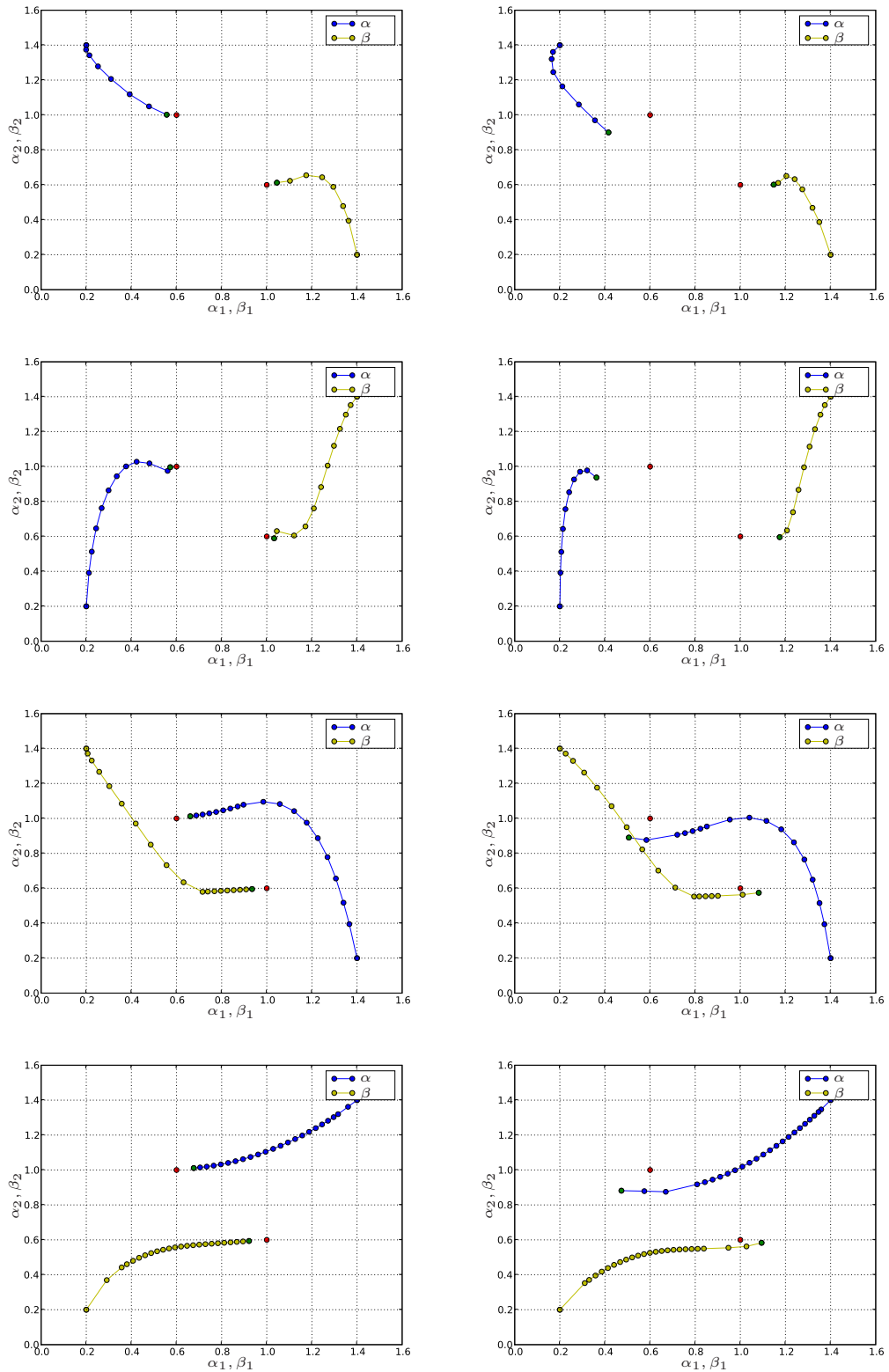


Figure 38.4: Landweber iterations. Clean (left) and noisy data (right).

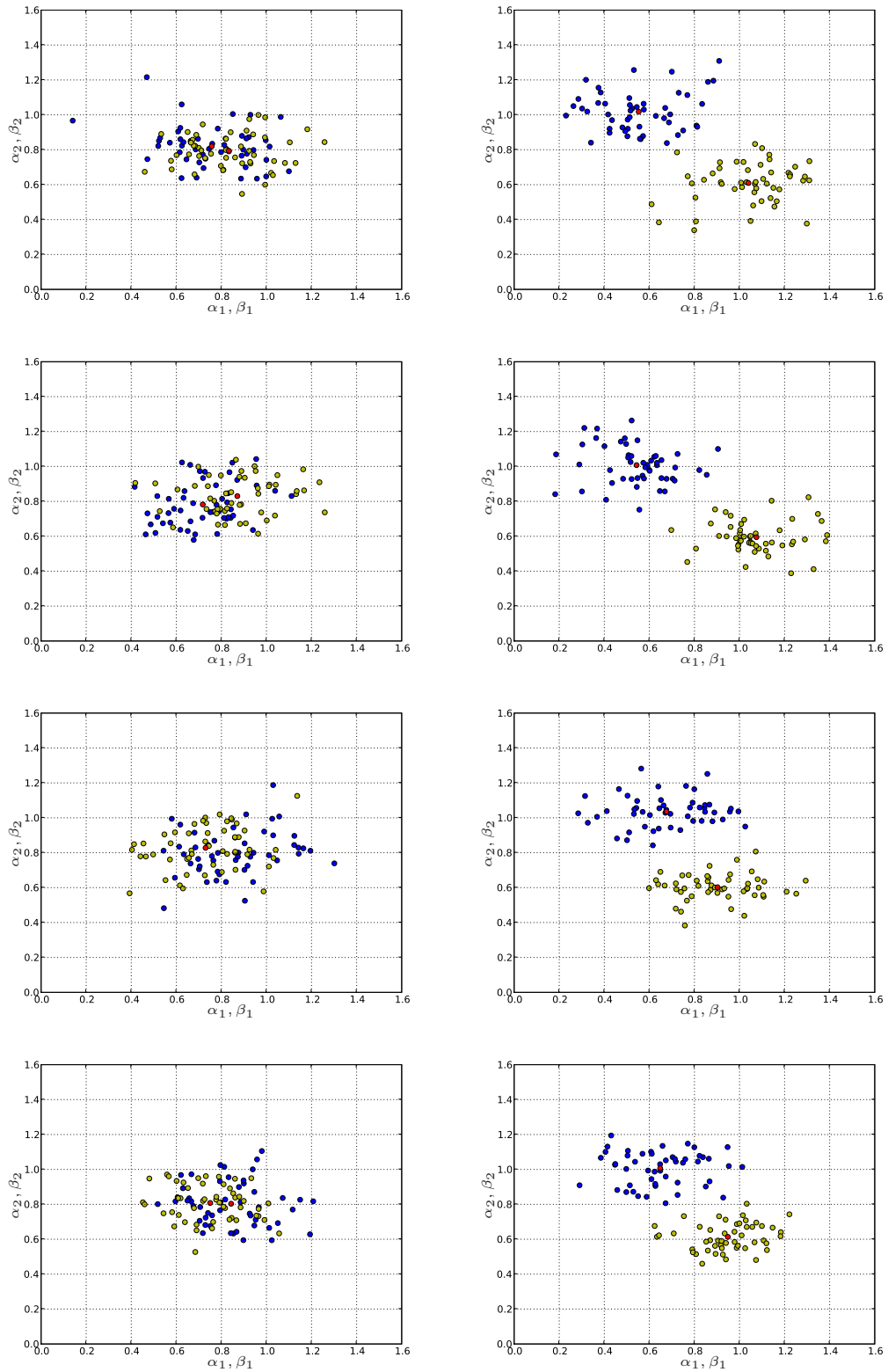


Figure 38.5: Sets of optimal parameters calibrated to noisy data, α blue, β yellow, average red.

Automatic Calibration of Depositional Models

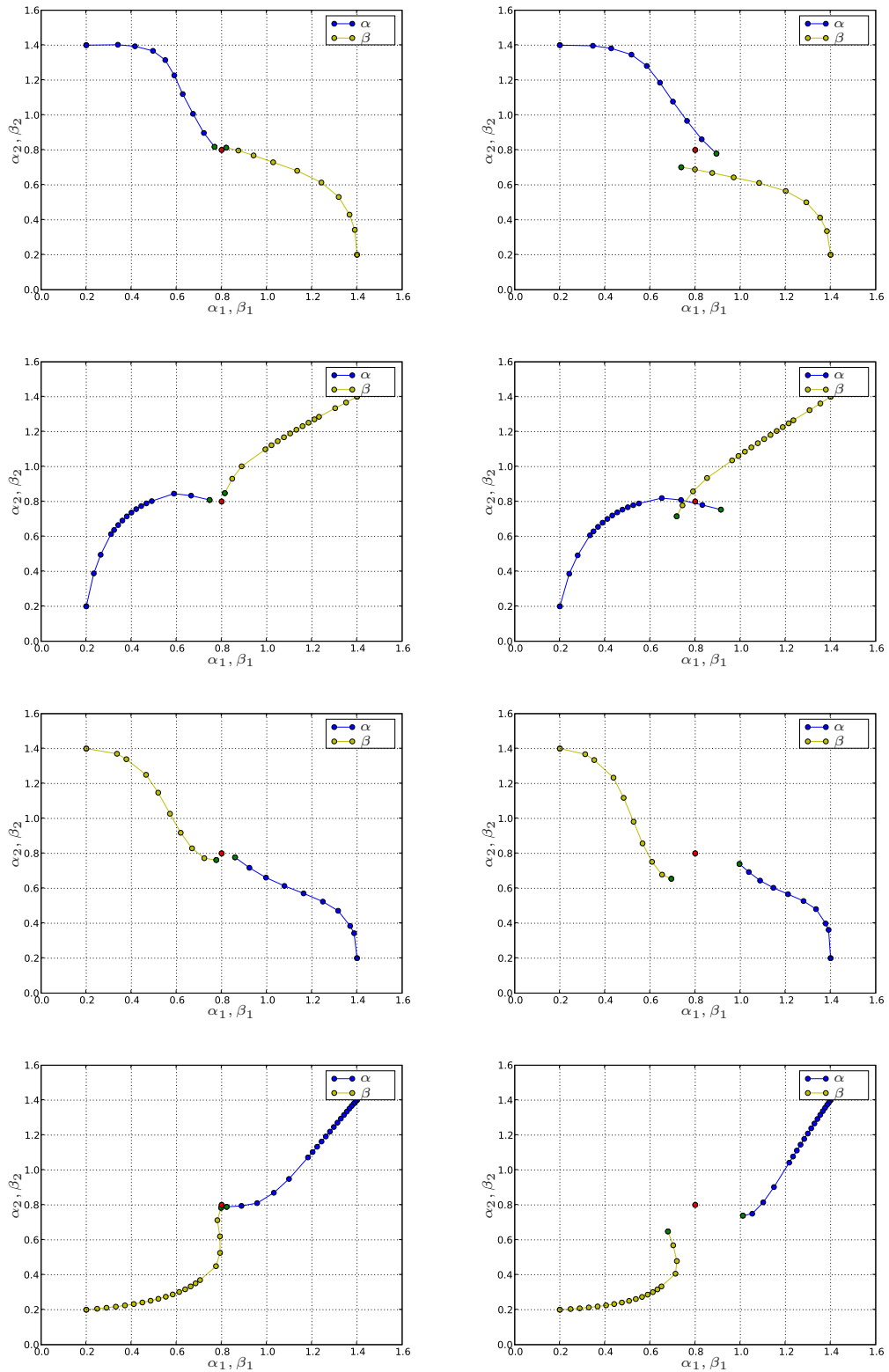


Figure 38.6: Landweber iterations. Clean (left) and noisy data (right).

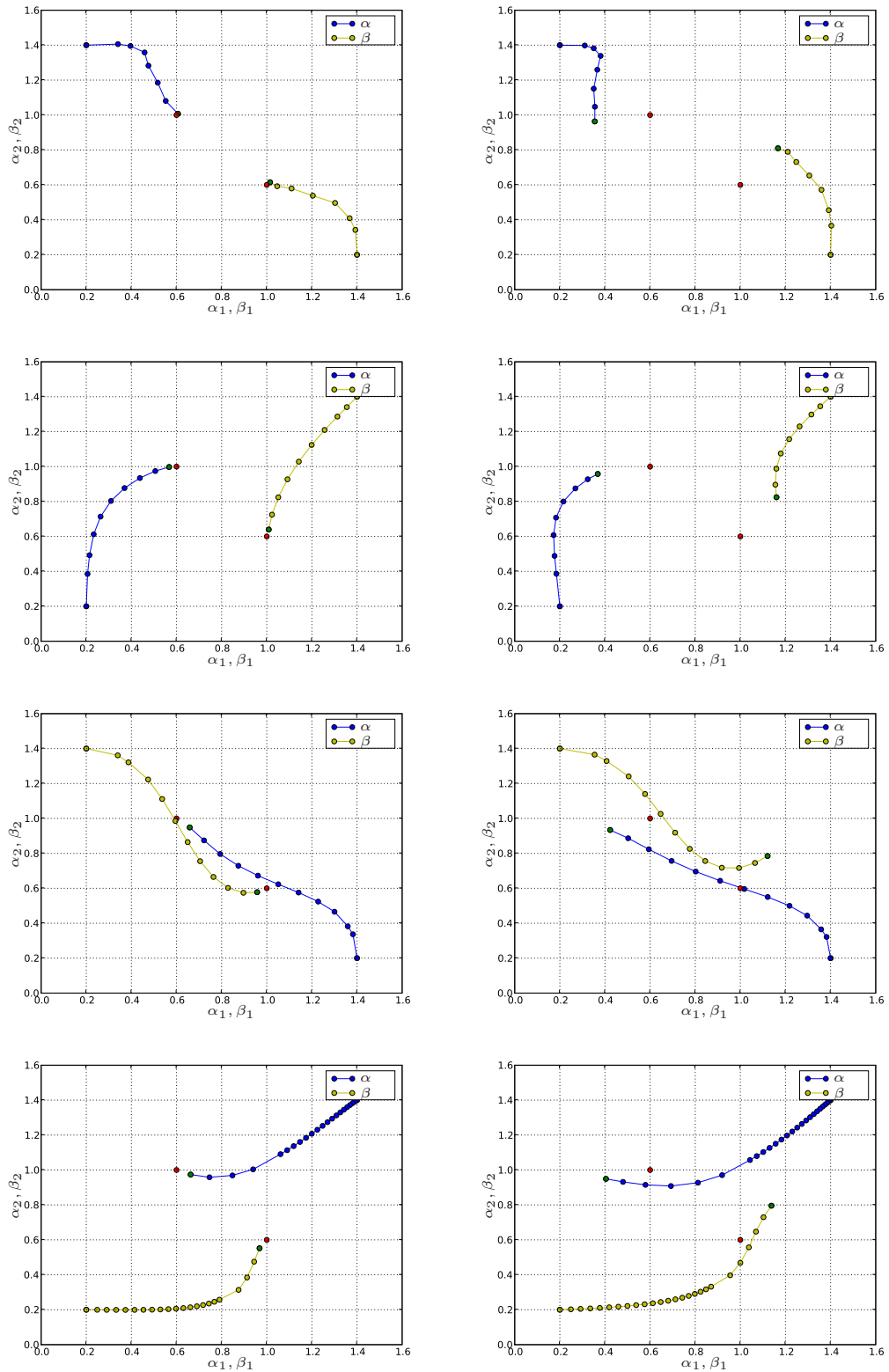


Figure 38.7: Landweber iterations. Clean (left) and noisy data (right).

Laboratory as part of CBC, a Center of Excellence awarded by the Norwegian Research Council.

Computational Thermodynamics

By Johan Hoffman, Claes Johnson and Murtazo Nazarov

Chapter ref: [**hoffman-3**]

- ▶ Editor note: *Missing figures, need to be added in EPS sub directory.*
- ▶ Editor note: *Missing bibliography, needs to be added to common bibliography.bib in correct format.*

We test the functionality of FEniCS on the challenge of computational thermodynamics in the form of a finite element solver of the Euler equations expressing conservation of mass, momentum and energy. We show that computational solutions satisfy a 2nd Law formulated in terms of kinetic energy, internal (heat) energy, work and shock/turbulent dissipation, without reference to entropy. We show that the 2nd Law expresses an irreversible transfer of kinetic energy to heat energy in shock/turbulent dissipation arising because the Euler equations lack pointwise solutions, and thus explains the occurrence of irreversibility in formally reversible systems as an effect of instability with blow-up of Euler residuals combined with finite precision computation, without resort to statistical mechanics or ad hoc viscous regularization. We simulate the classical Joule experiment of a gas expanding from rest under temperature drop followed by temperature recovery by turbulent dissipation until rest in the double volume.

39.1 FEniCS as Computational Science

The goal of the FEniCS project is to develop software for automated computational solution of differential equations based on a finite element methodology combining generality with efficiency. The FEniCS Application Unicorn offers a

solver for a fluid-solids continuum based on a unified Eulerian formulation of momentum balance combined with constitutive laws for fluid/solid in Eulerian/updated Lagrangian form, which shows the capability of FEniCS for automation of computational fluid-structure interaction.

Thermodynamics is a basic area of continuum mechanics with many important applications, which however is feared by both teachers, students and engineers as being difficult to understand and to apply, principally because of the appearance of turbulence. In this article we show that turbulent thermodynamics can be made understandable and useful by automated computational solution, as another example of the capability of FEniCS.

The biggest mystery of classical thermodynamics is the 2nd Law about entropy and automation cannot harbor any mystery. Expert systems are required for mysteries and FEniCS is not an expert system. Automation requires a continuum mechanics formulation of thermodynamics with a transparent 2nd Law. We present a formulation of thermodynamics based on finite precision computation with a 2nd Law without reference to entropy, which we show can serve as a basis for automated computational simulation of complex turbulent thermodynamics and thus can open to new insight and design, a main goal of FEniCS. In this setting the digital finite element model becomes the real model of the physics of thermodynamics viewed as a form of analog finite precision computation, a model which is open to inspection and analysis because solutions can be computed and put on the table. This represents a new kind of science in the spirit of Dijkstra [?] and Wolfram [?], which can be explored using FEniCS and which we present in non-technical form in My Book of Knols [?].

39.2 The 1st and 2nd Laws of Thermodynamics

Heat, a quantity which functions to animate, derives from an internal fire located in the left ventricle. (Hippocrates, 460 B.C.)

Thermodynamics is fundamental in a wide range of phenomena from macroscopic to microscopic scales. Thermodynamics essentially concerns the interplay between *heat energy* and *kinetic energy* in a *gas* or *fluid*. Kinetic energy, or *mechanical energy*, may generate heat energy by *compression* or *turbulent dissipation*. Heat energy may generate kinetic energy by *expansion*, but not through a *reverse* process of turbulent dissipation. The industrial society of the 19th century was built on the use of *steam engines*, and the initial motivation to understand thermodynamics came from a need to increase the efficiency of steam engines for conversion of heat energy to useful mechanical energy. Thermodynamics is closely connected to the dynamics of *slightly viscous* and *compressible* gases, since substantial compression and expansion can occur in a gas, but less in fluids (and solids).

The development of classical thermodynamics as a rational science based on logical deduction from a set of axioms, was initiated in the 19th century by Carnot [?], Clausius [?] and Lord Kelvin [?], who formulated the basic axioms in the form of the *1st Law* and the *2nd Law* of thermodynamics. The 1st Law states (for an isolated system) that the *total energy*, the sum of kinetic and heat energy, is conserved. The 1st Law is naturally generalized to include also conservation of mass and Newton's law of conservation of momentum and then can be expressed as the *Euler equations* for a gas/fluid with *vanishing viscosity*.

The 2nd Law has the form of an inequality $dS \geq 0$ for a quantity named *entropy* denoted by S , with dS denoting change thereof, supposedly expressing a basic feature of real thermodynamic processes. The classical 2nd Law states that the entropy cannot decrease; it may stay constant or it may increase, but it can never decrease (for an isolated system).

The role of the 2nd Law is to give a scientific basis to the many observations of *irreversible* processes, that is, processes which cannot be reversed in time, like running a movie backwards. Time reversal of a process with strictly increasing entropy, would correspond to a process with strictly decreasing entropy, which would violate the 2nd Law and therefore could not occur. A perpetum mobile would represent a reversible process and so the role of the 2nd Law is in particular to explain *why* it is impossible to construct a perpetum mobile, and *why* time is moving forward in the direction an *arrow of time*, as expressed by Max Planck [?, ?, ?]: *Were it not for the existence of irreversible processes, the entire edifice of the 2nd Law would crumble.*

While the 1st Law in the form of the Euler equations expressing conservation of mass, momentum and total energy can be understood and motivated on rational grounds, the nature of the 2nd Law is mysterious. It does not seem to be a consequence of the 1st Law, since the Euler equations seem to be time reversible, and the role of the 2nd Law is to explain irreversibility. Thus questions are lining up: nIf the 2nd Law is a new independent law of Nature, how can it be justified? What is the physical significance of that quantity named entropy, which Nature can only get more of and never can get rid of, like a steadily accumulating heap of waste? What mechanism prevents Nature from recycling entropy? How can irreversibility arise in a reversible system? How can viscous dissipation arise in a system with vanishing viscosity? Why is there no *Maxwell demon* [?]? Why can a gas by itself expand into a larger volume, but not by itself contract back again, if the motion of the gas molecules is governed by the reversible Newton's laws of motion? Why is there an arrow of time? This article presents answers.

39.3 The Enigma

Those who have talked of "chance" are the inheritors of antique superstition and ignorance...whose minds have never been illuminated by a ray of scien-

tific thought. (T. H. Huxley)

These were the questions which confronted scientists in the late 19th century, after the introduction of the concept of entropy by Clausius in 1865, and these showed to be tough questions to answer. After much struggle, agony and debate, the agreement of the physics community has become to view *statistical mechanics* based on an assumption of *molecular chaos* as developed by Boltzmann [?], to offer a rationalization of the classical 2nd Law in the form of a tendency of (isolated) physical processes to move from improbable towards more probable states, or from ordered to less ordered states. Boltzmann's assumption of molecular chaos in a dilute gas of colliding molecules, is that two molecules about to collide have independent velocities, which led to the *H-theorem* for *Boltzmann's equations* stating that a certain quantity denoted by H could not decrease and thus could serve as an entropy defining an arrow of time. Increasing disorder would thus represent increasing entropy, and the classical 2nd Law would reflect the eternal pessimists idea that things always get more messy, and that there is really no limit to this, except when everything is as messy as it can ever get. Of course, experience could give (some) support this idea, but the trouble is that it prevents things from ever becoming less messy or more structured, and thus may seem a bit too pessimistic. No doubt, it would seem to contradict the many observations of *emergence* of ordered non-organic structures (like crystals or waves and cyclons) and organic structures (like DNA and human beings), seemingly out of disordered chaos, as evidenced by the physics Nobel Laureate Robert Laughlin [?].

Most trained thermodynamicists would here say that emergence of order out of chaos, in fact does not contradict the classical 2nd Law, because it concerns "non-isolated systems". But they would probably insist that the Universe as a whole (isolated system) would steadily evolve towards a "heat-death" with maximal entropy/disorder (and no life), thus fulfilling the pessimists expectation. The question from where the initial order came from, would however be left open.

The standard presentation of thermodynamics based on the 1st and 2nd Laws, thus involves a mixture of deterministic models (Boltzmann's equations with the H-theorem) based on statistical assumptions (molecular chaos) making the subject admittedly difficult to both learn, teach and apply, despite its strong importance. This is primarily because the question *why* necessarily $dS \geq 0$ and never $dS < 0$, is not given a convincing understandable answer. In fact, statistical mechanics allows $dS < 0$, although it is claimed to be very unlikely. The basic objective of statistical mechanics as the basis of classical thermodynamics, thus is to (i) give the entropy a physical meaning, and (ii) to motivate its tendency to (usually) increase. Before statistical mechanics, the 2nd Law was viewed as an experimental fact, which could not be rationalized theoretically. The classical view on the 2nd Law is thus either as a statistical law of large numbers or as an experimental fact, both without a rational deterministic mechanistic theo-

retical foundation. The problem with thermodynamics in this form is that it is understood by very few, if any:

- *Every mathematician knows it is impossible to understand an elementary course in thermodynamics.* (V. Arnold)
- *...no one knows what entropy is, so if you in a debate use this concept, you will always have an advantage.* (von Neumann to Shannon)
- *As anyone who has taken a course in thermodynamics is well aware, the mathematics used in proving Clausius' theorem (the 2nd Law) is of a very special kind, having only the most tenuous relation to that known to mathematicians.* (S. Brush [?])
- *Where does irreversibility come from? It does not come from Newton's laws. Obviously there must be some law, some obscure but fundamental equation, perhaps in electricity, maybe in neutrino physics, in which it does matter which way time goes.* (Feynman [?])
- *For three hundred years science has been dominated by a Newtonian paradigm presenting the World either as a sterile mechanical clock or in a state of degeneration and increasing disorder...It has always seemed paradoxical that a theory based on Newtonian mechanics can lead to chaos just because the number of particles is large, and it is subjectively decided that their precise motion cannot be observed by humans... In the Newtonian world of necessity, there is no arrow of time. Boltzmann found an arrow hidden in Nature's molecular game of roulette.* (Paul Davies [?])
- *The goal of deriving the law of entropy increase from statistical mechanics has so far eluded the deepest thinkers.* (Lieb [?])
- *There are great physicists who have not understood it.* (Einstein about Boltzmann's statistical mechanics)

39.4 Computational Foundation

In this note we present a foundation of thermodynamics, further elaborated in [?, ?], where the basic assumption of statistical mechanics of molecular chaos, is replaced by *deterministic finite precision computation*, more precisely by a *least squares stabilized finite element method* for the Euler equations, referred to as *Euler General Galerkin* or *EG2*. In the spirit of Dijkstra [?], we thus view EG2 as the physical model of thermodynamics, that is the Euler equations together with a computational solution procedure, and not just the Euler equations without constructive solution procedure as in a classical non-computational approach.

Using EG2 as a model of thermodynamics changes the questions and answers and opens new possibilities of progress together with new challenges to mathematical analysis and computation. The basic new feature is that EG2 solutions

are computed and thus are available to inspection. This means that the analysis of solutions shifts from *a priori* to *a posteriori*; after the solution has been computed it can be inspected.

Inspecting computed EG2 solutions we find that they are *turbulent* and have *shocks*, which is identified by pointwise large Euler residuals, reflecting that pointwise solutions to the Euler equations are lacking. The enigma of thermodynamics is thus the enigma of turbulence (since the basic nature of shocks is understood). Computational thermodynamics thus essentially concerns computational turbulence. In this note and [?] we present evidence that EG2 opens to a resolution of the enigma of turbulence and thus of thermodynamics.

The fundamental question concerns *wellposedness* in the sense of Hadamard, that is what aspects or *outputs* of turbulent/shock solutions are stable under perturbations in the sense that small perturbations have small effects. We show that wellposedness of EG2 solutions can be tested a posteriori by computationally solving a *dual linearized problem*, through which the output sensitivity of non-zero Euler residuals can be estimated. We find that mean-value outputs such as drag and lift and total turbulent dissipation are wellposed, while point-values of turbulent flow are not. We can thus a posteriori in a case by case manner, assess the quality of EG2 solutions as solutions of the Euler equations.

We formulate a *2nd Law* for EG2 without the concept of entropy, in terms of the basic physical quantities of kinetic energy K , heat energy E , rate of *work* W and shock/turbulent dissipation $D > 0$. The new 2nd Law reads

$$\dot{K} = W - D, \quad \dot{E} = -W + D, \tag{39.1}$$

where the dot indicates time differentiation. Slightly viscous flow always develops turbulence/shocks with $D > 0$, and the 2nd Law thus expresses an irreversible transfer of kinetic energy into heat energy, while the total energy $E + K$ remains constant.

With the 2nd Law in the form (39.1), we avoid the (difficult) main task of statistical mechanics of specifying the physical significance of entropy and motivating its tendency to increase by probabilistic considerations based on (tricky) combinatorics. Thus using *Ockham's razor* [?], we rationalize a scientific theory of major importance making it both more understandable and more useful. The new 2nd Law is closer to classical Newtonian mechanics than the 2nd Law of statistical mechanics, and thus can be viewed to be more fundamental.

The new 2nd Law is a consequence of the 1st Law in the form of the Euler equations combined with EG2 finite precision computation effectively introducing viscosity and viscous dissipation. These effects appear as a consequence of the non-existence of pointwise solutions to the Euler equations reflecting instabilities leading to the development shocks and turbulence in which large scale kinetic energy is transferred to small scale kinetic energy in the form of heat energy. The viscous dissipation can be interpreted as a penalty on pointwise large

Euler residuals arising in shocks/turbulence, with the penalty being directly coupled to the violation following a principle of criminal law exposed in [?]. EG2 thus explains the 2nd Law as a consequence of the non-existence of pointwise solutions with small Euler residuals. This offers an understanding to the emergence of irreversible solutions of the formally reversible Euler equations. If pointwise solutions had existed, they would have been reversible without dissipation, but they don't exist, and the existing computational solutions have dissipation and thus are irreversible.

39.5 Viscosity Solutions

An EG2 solution can be viewed as particular *viscosity solution* of the Euler equations, which is a solution of *regularized Euler equations* augmented by additive terms modeling viscosity effects with small viscosity coefficients. The effective viscosity in an EG2 solution typically may be comparable to the mesh size.

For incompressible flow the existence of viscosity solutions, with suitable solution dependent viscosity coefficients, can be proved a priori using standard techniques of analytical mathematics. Viscosity solutions are pointwise solutions of the regularized equations. But already the most basic problem with constant viscosity, the incompressible Navier-Stokes equations for a Newtonian fluid, presents technical difficulties, and is one of the open Clay Millennium Problems.

For compressible flow the technical complications are even more severe, and it is not clear which viscosities would be required for an analytical proof of the existence of viscosity solutions [?] to the Euler equations. Furthermore, the question of wellposedness is typically left out, as in the formulation of the Navier-Stokes Millennium Problem, with the motivation that first the existence problem has to be settled. Altogether, analytical mathematics seems to have little to offer a priori concerning the existence and wellposedness of solutions of the compressible Euler equations. In contrast, EG2 computational solutions of the Euler equations seem to offer a wealth of information a posteriori, in particular concerning wellposedness by duality.

An EG2 solution thus can be viewed as a specific viscosity solution with a specific regularization from the least squares stabilization, in particular of the momentum equation, which is necessary because pointwise momentum balance is impossible to achieve in the presence of shocks/turbulence. The EG2 viscosity can be viewed to be the minimal viscosity required to handle the contradiction behind the non-existence of pointwise solutions. For a shock EG2 could then be directly interpreted as a certain physical mechanism preventing a shock wave from turning over, and for turbulence as a form of automatic computational turbulence model.

EG2 thermodynamics can be viewed as form of deterministic chaos, where

the mechanism is open to inspection and can be used for prediction. On the other hand, the mechanism of statistical mechanics is not open to inspection and can only be based on ad hoc assumption, as noted by e.g. Einstein [?]. If Boltzmann's assumption of molecular chaos cannot be justified, and is not needed, why consider it at all, [?]

Figure 39.1: Joule's 1845 experiment

► Editor note: *Missing figure*.

39.6 Joule's 1845 Experiment

To illustrate basic aspects of thermodynamics, we recall Joule's experiment from 1845 with a gas initially at rest with temperature $T = 1$ at a certain pressure in a certain volume immersed into a container of water, see Fig. 39.1. At initial time a valve was opened and the gas was allowed to expand into the double volume while the temperature change in the water was carefully measured by Joule. To the great surprise of both Joule and the scientific community, no change of the temperature of the water could be detected, in contradiction with the expectation that the gas would cool off under expansion. Moreover, the expansion was impossible to reverse; the gas had no inclination to contract back to the original volume. Simulating Joule's experiment using EG2, we discover the following as displayed in Fig. 39.2-39.7

Figure 39.2: Density at two time instants

Figure 39.3: Temperature at two time instants

In a first phase the temperature drops below 1 as the gas expands with increasing velocity, and in a second phase shocks/turbulence appear and heat the gas towards a final state with the gas at rest in the double volume and the temperature back to $T = 1$. The total (heat) energy is, of course, conserved since the density is reduced by a factor 2 after expansion to double volume. We can also understand that the rapidity of the expansion process makes it difficult to detect any temperature drop in the water in the initial phase. Altogether, using EG2 we can first simulate and then understand Joule's experiment, and we thus see no reason to be surprised. We shall see below as a consequence of the 2nd Law that reversal of the process with the gas contracting back to the original small

Figure 39.4: Average density in left and right chamber

Figure 39.5: Average temperature in left and right chamber

volume, is impossible because the only way the gas can be put into motion is by expansion, and thus contraction is impossible.

In statistical mechanics the dynamics of the process would be dismissed and only the initial and final state would be subject to analysis. The final state would then be viewed as being “less ordered” or “more probable” or having “higher entropy”, because the gas would occupy a larger volume, and the reverse process with the gas contracting back to the initial small volume, if not completely impossible, would be “improbable”. But to say that a probable state is more probable than an improbable state is more mystifying than informative. Taking the true dynamics of the process into account including in particular the second phase with heat generation from shocks or turbulence, we can understand the observation of constant temperature and irreversibility in a deterministic fashion without using any mystics of entropy based on mystics of statistics. In [?] we develop a variety of aspects of the arrow of time enforced by the new 2nd Law.

39.7 The Euler Equations

We consider the Euler equations for an inviscid perfect gas enclosed in a volume Ω in \mathbb{R}^3 with boundary Γ over a time interval $I = (0, 1]$ expressing conservation of *mass density* ρ , *momentum* $m = (m_1, m_2, m_3)$ and *internal energy* e : Find $\hat{u} = (\rho, m, e)$ depending on $(x, t) \in Q \equiv \Omega \times I$ such that

$$\begin{aligned} R_\rho(\hat{u}) &\equiv \dot{\rho} + \nabla \cdot (\rho u) &= 0 &\text{ in } Q, \\ R_m(\hat{u}) &\equiv \dot{m} + \nabla \cdot (mu + p) &= f &\text{ in } Q, \\ R_e(\hat{u}) &\equiv \dot{e} + \nabla \cdot (eu) + p \nabla \cdot u &= g &\text{ in } Q, \\ u \cdot n &= 0 &&\text{ on } \Gamma \times I \\ \hat{u}(\cdot, 0) &= \hat{u}^0 &&\text{ in } \Omega, \end{aligned} \tag{39.2}$$

where $u = \frac{m}{\rho}$ is the velocity, $p = (\gamma - 1)e$ with $\gamma > 1$ a *gas constant*, f is a given volume force, g a heat source/sink and \hat{u}^0 a given initial state. We here express energy conservation in terms of the internal energy $e = \rho T$, with T the temperature, and not as conservation of the *total energy* $\epsilon = e + k$ with $k = \frac{\rho v^2}{2}$ the *kinetic energy*, in the form $\dot{\epsilon} + \nabla \cdot (\epsilon u) = 0$. Because of the appearance of

Figure 39.6: Average kinetic energy and temperature: short time

Figure 39.7: Average kinetic energy in both, left and right chamber(s): long time

shocks/turbulence, the Euler equations lack pointwise solutions, except possible for short time, and regularization is therefore necessary. For a mono-atomic gas $\gamma = 5/3$ and (39.2) then is a *parameter-free model*, the ideal form of mathematical model according to Einstein...

39.8 Energy Estimates for Viscosity Solutions

For the discussion we consider the following regularized version of (39.2) assuming for simplicity that $f = 0$ and $g = 0$: Find $\hat{u}_{\nu,\mu} \equiv \hat{u} = (\rho, m, e)$ such that

$$\begin{aligned} R_\rho(\hat{u}) &= 0 \quad \text{in } Q, \\ R_m(\hat{u}) &= -\nabla \cdot (\nu \nabla u) + \nabla(\mu p \nabla \cdot u) \quad \text{in } Q, \\ R_e(\hat{u}) &= \nu |\nabla u|^2 \quad \text{in } Q, \\ u &= 0 \quad \text{on } \Gamma \times I, \\ \hat{u}(\cdot, 0) &= \hat{u}^0 \quad \text{in } \Omega, \end{aligned} \tag{39.3}$$

where $\nu > 0$ is a *shear viscosity* $\mu \gg \nu \geq 0$ if $\nabla \cdot u > 0$ in expansion (with $\mu = 0$ if $\nabla \cdot u \leq 0$ in compression), is a small *bulk viscosity*, and we use the notation $|\nabla u|^2 = \sum_i |\nabla u_i|^2$. We shall see that the bulk viscosity is a safety feature putting a limit to the work $p \nabla \cdot u$ in expansion appearing in the energy balance.

We note that only the momentum equation is subject to viscous regularization. Further, we note that the shear viscosity term in the momentum equation multiplied by the velocity u (and formally integrated by parts) appears as a positive right hand side in the equation for the internal energy, reflecting that the dissipation from shear viscosity is transformed into internal heat energy. In contrast, the dissipation from the bulk viscosity represents another form of internal energy not accounted for as heat energy, acting only as a safety feature in the sense that its contribution to the energy balance in general will be small, while that from the shear viscosity in general will be substantial reflecting shock/turbulent dissipation.

Below we will consider instead regularization by EG2 with the advantage that the EG2 solution is computed and thus is available to inspection, while $\hat{u}_{\nu,\mu}$ is not. We shall see that EG2 regularization can be interpreted as a (mesh-dependent) combination of bulk and shear viscosity and thus (39.3) can be viewed as an analytical model of EG2 open to simple form of analysis in the form of energy estimates.

As indicated, the existence of a pointwise solution $\hat{u} = \hat{u}_{\nu,\mu}$ to the regularized equations (39.3) is an open problem of analytical mathematics, although with suitable additional regularization it could be possible to settle [?]. Fortunately, we can leave this problem aside, since EG2 solutions will be shown to exist a

posteriori by computation. We thus formally assume that (39.3) admits a pointwise solution, and derive basic energy estimates which will be paralleled below for EG2. We thus use the regularized problem (39.3) to illustrate basic features of EG2, including the 2nd Law.

We shall prove now that a regularized solution \hat{u} is an approximate solution of the Euler equations in the sense that $R_\rho(\hat{u}) = 0$ and $R_e(\hat{u}) \geq 0$ pointwise, $R_m(\hat{u})$ is weakly small in the sense that

$$\|R_m(\hat{u})\|_{-1} \leq \frac{\sqrt{\nu}}{\sqrt{\mu}} + \sqrt{\mu} \ll 1, \quad (39.4)$$

where $\|\cdot\|_{-1}$ denotes the $L_2(I; H^{-1}(\Omega))$ -norm, and the following 2nd Law holds:

$$\dot{K} \leq W - D, \quad \dot{E} = -W + D, \quad (39.5)$$

where

$$K = \int_{\Omega} k \, dx, \quad E = \int_{\Omega} e \, dx, \quad W = \int_{\Omega} p \nabla \cdot u \, dx, \quad D = \int_{\Omega} \nu |\nabla u|^2 \, dx.$$

Choosing $\nu \ll \mu$ we can assure that $\|R_m(\hat{u}_{\nu,\mu})\|_{-1}$ is small. We can view the 2nd Law as a compensation for the fact that the momentum equation is only satisfied in a weak sense, and the equation for internal energy with inequality.

The 2nd Law (39.5) states an irreversible transfer of kinetic energy to heat energy in the presence of shocks/turbulence with $D > 0$, which is the generic case. On the other hand, the sign of W is variable and thus the corresponding energy transfer may go in either direction.

The basic technical step is to multiply the momentum equation by u , and use the mass balance equation in the form $\frac{|u|^2}{2}(\dot{\rho} + \nabla \cdot (\rho u)) = 0$, to get

$$\dot{k} + \nabla \cdot (ku) + p \nabla \cdot u - \nabla(\mu p \nabla \cdot u) \cdot u - \nabla \cdot (\nu \nabla u) \cdot u = 0. \quad (39.6)$$

By integration in space it follows that $\dot{K} \leq W - D$, and similarly it follows that $\dot{E} = -W + D$ from the equation for e , which proves the 2nd Law. Adding next (39.6) to the equation for the internal energy e and integrating in space, gives

$$\dot{K} + \dot{E} + \int_{\Omega} \mu p (\nabla \cdot u)^2 \, dx = 0,$$

and thus after integration in time

$$K(1) + E(1) + \int_Q \mu p (\nabla \cdot u)^2 \, dx dt = K(0) + E(0). \quad (39.7)$$

We now need to show that $E(1) \geq 0$ (or more generally that $E(t) > 0$ for $t \in I$), and to this end we rewrite the equation for the internal energy as follows:

$$D_u e + \gamma e \nabla \cdot u = \nu |\nabla u|^2,$$

where $D_u e = \dot{e} + u \cdot \nabla e$ is the material derivative of e following the fluid particles with velocity u . Assuming that $e(x, 0) > 0$ for $x \in \Omega$, it follows that $e(x, 1) > 0$ for $x \in \Omega$, and thus $E(1) > 0$. Assuming $K(0) + E(0) = 1$ the energy estimate (39.7) thus shows that

$$\int_Q \mu p (\nabla \cdot u)^2 dx dt \leq 1, \quad (39.8)$$

and also that $E(t) \leq 1$ for $t \in I$. Next, integrating (39.6) in space and time gives, assuming for simplicity that $K(0) = 0$,

$$K(1) + \int_Q \nu (\Delta u)^2 dx dt = \int_Q p \nabla \cdot u dx dt - \int_Q \mu p (\nabla \cdot u)^2 dx dt \leq \frac{1}{\mu} \int_Q p dx dt \leq \frac{1}{\mu},$$

where we used that $\int_Q p dx dt = (\gamma - 1) \int_Q e dx dt \leq \int_I E(t) dt \leq 1$. It follows that

$$\int_Q \nu |\nabla u|^2 dx dt \leq \frac{1}{\mu}. \quad (39.9)$$

By standard estimation (assuming that p is bounded), it follows from (39.8) and (39.9) that

$$\|R_m(\hat{u})\|_{-1} \leq C(\sqrt{\mu} + \frac{\sqrt{\nu}}{\sqrt{\mu}}),$$

with C a constant of moderate size, which completes the proof. As indicated, $\|R_m(\hat{u})\|_{-1}$ is estimated by computation, as shown below. The role of the analysis is thus to rationalize computational experience, not to replace it.

39.9 Compression and Expansion

The 2nd Law (39.5) states that there is a transfer of kinetic energy to heat energy if $W < 0$, that is under compression with $\nabla \cdot u < 0$, and a transfer from heat to kinetic energy if $W > 0$, that is under expansion with $\nabla \cdot u > 0$. Returning to Joule's experiment, we see by the 2nd Law that contraction back to the original volume from the final rest state in the double volume, is impossible, because the only way the gas can be set into motion is by expansion. To see this no reference to entropy is needed.

39.10 A 2nd Law without Entropy

We note that the 2nd Law (39.5) is expressed in terms of the physical quantities of kinetic energy K , heat energy E , work W , and dissipation D and does not involve any concept of entropy. This relieves us from the task of finding a physical significance of entropy and justification of a classical 2nd Law stating

that entropy cannot decrease. We thus circumvent the main difficulty of classical thermodynamics based on statistical mechanics, while we reach the same goal as statistical mechanics of explaining irreversibility in formally reversible Newtonian mechanics.

We thus resolve *Loschmidt's paradox* [?] asking how irreversibility can occur in a formally reversible system, which Boltzmann attempted to solve. But Loschmidt pointed out that Boltzmann's equations are not formally reversible, because of the assumption of molecular chaos that velocities are independent before collision, and thus Boltzmann effectively assumes what is to be proved. Boltzmann and Loschmidt's met in heated debates without conclusion, but after Boltzmann's tragic death followed by the experimental verification of the molecular nature of gases, Loschmidt's paradox evaporated as if it had been resolved, while it had not. Postulating molecular chaos still amounts to assume what is to be proved.

39.11 Comparison with Classical Thermodynamics

Classical thermodynamics is based on the relation

$$Tds = dT + pdv, \quad (39.10)$$

where ds represents change of entropy s per unit mass, dv change of volume and dT denotes the change of temperature T per unit mass, combined with a 2nd Law in the form $ds \geq 0$. On the other hand, the new 2nd Law takes the symbolic form

$$dT + pdv \geq 0, \quad (39.11)$$

effectively expressing that $Tds \geq 0$, which is the same as $ds \geq 0$ since $T > 0$. In symbolic form the new 2nd Law thus expresses the same as the classical 2nd Law, without referring to entropy.

Integrating the classical 2nd Law (39.10) for a perfect gas with $p = (\gamma - 1)\rho T$ and $dv = d(\frac{1}{\rho}) = -\frac{d\rho}{\rho^2}$, we get

$$ds = \frac{dT}{T} + \frac{p}{T}d(\frac{1}{\rho}) = \frac{dT}{T} + (1 - \gamma)\frac{d\rho}{\rho},$$

and we conclude that with $e = \rho T$,

$$s = \log(T\rho^{1-\gamma}) = \log(\frac{e}{\rho^\gamma}) = \log(e) - \gamma \log(\rho) \quad (39.12)$$

up to a constant. Thus, the entropy $s = s(\rho, e)$ for a perfect gas is a function of the physical quantities ρ and $e = \rho T$, thus a *state function*, suggesting that s might have a physical significance, because ρ and e have. We thus may decide

to introduce a quantity s defined this way, but the basic questions remains: (i) What is the physical significance of s ? (ii) Why is $ds \geq 0$? What is the entropy non-perfect gas in which case s may not be a state function?

To further exhibit the connection between the classical and new forms of the 2nd Law, we observe that by the chain rule,

$$\rho D_u s = \frac{\rho}{e} D_u e - \gamma D_u \rho = \frac{1}{T} (D_u e + \gamma \rho T \nabla \cdot u) = \frac{1}{T} (D_u e + e \nabla \cdot u + (\gamma - 1) \rho T \nabla \cdot u)$$

since by mass conservation $D_u \rho = -\rho \nabla \cdot u$. It follows that the entropy $S = \rho s$ satisfies

$$\dot{S} + \nabla \cdot (Su) = \rho D_u s = \frac{1}{T} (\dot{e} + \nabla \cdot (eu) + p \nabla \cdot u) = \frac{1}{T} R_e(\hat{u}). \quad (39.13)$$

A solution \hat{u} of the regularized Euler equations (39.3) thus satisfies

$$\dot{S} + \nabla \cdot (Su) = \frac{\nu}{T} |\nabla u|^2 \geq 0 \quad \text{in } Q, \quad (39.14)$$

where $S = \rho \log(e\rho^{-\gamma})$. In particular, in the case of the Joule experiment with T the same in the initial and final states, we have $s = \gamma \log(V)$ showing an increase of entropy in the final state with larger volume.

We sum up by noting that the classical and new form of the second law effectively express the same inequality $ds \geq 0$ or $Tds \geq 0$. The new 2nd law is expressed in terms of the fundamental concepts of kinetic energy, heat energy and work without resort to any form of entropy and statistical mechanics with all its complications. Of course, the new 2nd Law readily extends to the case of a general gas.

39.12 EG2

EG2 in cG(1)cG(1)-form for the Euler equations (39.2), reads: Find $\hat{u} = (\rho, m, \epsilon) \in V_h$ such that for all $(\bar{\rho}, \bar{u}, \bar{\epsilon}) \in W_h$

$$\begin{aligned} ((R_\rho(\hat{u}), \bar{\rho})) + ((hu \cdot \nabla \rho, u \cdot \nabla \bar{\rho})) &= 0, \\ ((R_m(\hat{u}), \bar{u})) + ((hu \cdot \nabla m, u \cdot \nabla \bar{u})) + (\nu_{sc} \nabla u, \nabla \bar{u}) &= 0, \\ ((R_\epsilon(\hat{u}), \bar{\epsilon})) + ((hu \cdot \nabla \epsilon, u \cdot \nabla \bar{\epsilon})) &= 0, \end{aligned} \quad (39.15)$$

where V_h is a trial space of continuous piecewise linear functions on a space-time mesh of size h satisfying the initial condition $\hat{u}(0) = \hat{u}^0$ with $u \in V_h$ defined by nodal interpolation of $\frac{m}{\rho}$, and W_h is a corresponding test space of function which are continuous piecewise linear in space and piecewise constant in time, all functions satisfying the boundary condition $u \cdot n = 0$ at the nodes on Γ . Further, $((\cdot, \cdot))$

denotes relevant $L_2(Q)$ scalar products, and $\nu_{sc} = h^2|R_m(\hat{u})|$ is a residual dependent *shock-capturing viscosity*, see [?]. We here use the conservation equation for the total energy ϵ rather than for the internal energy e .

EG2 combines a weak satisfaction of the Euler equations with a weighted least squares control of the residual $R(\hat{u}) \equiv (R_\rho(\hat{u}), R_m(\hat{u}), R_e(\hat{u}))$ and thus represents a midway between the Scylla of weak solution and Carybdis of least squares strong solution.

39.13 The 2nd Law for EG2

Subtracting the mass equation with $\bar{\rho}$ a nodal interpolant of $\frac{|u|^2}{2}$ from the momentum equation with $\bar{u} = u$ and using the heat energy equation with $\bar{e} = 1$, we obtain the following 2nd Law for EG2 (up to a \sqrt{h} -correction controlled by the shockcapturing viscosity [?]):

$$\dot{K} = W - D_h, \quad \dot{E} = -W + D_h, \quad (39.16)$$

where

$$D_h = ((h\rho u \cdot \nabla u, u \cdot \nabla u)). \quad (39.17)$$

For solutions with turbulence/shocks, $D_h > 0$ expressing an irreversible transfer of kinetic energy into heat energy, just as above for regularized solutions. We note that in EG2 only the momentum equation is subject to viscous regularization, since D_h expresses a penalty on $u \cdot \nabla u$ appearing in the momentum residual.

39.14 The Stabilization in EG2

The stabilization in EG2 is expressed by the dissipative term D_h which can be viewed as a weighted least squares control of the term $\rho u \cdot \nabla u$ in the momentum residual. The rationale is that least squares control of a part of a residual which is large, effectively may give control of the entire residual, and thus EG2 gives a least squares control of the momentum residual. But the EG2 stabilization does not correspond to an ad hoc viscosity, as in classical regularization, but to a form of penalty arising because Euler residuals of turbulent/shock solutions are not pointwise small. In particular the dissipative mechanism of EG2 does not correspond to a simple shear viscosity, but rather to a form of “streamline viscosity” preventing fluid particles from colliding while allowing strong shear.

39.15 Output Uniqueness and Stability

Defining a mean-value output in the form of a space-time integral $((\hat{u}, \psi))$ defined by a smooth weight function ψ , we obtain by duality-based error estimation as in

[?] an a posteriori error estimate of the form

$$|((\hat{u}, \psi)) - ((\hat{w}, \psi))| \leq S(\|R(\hat{u})\|_{-1} + (\|R(\hat{w})\|_{-1})) \leq S(\|hR(\hat{u})\|_0 + (\|hR(\hat{w})\|_0)),$$

where \hat{u} and \hat{w} are two EG2 solutions on meshes of meshsize h , $S = S(\hat{u}, \hat{w})$ is a stability factor defined as the $H^1(Q)$ -norm of a solution to a linearized dual problem with coedd with the data ψ and $\|\cdot\|_0$ is the $L_2(Q)$ -norm. An output $((\hat{u}, \hat{\psi}))$ is *wellposed* if $S\|hR(\hat{u})\|_0 \leq TOL$ with $S = S(\hat{u}, \hat{u})$ and TOL a small tolerance TOL of interest.

In the case shocks/turbulence $\|R(\hat{u})\|_0$ will be large $\sim h^{-1/2}$, while $\|hR(\hat{u})\|_0$ may be small $\sim h^{1/2}$, and an output thus is wellposed if $S \ll h^{-1/2}$. In [?] we present computed stability factors for different weights $\hat{\psi}$, with global support corresponding to global mean-value outputs and local support to pointwise outputs. We find that global mean-values of turbulent flow are wellposed, but local not.

Saddle Point Stability

By Marie E. Rognes

Chapter ref: **[rognes]**

The stability of finite element approximations for abstract saddle point problems has been an active research field for the last four decades. The well-known Babuska- Brezzi conditions provide stability for saddle point problems of the form

$$a(v, u) + b(v, p) + b(u, q) = \langle f, v \rangle + \langle g, q \rangle \quad \forall (v, q) \in V \times Q, \quad (40.1)$$

where a, b are bilinear forms and V, Q Hilbert spaces. For a choice of discrete spaces $V_h \subset V$ and $Q_h \subset Q$, the corresponding discrete Babuska-Brezzi conditions guarantee stability.

However, there are finite element spaces used in practice, with success¹, that do not satisfy the stability conditions in general. The element spaces may satisfy the conditions of certain classes of meshes. Or, there are only a few spurious modes to be filtered out before the method is stable.

The task of determining the stability of a given set of finite element spaces for a given set of equations has mainly been a manual task. However, the flexibility of the FEniCS components has made automation feasible.

For each set of discrete spaces, the discrete Brezzi conditions can be equivalently formulated in terms of an eigenvalue problem. For instance, [...] where B is the element matrix associated with the form b and M, N are the matrices induced by the inner-products on V and Q respectively. Hence, the stability of a set of finite element spaces on a type of meshes can be tested numerically by solving a series of eigenvalue problems.

A small library FEAST (Finite Element Automatic Stability Tester) has been built on top of the core FEniCS components, providing automated functionality

for the testing of finite element spaces for a given equation on given meshes. With some additional input, convergence rates and in particular optimal choices of element (in some measure such as error per degrees of freedom) can be determined.

In this note, the functionality provided by FEAST is explained and results for equations such as the Stokes equations, Darcy flow and mixed elasticity are demonstrated.

References

- [AB85] D. N. Arnold and F. Brezzi. Mixed and nonconforming finite element methods: implementation, postprocessing and error estimates. *RAIRO Modél. Math. Anal. Numér.*, 19(1):7–32, 1985.
- [ABD84] Douglas N. Arnold, Franco Brezzi, and Jim Douglas, Jr. PEERS: a new mixed finite element for plane elasticity. *Japan J. Appl. Math.*, 1(2):347–367, 1984.
- [ADK⁺98] Todd Arbogast, Clint N. Dawson, Philip T. Keenan, Mary F. Wheeler, and Ivan Yotov. Enhanced cell-centered finite differences for elliptic equations on general geometry. *SIAM J. Sci. Comput.*, 19(2):404–425 (electronic), 1998.
- [AF89] Douglas N. Arnold and Richard S. Falk. A uniformly accurate finite element method for the Reissner–Mindlin plate. *SIAM J. Num. Anal.*, 26:1276–1290, 1989.
- [AFS68] J.H. Argyris, I. Fried, and D.W. Scharpf. The TUBA family of plate elements for the matrix displacement method. *The Aeronautical Journal of the Royal Aeronautical Society*, 72:701–709, 1968.
- [AFW06a] D.N. Arnold, R.S. Falk, and R. Winther. Finite element exterior calculus, homological techniques, and applications. *Acta Numerica*, 15:1–155, 2006.
- [AFW06b] Douglas N. Arnold, Richard S. Falk, and Ragnar Winther. Differential complexes and stability of finite element methods. II. The elasticity complex. In *Compatible spatial discretizations*, volume 142 of *IMA Vol. Math. Appl.*, pages 47–67. Springer, New York, 2006.

- [AFW07] Douglas N. Arnold, Richard S. Falk, and Ragnar Winther. Mixed finite element methods for linear elasticity with weakly imposed symmetry. *Math. Comp.*, 76(260):1699–1723 (electronic), 2007.
- [AGSNR80] J. C. Adam, A. Gourdin Serveniére, J.-C. Nédélec, and P.-A. Raviart. Study of an implicit scheme for integrating Maxwell’s equations. *Comput. Methods Appl. Mech. Engrg.*, 22(3):327–346, 1980.
- [AM09] M. S. Alnæs and K.-A. Mardal. *SyFi*, 2009. <http://www.fenics.org/wiki/SyFi/>.
- [AO93] M. Ainsworth and J.T. Oden. A unified approach to a posteriori error estimation using element residual methods. *Numerische Mathematik*, 65(1):23–50, 1993.
- [AO00] Mark Ainsworth and J. Tinsley Oden. *A Posteriori Error Estimation in Finite Element Analysis*. Wiley and Sons, New York, 2000.
- [AW02] Douglas N. Arnold and Ragnar Winther. Mixed finite elements for elasticity. *Numer. Math.*, 92(3):401–419, 2002.
- [BBE⁺04] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.
- [BCO81] E. B. Becker, G. F. Carey, and J. T. Oden. *Finite Elements: An Introduction*. Prentice–Hall, Englewood–Cliffs, 1981.
- [BDM85a] F. Brezzi, J. Douglas, Jr., and L. D. Marini. Variable degree mixed methods for second order elliptic problems. *Mat. Apl. Comput.*, 4(1):19–34, 1985.
- [BDM85b] Franco Brezzi, Jim Douglas, Jr., and L. D. Marini. Two families of mixed finite elements for second order elliptic problems. *Numer. Math.*, 47(2):217–235, 1985.
- [BF91] Franco Brezzi and Michel Fortin. *Mixed and hybrid finite element methods*, volume 15 of *Springer Series in Computational Mathematics*. Springer-Verlag, New York, 1991.
- [BR78] I. Babuška and W. C. Rheinboldt. A posteriori error estimates for the finite element method. *Int. J. Numer. Meth. Engrg.*, pages 1597–1615, 1978.

- [BR01] R. Becker and R. Rannacher. An optimal control approach to a posteriori error estimation in finite element methods. *Acta Numerica*, 10:1–102, 2001.
- [Bra77] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, pages 333–390, 1977.
- [Bra07] Dietrich Braess. *Finite elements*. Cambridge University Press, Cambridge, third edition, 2007. Theory, fast solvers, and applications in elasticity theory, Translated from the German by Larry L. Schumaker.
- [BS94] S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods*. Springer-Verlag, 1994.
- [BS08] Susanne C. Brenner and L. Ridgway Scott. *The mathematical theory of finite element methods*, volume 15 of *Texts in Applied Mathematics*. Springer, New York, third edition, 2008.
- [BW85] RE Bank and A. Weiser. Some a posteriori error estimators for elliptic partial differential equations. *Mathematics of Computation*, pages 283–301, 1985.
- [Cas79] C. A. P. Castigliano. *Théorie de l'équilibre des systèmes élastiques et ses applications*. A.F. Negro ed., Torino, 1879.
- [CF89] M. Crouzeix and Richard S. Falk. Nonconforming finite elements for the stokes problem. *Mathematics of Computation*, 52:437–456, 1989.
- [Cia75] P.G. Ciarlet. Lectures on the finite element method. *Lectures on Mathematics and Physics, Tata Institute of Fundamental Research, Bombay*, 49, 1975.
- [Cia76] P. G. Ciarlet. *Numerical Analysis of the Finite Element Method*. Les Presses de l'Université de Montréal, 1976.
- [Cia78] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. North-Holland, Amsterdam, New York, Oxford, 1978.
- [Cia02] Philippe G. Ciarlet. *The finite element method for elliptic problems*, volume 40 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2002. Reprint of the 1978 original [North-Holland, Amsterdam; MR0520174 (58 #25001)].

- [Cou43] R. Courant. Variational methods for the solution of problems of equilibrium and vibrations. *Bull. Amer. Math. Soc.*, pages 1–23, 1943.
- [CR72] P. G. Ciarlet and P.-A. Raviart. General Lagrange and Hermite interpolation in \mathbb{R}^n with applications to finite element methods. *Arch. Rational Mech. Anal.*, 46:177–199, 1972.
- [CR73] M. Crouzeix and P A Raviart. Conforming and non-conforming finite element methods for solving the stationary Stokes equations. *R.A.I.R.O Anal. Numer.*, 7:33–76, 1973.
- [Dav04] T.A. Davis. Algorithm 832: UMFPACK V4. 3an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.
- [EEHJ95] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. Introduction to adaptive methods for differential equations. *Acta Numerica*, 4:105–158, 1995.
- [EJ] K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems III: Time steps variable in space. *in preparation*.
- [EJ91] K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems I: A linear model problem. *SIAM J. Numer. Anal.*, 28, No. 1:43–77, 1991.
- [EJ95a] K. Eriksson and C. Johnson. Adaptive Finite Element Methods for Parabolic Problems II: Optimal Error Estimates in L_2 and L_∞ . *SIAM Journal on Numerical Analysis*, 32:706, 1995.
- [EJ95b] K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems IV: Nonlinear problems. *SIAM Journal on Numerical Analysis*, pages 1729–1749, 1995.
- [EJ95c] K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems V: Long-time integration. *SIAM J. Numer. Anal.*, 32:1750–1763, 1995.
- [EJL98] K. Eriksson, C. Johnson, and S. Larsson. Adaptive finite element methods for parabolic problems VI: Analytic semigroups. *SIAM J. Numer. Anal.*, 35:1315–1325, 1998.
- [Fel66] C.A. Felippa. *Refined Finite Element Analysis of Linear and Non-linear Two-dimensional Structures*. PhD thesis, The University of California at Berkeley, 1966.

- [FY02] R.D. Falgout and U.M. Yang. Hypre: A library of high performance preconditioners. In *Proceedings of the International Conference on Computational Science-Part III*, pages 632–641. Springer-Verlag London, UK, 2002.
- [Gal15] B. G. Galerkin. Series solution of some problems in elastic equilibrium of rods and plates. *Vestnik inzhenerov i tekhnikov*, 19:897–908, 1915.
- [HBH⁺05] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.
- [HL03] Peter Hansbo and Mats G. Larson. Discontinuous Galerkin and the Crouzeix–Raviart element: application to elasticity. *Math. Model. Numer. Anal.*, 37:63–72, 2003.
- [HS52] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand*, 49(6):409–436, 1952.
- [Hug87] T. J. R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall, 1987.
- [Kir04] R. C. Kirby. FIAT: A new paradigm for computing finite element basis functions. *ACM Trans. Math. Software*, 30:502–516, 2004.
- [Li05] X.S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):302–325, 2005.
- [LL75] P. Lascaux and P. Lesaint. Some nonconforming finite elements for the plate bending problem. *Rev. Française Automat. Informat. Recherche Operationnelle RAIRO Analyse Numérique*, 9(R-1):9–53, 1975.
- [LM54] PD Lax and AN Milgram. Parabolic equations. *Annals of Mathematics Studies*, 33:167–190, 1954.
- [Log07] Anders Logg. Efficient representation of computational meshes. In B. Skallerud and H. I. Andersson, editors, *MekIT⁰⁷*. Tapir Akademisk Forlag, 2007. <http://www.ntnu.no/meKIT07>, <http://butikk.tapirforlag.no/no/node/1073>.

- [Mor68] L.S.D. Morley. The triangular equilibrium element in the solution of plate bending problems. *Aero. Quart.*, 19:149–169, 1968.
- [OD96] J.T. Oden and L. Demkowicz. *Applied functional analysis*. CRC press, 1996.
- [Ray70] Rayleigh. On the theory of resonance. *Trans. Roy. Soc.*, A161:77–118, 1870.
- [Rit08] W. Ritz. Über eine neue Methode zur Lösung gewisser Variationprobleme der mathematischen Physik. *J. reine angew. Math.*, 135:1–61, 1908.
- [RKL08] M. Rognes, Robert C. Kirby, and Anders Logg. Efficient assembly of $h(\text{div})$ and $h(\text{curl})$ conforming finite elements. *SIAM J. Sci. Comput.*, 2008. submitted by Marie 2008-10-24, resubmitted by Marie 2009-05-22.
- [RT77] P.-A. Raviart and J. M. Thomas. A mixed finite element method for 2nd order elliptic problems. In *Mathematical aspects of finite element methods (Proc. Conf., Consiglio Naz. delle Ricerche (C.N.R.), Rome, 1975)*, pages 292–315. Lecture Notes in Math., Vol. 606. Springer, Berlin, 1977.
- [RW83] T. F. Russell and M. F. Wheeler. *The Mathematics of Reservoir Simulation*, volume 1 of *Frontiers Applied Mathematics*, chapter Finite element and finite difference methods for continuous flow in porous media, pages 35–106. SIAM, 1983.
- [SF73] G. Strang and G. J. Fix. *An Analysis of the Finite Element Method*. Prentice-Hall, Englewood Cliffs, 1973.
- [SS86] Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986.
- [Ver94] R. Verfürth. A posteriori error estimation and adaptive mesh-refinement techniques. In *Proceedings of the fifth international conference on Computational and applied mathematics table of contents*, pages 67–83. Elsevier Science Publishers BV Amsterdam, The Netherlands, The Netherlands, 1994.
- [Ver99] R. Verfürth. A review of a posteriori error estimation techniques for elasticity problems. *Computer Methods in Applied Mechanics and Engineering*, 176(1-4):419–440, 1999.

- [Wes92] P. Wesseling. *An introduction to multigrid methods*. Wiley Chichester, 1992.
- [ZTZ67] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu. *The Finite Element Method — Its Basis and Fundamentals, 6th edition*. Elsevier, 2005, first published in 1967.
- [ZZ87] OC Zienkiewicz and JZ Zhu. A simple error estimator and adaptive procedure for practical engineering analysis. *International Journal for Numerical Methods in Engineering*, 24(2), 1987.

The following notation is used throughout this book.

- A – the *global tensor* with entries $\{A_i\}_{i \in \mathcal{I}}$
- A^K – the *element tensor* with entries $\{A_i^K\}_{i \in \mathcal{I}_K}$
- A^0 – the *reference tensor* with entries $\{A_{i\alpha}^0\}_{i \in \mathcal{I}_K, \alpha \in \mathcal{A}}$
- a – a multilinear form
- a_K – the local contribution to a multilinear form a from a cell K
- \mathcal{A} – the set of *secondary indices*
- \mathcal{B} – the set of *auxiliary indices*
- e – the *error*, $e = u_h - u$
- F_K – the mapping from the reference cell K_0 to K
- G_K – the *geometry tensor* with entries $\{G_K^\alpha\}_{\alpha \in \mathcal{A}}$
- \mathcal{I} – the set $\prod_{j=1}^{\rho} [1, N^j]$ of indices for the global tensor A
- \mathcal{I}_K – the set $\prod_{j=1}^{\rho} [1, n_K^j]$ of indices for the element tensor A^K (*primary indices*)
- ι_K – the *local-to-global mapping* from $[1, n_K]$ to $[1, N]$
- K – a *cell* in the mesh \mathcal{T}
- K_0 – the *reference cell*
- L – a linear form (functional) on \hat{V} or \hat{V}_h
- \mathcal{L} – the degrees of freedom (linear functionals) on V_h
- \mathcal{L}_K – the degrees of freedom (linear functionals) on \mathcal{P}_K
- \mathcal{L}_0 – the degrees of freedom (linear functionals) on \mathcal{P}_0
- N – the dimension of \hat{V}_h and V_h
- n_K – the dimension of \mathcal{P}_K
- ℓ_i – a degree of freedom (linear functional) on V_h
- ℓ_i^K – a degree of freedom (linear functional) on \mathcal{P}_K
- ℓ_i^0 – a degree of freedom (linear functional) on \mathcal{P}_0
- \mathcal{P}_K – the local function space on a cell K
- \mathcal{P}_0 – the local function space on the reference cell K_0

- $P_q(K)$ – the space of polynomials of degree $\leq q$ on K
- r – the (weak) residual, $r(v) = a(v, u_h) - L(v)$ or $r(v) = F(u_h; v)$
- u_h – the finite element solution, $u_h \in V_h$
- U – the vector of degrees of freedom for $u_h = \sum_{i=1}^N U_i \phi_i$
- u – the exact solution of a variational problem, $u \in V$
- \hat{V} – the test space
- V – the trial space
- \hat{V}^* – the dual test space, $\hat{V}^* = V_0$
- V^* – the dual trial space, $V^* = \hat{V}$
- \hat{V}_h – the discrete test space
- V_h – the discrete trial space
- ϕ_i – a basis function in V_h
- $\hat{\phi}_i$ – a basis function in \hat{V}_h
- ϕ_i^K – a basis function in \mathcal{P}_K
- Φ_i – a basis function in \mathcal{P}_0
- z – the *dual solution*, $z \in V^*$
- \mathcal{T} – the *mesh*, $\mathcal{T} = \{K\}$
- Ω – a bounded domain in \mathbb{R}^d

List of Authors

- ▶ Editor note: *Include author affiliations here.*
- ▶ Editor note: *Sort alphabetically by last name, not first.*

- Anders Logg
- Andy R. Terrel
- Bjørn Fredrik Nielsen
- Claes Johnson
- David B. Davidson
- Emil Alf Løvgren
- Evan Lezar
- Garth N. Wells
- Hans Joachim Schroll
- Hans Petter Langtangen
- Ilmar M. Wilbers
- Joakim Sundnes
- Johan Hake
- Johan Hoffman
- Johan Jansson

- Kent-Andre Mardal
- Kristian B. Ølgaard
- Kristian Valen-Sendstad
- Kristoffer Selim
- L. Ridgway Scott
- L. Trabucho
- Luca Antiga
- Marie E. Rognes
- Martin S. Alnæs
- Matthew G. Knepley
- Mehdi Nikbakht
- Murtazo Nazarov
- Niclas Jansson
- N. Lopes
- Oddrun Christine Myklebust
- Ola Skavhaug
- P. Pereira
- Robert C. Kirby
- Susanne Hentschel
- Svein Linge
- Xuming Shan

∇ , 165
instant-clean, 136
instant-showcache, 136
BasisFunctions, 161
BasisFunction, 161
ComponentTensor, 162
Constant, 161
Dx, 165
Expr, 172
FiniteElement, 156
Form, 159
Functions, 161
Function, 161
Identity, 160
IndexSum, 162
Indexed, 162
Index, 162
Integral, 159
ListTensor, 162
Measure, 159
MixedElement, 156
Operator, 172
TensorConstant, 161
TensorElement, 156
Terminal, 160, 172
TestFunctions, 161
TestFunction, 161
TrialFunctions, 161
TrialFunction, 161
VectorConstant, 161
VectorElement, 156
action, 168
adjoint, 168
as_matrix, 162
as_tensor, 162
as_vector, 162
cos, 164
cross, 164
curl, 165
derivative, 168
det, 164
diff, 165
div, 165
dot, 164
dx, 165
energy_norm, 168
exp, 164
grad, 165
indices, 162
inner, 164
inv, 164
lhs, 168
ln, 164
outer, 164
pow, 164
replace, 168
rhs, 168
rot, 165
sensitivity_rhs, 168
sin, 164

split, 161
 sqrt, 164
 system, 168
 transpose, 164
 tr, 164
 UFL, 153

AD, 176
 Adaptive mesh refinement, 94
 algebraic operators, 164
 algorithms, 183
 atomic value, 160
 Automatic Differentiation, 176
 avg, 167

basis function, 160
 basis functions, 161
 boundary conditions, 256
 boundary measure, 159
 Boussinesq models, 247

Cache, 140
 cell integral, 159
 cerebrospinal fluid, 221
 Chiari I malformation, 221
 coefficient function, 160
 coefficient functions, 161
 coefficients, 161
 computational graph, 175
 computing derivatives, 176
 cross product, 164
 CSF, 221
 Cython, 128

derivatives, 176
 determinant, 164
 DG operators, 167
 differential operators, 165
 differentiation, 176
 discontinuous Galerkin, 167
 discontinuous Lagrange element, 156
 dispersion curves, 296, 301, 303, 305
 dispersion relation, 254
 Distutils, 127

domain specific language, 153
 dot product, 164
 Dynamic load balancing, 97

eigenvalue problem, 292, 293
 electromagnetics, 289
 expression, 172
 expression representations, 186
 expression transformations, 185, 186
 expression tree, 172
 expression trees, 183
 exterior facet integral, 159

F2PY, 128
 facet normal, 160
 finite element, 156
 finite element space, 156
 foramen magnum, 221
 form argument, 160
 form arguments, 161
 form language, 153
 form operators, 168
 forms, 159
 forward mode AD, 176
 functional, 153
 functions, 161

identity matrix, 160
 implicit summation, 162
 index notation, 162
 indices, 162
 inner product, 164
 integrals, 159
 interior facet integral, 159
 interior measure, 159
 inverse, 164

jump, 167

Lagrange element, 156
 language operators, 164

Maxwell's equations, 289
 microstrip, *see* shielded microstrip
 multifunctions, 183

- Navier-Stokes, 223
- Newtonian fluid, 223

- OpenMP, 132
- operator, 172
- outer product, 164

- Parallel radix sort, 99
- potential, 249
- Predictor-Corrector, 258
- program, 172
- propagation constant, 290

- referential transparency, 172
- reflective boundaries, 257
- restriction, 167
- reverse mode AD, 176
- Runge-Kutta, 258

- S-parameters, 305–307
- SAS, 221
- scattering parameters, *see* S-parameters
- shielded microstrip, 303
- Signature, 140
- signatures, 189
- source function, 256
- spatial coordinates, 160
- spinal canal, 221
- spinal cord, 222
- sponge-layers, 257
- spurious modes, 293
- subarachnoid space, 221
- SWIG, 127
- symbolic differentiation, 176

- tensor algebra operators, 164
- terminal value, 160, 172
- trace, 164
- transpose, 164
- tree traversal, 183
- Typemap, 128, 136

- UFL, 190
- Unified Form Language, 153

- variational form, 153
- vector Helmholtz equation, 290

- water waves, 247
- waveguide, 289
 - cutoff analysis, 291, 299, 302
 - discontinuities, 305–307
 - dispersion analysis, 293, 301, 302, 304
 - half-loaded rectangular, 301–303
 - hollow rectangular, 298–301, 306
- wavenumber
 - cutoff, 292, 296, 298, 299, 304
 - operating, 290, 292
- weak form, 153
- Weave, 128
- Windkessel model, 130