

# PETSc for Partial Differential Equations

---

# SOFTWARE • ENVIRONMENTS • TOOLS

---

The SIAM series on Software, Environments, and Tools focuses on the practical implementation of computational methods and the high performance aspects of scientific computation by emphasizing in-demand software, computing environments, and tools for computing. Software technology development issues such as current status, applications and algorithms, mathematical software, software tools, languages and compilers, computing environments, and visualization are presented.

---

## Editor-in-Chief

Jack J. Dongarra

University of Tennessee and Oak Ridge National Laboratory

## Series Editors

### Timothy A. Davis

Texas A&M University

### Laura Grigori

INRIA

### Padma Raghavan

Pennsylvania State University

### James W. Demmel

University of California,  
Berkeley

### Michael A. Heroux

Sandia National Laboratories

### Yves Robert

ENS Lyon

## Software, Environments, and Tools

Ed Bueler, *PETSc for Partial Differential Equations: Numerical Solutions in C and Python*

D. L. Chopp, *Introduction to High Performance Scientific Computing*

Thomas Huckle and Tobias Neckel, *Bits and Bugs: A Scientific and Historical Review of Software Failures in Computational Science*

Thomas F. Coleman and Wei Xu, *Automatic Differentiation in MATLAB Using ADMAT with Applications*

Walter Gautschi, *Orthogonal Polynomials in MATLAB: Exercises and Solutions*

Daniel J. Bates, Jonathan D. Hauenstein, Andrew J. Sommese, and Charles W. Wampler, *Numerically Solving Polynomial Systems with Bertini*

Uwe Naumann, *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*

C. T. Kelley, *Implicit Filtering*

Jeremy Kepner and John Gilbert, editors, *Graph Algorithms in the Language of Linear Algebra*

Jeremy Kepner, *Parallel MATLAB for Multicore and Multinode Computers*

Michael A. Heroux, Padma Raghavan, and Horst D. Simon, editors, *Parallel Processing for Scientific Computing*

Gérard Meurant, *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations*

Bo Einarsson, editor, *Accuracy and Reliability in Scientific Computing*

Michael W. Berry and Murray Browne, *Understanding Search Engines: Mathematical Modeling and Text Retrieval, Second Edition*

Craig C. Douglas, Gundolf Haase, and Ulrich Langer, *A Tutorial on Elliptic PDE Solvers and Their Parallelization*

Louis Komzsik, *The Lanczos Method: Evolution and Application*

Bard Ermentrout, *Simulating, Analyzing, and Animating Dynamical Systems: A Guide to XPPAUT for Researchers and Students*

V. A. Barker, L. S. Blackford, J. Dongarra, J. Du Croz, S. Hammarling, M. Marinova, J. Wasniewski, and P. Yalamov, *LAPACK95 Users' Guide*

Stefan Goedecker and Adolfy Hoisie, *Performance Optimization of Numerically Intensive Codes*

Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*

Lloyd N. Trefethen, *Spectral Methods in MATLAB*

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide, Third Edition*

Michael W. Berry and Murray Browne, *Understanding Search Engines: Mathematical Modeling and Text Retrieval*

Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst, *Numerical Linear Algebra for High-Performance Computers*

R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*

Randolph E. Bank, *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations, Users' Guide 8.0*

L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*

Greg Astfalk, editor, *Applications on Advanced Architecture Computers*

Roger W. Hockney, *The Science of Computer Benchmarking*

Françoise Chaitin-Chatelin and Valérie Frayssé, *Lectures on Finite Precision Computations*

# PETSc for Partial Differential Equations

Numerical Solutions in C and Python

**Ed Bueler**

University of Alaska Fairbanks  
Fairbanks, Alaska



Society for Industrial and Applied Mathematics  
Philadelphia

Copyright © 2021 by the Society for Industrial and Applied Mathematics

10 9 8 7 6 5 4 3 2 1

All rights reserved. Printed in the United States of America. No part of this book may be reproduced, stored, or transmitted in any manner without the written permission of the publisher. For information, write to the Society for Industrial and Applied Mathematics, 3600 Market Street, 6th Floor, Philadelphia, PA 19104-2688 USA.

No warranties, express or implied, are made by the publisher, authors, and their employers that the programs contained in this volume are free of error. They should not be relied on as the sole basis to solve a problem whose incorrect solution could result in injury to person or property. If the programs are employed in such a manner, it is at the user's own risk and the publisher, authors, and their employers disclaim all liability for such misuse.

Trademarked names may be used in this book without the inclusion of a trademark symbol. These names are used in an editorial context only; no infringement of trademark is intended.

MATLAB is a registered trademark of The MathWorks, Inc. For MATLAB product information, please contact The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098 USA, 508-647-7000, Fax: 508-647-7001, [info@mathworks.com](mailto:info@mathworks.com), [www.mathworks.com](http://www.mathworks.com).

Python is a registered trademark of Python Software Foundation.

ParaView is a registered trademark of Kitware Inc.

MPI is a registered trademark of Argonne National Laboratory.

PETSc is a registered copyright © 1991-2014 of UChicago Argonne, LLC and the PETSc Development Team.

Gmsh is a registered copyright © 1997-2020 of C. Geuzaine and J.-F. Remacle.

Firedrake is a registered copyright of Imperial College London.

<i>Publications Director</i>	Kivmars H. Bowling
<i>Acquisitions Editor</i>	Elizabeth Greenspan
<i>Developmental Editor</i>	Mellisa Pascale
<i>Managing Editor</i>	Kelly Thomas
<i>Production Editor</i>	Lisa Briggeman
<i>Copy Editor</i>	Matthew Bernard
<i>Production Manager</i>	Donna Witzleben
<i>Production Coordinator</i>	Cally A. Shrader
<i>Compositor</i>	Cheryl Hufnagle
<i>Graphic Designer</i>	Doug Smock

### Library of Congress Cataloging-in-Publication Data

Names: Bueler, Edward L. (Edward Lee), author.

Title: PETSc for partial differential equations : numerical solutions in C and Python / Ed Bueler, University of Alaska, Fairbanks, Alaska.

Description: Philadelphia : Society for Industrial and Applied Mathematics, [2020] | Series: Software, environments, and tools; 31 | Includes bibliographical references and index. | Summary: "PETSc for Partial Differential Equations is the first textbook to cover PETSc programming for nonlinear PDEs"— Provided by publisher.

Identifiers: LCCN 2020027102 (print) | LCCN 2020027103 (ebook) | ISBN 9781611976304 (paperback) | ISBN 9781611976311 (ebook)

Subjects: LCSH: Differential equations. | Partial-Computer programs. | Numerical analysis. | Parallel programming (Computer science) | C (Computer program language) | Python (Computer program language)

Classification: LCC QA377 .B84 2020 (print) | LCC QA377 (ebook) | DDC 515/.353028553–dc23

LC record available at <https://lccn.loc.gov/2020027102>

LC ebook record available at <https://lccn.loc.gov/2020027103>

# Contents

Preface	ix
<b>I      Essentials</b>	<b>1</b>
<b>1    Getting started with PETSc</b>	<b>3</b>
A code that does almost nothing, but in parallel . . . . .	3
Compiling and running the first code . . . . .	4
Every PETSC program . . . . .	6
Exploring the example codes . . . . .	7
Exercises . . . . .	8
<b>2    Finite-dimensional linear systems</b>	<b>9</b>
Vectors, matrices, and norms . . . . .	9
Linear systems, and some facts of (numerical) life . . . . .	10
Residuals and errors . . . . .	12
Richardson iteration . . . . .	12
A first look at preconditioning . . . . .	14
Krylov space methods . . . . .	17
Chebyshev iteration . . . . .	21
PETSC objects . . . . .	23
Parallel layout of <code>Vec</code> and <code>Mat</code> objects . . . . .	24
Assemble and view a <code>Mat</code> . . . . .	25
A small linear system . . . . .	27
Revealing solvers at run time . . . . .	29
A tridiagonal linear system . . . . .	31
Saving and loading linear systems from files . . . . .	35
Parallel preconditioning . . . . .	38
Exercises . . . . .	38
<b>3    Poisson equation on a structured grid</b>	<b>43</b>
Poisson problem on a square domain . . . . .	43
Creating structured grids . . . . .	44
A finite difference method . . . . .	47
Structured-grid matrix assembly . . . . .	50
A particular problem . . . . .	52
Solving the PDE . . . . .	53
Run-time visualization . . . . .	54

---

Convergence in practice and theory . . . . .	55
A first look at performance . . . . .	59
Scaling of preconditioned CG iterations . . . . .	60
Krylov is not enough: Better preconditioning is needed! . . . . .	63
Exercises . . . . .	65
<b>4 Nonlinear equations by Newton's method</b>	<b>67</b>
Newton's method . . . . .	67
Inside the first SNES code . . . . .	68
Convergence of the Newton iteration . . . . .	72
User-supplied Jacobians . . . . .	74
A nonlinear diffusion-reaction equation . . . . .	78
A numerical method and its convergence . . . . .	78
Finite-differenced Jacobians by coloring . . . . .	83
Jacobian-free Newton-Krylov (JFNK) . . . . .	86
Testing Jacobian cases . . . . .	88
Line-search methods . . . . .	90
Exercises . . . . .	92
<b>5 Time-stepping</b>	<b>95</b>
Systems of ordinary differential equations (ODEs) . . . . .	95
Numerical methods for ODE initial value problems . . . . .	96
Higher-order and adaptive methods . . . . .	98
A first TS example . . . . .	100
Controlling TS . . . . .	102
Implicit methods and stiffness . . . . .	103
Absolute stability . . . . .	105
Jacobians for implicit methods . . . . .	109
A time-dependent heat equation problem . . . . .	111
Method of lines . . . . .	112
Visualization and performance . . . . .	117
Coupled reaction-diffusion equations . . . . .	120
Generating patterns . . . . .	125
Exercises . . . . .	127
<b>6 Preconditioners for PDEs</b>	<b>129</b>
Preconditioners in PETSc . . . . .	130
Classical iterations . . . . .	131
Smothers . . . . .	135
Restricting to subgrids . . . . .	137
Subgrid corrections and their compositions . . . . .	141
A better Poisson code . . . . .	143
Single-level domain decomposition . . . . .	147
Coarse grids . . . . .	152
Geometric multigrid . . . . .	156
Multigrid cycle types and costs . . . . .	159
Controlling multigrid . . . . .	160
Exercises . . . . .	165

<b>Interlude: Quadrature</b>	<b>171</b>
Exercises . . . . .	174
<b>7 Optimal solvers for elliptic PDEs</b>	<b>175</b>
Solver complexity . . . . .	175
An optimal solver for the Poisson equation . . . . .	177
Parallel multigrid and the coarse-grid problem . . . . .	178
The minimal surface equation . . . . .	183
Grid sequencing . . . . .	186
A SNES monitor for the minimal surface equation . . . . .	188
The biharmonic equation as a coupled system . . . . .	189
Block-structured preconditioning . . . . .	191
Exercises . . . . .	194
<b>8 Parallel scaling</b>	<b>199</b>
Consumable resources on clusters . . . . .	199
The <code>streams</code> benchmark . . . . .	202
The classic language of speedup . . . . .	203
Weak and static scaling . . . . .	207
Toward ideal scaling . . . . .	210
Caveats . . . . .	212
Parallel multigrid with PCTelescope . . . . .	213
Parallel nondeterminacy . . . . .	214
Exercises . . . . .	215
<b>II Constructions</b>	<b>217</b>
<b>9 Finite element method I: Nonlinear optimization</b>	<b>219</b>
A $p$ -Helmholtz equation as minimization . . . . .	219
Structured $Q_1$ finite elements . . . . .	221
Objective only: Implementation . . . . .	225
Objective only: Solvers . . . . .	228
Using a gradient function . . . . .	230
Regularization for $p \neq 2$ . . . . .	233
Convergence and (near) optimality . . . . .	235
Exercises . . . . .	237
<b>10 Finite element method II: Naive and unstructured</b>	<b>243</b>
A nonlinear Poisson problem . . . . .	243
Unstructured $P_1$ finite elements . . . . .	245
Assembly of the residual equations . . . . .	247
Meshes from Gmsh . . . . .	249
Loading the mesh into PETSc data structures . . . . .	252
Initial implementation and testing . . . . .	254
Picard iteration as a Newton-like step . . . . .	259
Preallocating and assembling the matrix . . . . .	260
Convergence and profiling . . . . .	263
Algebraic multigrid . . . . .	266
Nonlinear performance . . . . .	269

Performance relative to a DMDA structured grid . . . . .	269
Poisson equation on the Koch snowflake . . . . .	272
Toward mature unstructured-mesh FE method capabilities . . . . .	274
Exercises . . . . .	275
<b>11 Advection without, and then with, diffusion</b>	<b>279</b>
Flux conservation and finite volumes . . . . .	279
Upwinding, and the goals of advection schemes . . . . .	282
High-resolution flux discretizations . . . . .	286
Implementation in 2D . . . . .	289
Advection results . . . . .	294
Implicit time-stepping for advection . . . . .	297
Steady-state advection-diffusion problems . . . . .	298
Advection is not stagnation . . . . .	301
Multigrid for advection-diffusion problems . . . . .	305
Exercises . . . . .	309
<b>12 Inequality constraints</b>	<b>315</b>
The classical obstacle problem . . . . .	315
Newton solvers for bound-constrained problems . . . . .	319
Newton-multigrid and grid sequencing . . . . .	322
Optimal and scalable solver combinations . . . . .	324
Exercises . . . . .	327
<b>13 Finite element method III: Firedrake and DMPlex</b>	<b>331</b>
The Poisson equation (one last time) . . . . .	332
Polynomial-degree refinement . . . . .	335
The underlying DMPlex object . . . . .	337
Exercises . . . . .	340
<b>14 Stokes equations (with Firedrake)</b>	<b>343</b>
Incompressible viscous fluids . . . . .	343
Mixed FE methods and the discrete equations . . . . .	347
A Stokes flow code . . . . .	348
The discrete Stokes matrix and its spectral personality . . . . .	352
Preconditioning a symmetric, indefinite, and block system . . . . .	354
Stable elements and Schur complements . . . . .	357
Options for Schur+GMG preconditioners . . . . .	359
Convergence and solver performance . . . . .	361
Moffatt eddies . . . . .	365
Exercises . . . . .	366
<b>Appendix. Some numerical facts of life</b>	<b>371</b>
<b>Bibliography</b>	<b>373</b>
<b>Index</b>	<b>383</b>

# Preface

This book is about solving partial differential equations (PDEs) numerically by writing C and Python codes that call PETSc,<sup>1</sup> the Portable, Extensible Toolkit for Scientific computation [10, 11]. Concepts are explained and illustrated through examples, with sufficient context to facilitate further development. Because demonstrable performance and scalability are the primary goals, run-time options are explored and compared in both the text and the exercises.

The inside front cover lists the major PDE examples, ordered by ease of introduction of the algorithms and components from PETSc. The focus of this book is on PDE problems wherein discretization leads to large, generally nonlinear systems of algebraic equations. Such systems are either solved once to compute a steady state solution or at each time step in an implicit time-stepping scheme.

Many of the concepts and algorithms herein have become common knowledge among experts, but scientific computing moves forward when access to advanced techniques is expanded. An expert in PDEs and PETSc might say about this book *both* “I know that,” for most of the content, and “the book is a fast on-ramp to what I already know.”

So, suppose you have taken courses in differential equations and linear algebra, have written a few codes in C, and you are interested in numerically solving nonlinear PDE problems in parallel using advanced algorithms. Then this book is for you.

## What is PETSc?

PETSc, pronounced “PET-see,” is an open-source mathematical software library built on top of the standard software layer for distributed-memory parallel computation, namely the Message Passing Interface (MPI) [72]. Both are developed by the Mathematics and Computer Science Division of the Argonne National Laboratory, within the U.S. Department of Energy. They form a framework capable of solving problems, such as discretized nonlinear PDEs, with millions or billions of degrees of freedom, on supercomputers with thousands of cores. But PETSc codes will also run on a standard laptop or workstation, which is where all examples from this book should be tried first.

PETSc is not particularly new. A well-known monograph [134] from the 1990s used PETSc 2.0 for scalable solutions of linear PDEs, focusing on preconditioned iterative linear solvers and domain decomposition. However, PETSc is under active development, and it is now at version 3.13. It has evolved into a powerful toolbox with a large API (application program interface). The PETSc strategy is to combine (“compose”), at run time, grid/mesh distribution tools, time-stepping schemes for ordinary differential equations (ODEs), nonlinear iterations, iterative linear

---

<sup>1</sup>The examples in the text and the source codes at the repository ([github.com/bueler/p4pdes](https://github.com/bueler/p4pdes), or [bookstore.siam.org/se31/bonus](http://bookstore.siam.org/se31/bonus) which redirects to the same place) were tested with PETSc version 3.13. The repository codes will be maintained for future PETSc versions.

solvers, and a rich selection of preconditioners, including multigrid schemes. All of these are exposed at, and can be controlled from, the command line. Navigating this algorithmic “stack,” via a plethora of run-time options, requires broader and more mathematical user knowledge than the tools of previous generations. So a new book, introducing PETSC for PDEs, is appropriate.

Regarding the required user commitment, the PETSC developers<sup>2</sup> have said that

... developing parallel, nontrivial PDE solvers that deliver high performance is still difficult and requires months (or even years) of concentrated effort. PETSC is a toolkit that can ease these difficulties and reduce the development time, but it is not a black-box PDE solver, nor a silver bullet.

The homepage for PETSC, with download and installation instructions, is

[www.mcs.anl.gov/petsc](http://www.mcs.anl.gov/petsc)

## What does the reader bring?

To make sense of this book, some of the theory and practice of PDEs, and of basic numerical mathematics, must be familiar. In particular, many ideas from numerical linear algebra appear, often with only a brief introduction. For example, the definitions of vector and matrix norms, not to mention the LU decomposition, are reviewed in the briefest way. A textbook like Trefethen and Bau [143] on numerical linear algebra is, therefore, close to an actual prerequisite.

Presumably all applied mathematicians are wanting when it comes to deep understanding of nonlinear PDE problems—certainly true of the author—but good introductions exist to the 20th-century theory of PDEs [51, 111], to methods for solving them exactly [70], and for extracting meaning from them via asymptotic expansions [119]. On the numerical side there are multiple discretization approaches, including finite difference (FD) [70, 84, 104, 115], finite element (FE) [19, 49, 87], and finite volume (FV) [103] methods. All of these approaches are used here, with an emphasis on FD and FE. Spectral methods [142] are outside of our scope.<sup>3</sup> It helps if the reader has been exposed to at least one discretization paradigm already, but we always recall the basics at first use. The author keeps references [7, 49, 51, 66, 84, 115, 143, 144] within reach.

## What programming skills are assumed?

We assume that the reader has some skills, namely a bit more than those needed for an introductory numerical methods course. The programming in Chapters 1–12 is standard C [90, 94], using a modest language subset and only simple data structures. Types used here include the basic integer and floating-point types, plus pointers, functions, arrays, and `structs`, along with the opaque-pointer PETSC objects on which we focus. The examples use ISO C99 features and were tested with the GNU C compiler ([gcc.gnu.org](http://gcc.gnu.org)).

Some experience writing and compiling C programs is thus assumed. Concepts of linking, header files, and passing arguments by value and pointer should all be familiar. (The C language has no “pass-by-reference” syntax, but that is often the intent when passing a pointer by value.) Doing the exercises and/or modifying the examples will inevitably expose some language subtleties, but no more than would appear in the exercises in a first university course in computer programming using a C-like language.

---

<sup>2</sup><http://wgropp.cs.illinois.edu/bib/talks/tdata/2001/PETSc.pdf>

<sup>3</sup>The spectral method idea is mentioned in Chapter 13 when we express the limitations of our use of “optimal” for describing the complexity of solution algorithms.

Both compile-time and run-time errors will inevitably arise, so the ability to run codes within a debugger is strongly recommended. Furthermore, use of `valgrind` ([valgrind.org](http://valgrind.org)) will help to resolve certain memory or pointer-related run-time errors.

The Python codes in Chapters 13 and 14 use mathematical concepts and run-time options explored in the first twelve chapters. Therefore, addressing the C example codes first is recommended even if Python/Firedrake is the preferred environment.

We assume a Bash shell ([www.gnu.org/software/bash/bash.html](http://www.gnu.org/software/bash/bash.html)), or one that interprets Bash syntax. In the text a Bash command line starts with “`$.`” The syntax is supposed to be transparent, but here is a reminder of loop syntax:

```
$ for X in useful fun cool; do echo "PETSc is $X"; done
PETSc is useful
PETSc is fun
PETSc is cool
```

We also use the Unix utilities `grep` and `less` in straightforward ways; see Chapter 1.

## Scope and coverage

Of the many possible uses of PETSc, this book assumes you want to solve linear and nonlinear PDEs. With this in mind, our scope is to first discretize PDE problems and then apply the following class of numerical solvers to the resulting systems of algebraic equations:

$$\underbrace{\text{preconditioned}}_{\text{Chs. 2 \& 6}} \quad \underbrace{\text{Newton} - \text{Krylov}}_{\text{Ch. 4}} \quad \underbrace{\text{methods}}_{\text{Ch. 2}}$$

These terms may be completely new, or buzzwords with a whiff of meaning, but this book aims to convert them into powerful working vocabulary.

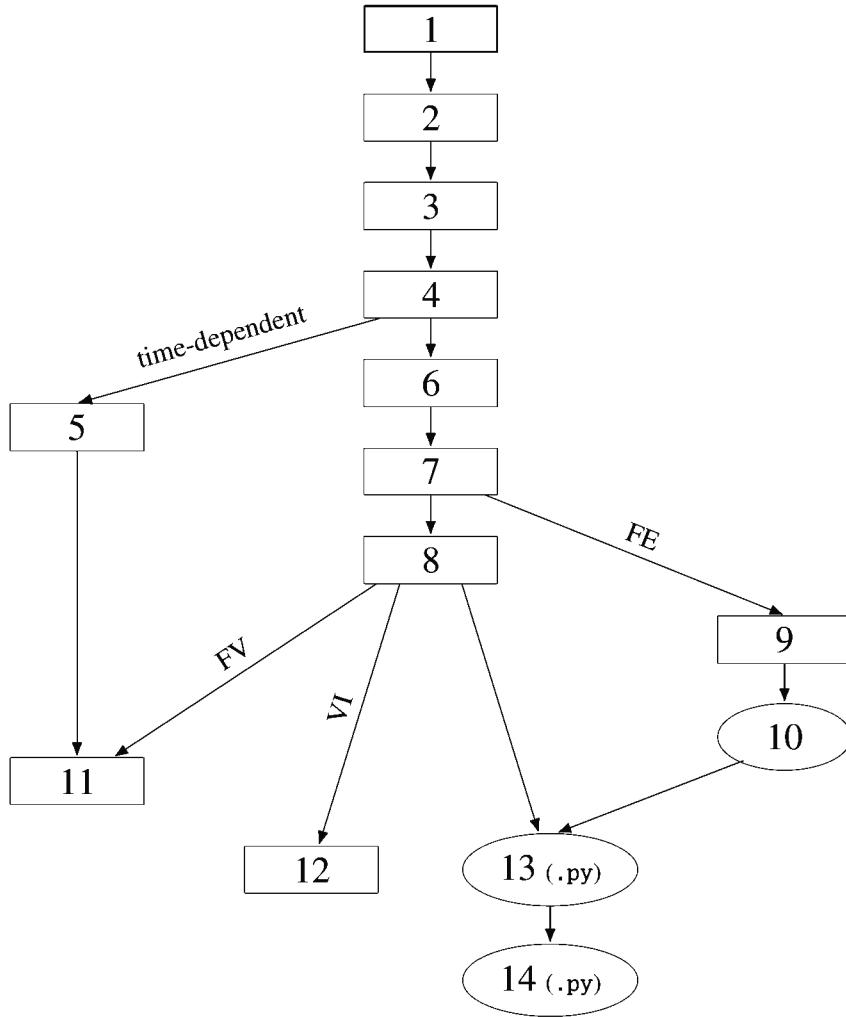
For the reader who has heard these words before, this scope is not confining. For example, suppose we wish to solve a linear, elliptic PDE by discretizing it into a linear system and then solving that system by Gaussian elimination (i.e., LU decomposition). Because the first step of a Newton iteration solves a linear system, and because LU decomposition is one of our many preconditioners, we include this solution method into the “preconditioned Newton–Krylov” class.

On the other hand, when solving linear systems of  $m$  equations in  $m$  variables, Trefethen and Bau [143] identify *optimal complexity*, namely  $O(m)$  work, as the modern goal:

The ideal iterative method in linear algebra reduces the number of steps from  $m$  to  $O(1)$  and the work per step from  $O(m^2)$  to  $O(m)$ , reducing the total work from  $O(m^3)$  to  $O(m)$ .

This goal, as applied to discretized nonlinear PDE problems, is our central thread. For us,  $m$  is the number of grid points or, more accurately, the number of discrete degrees of freedom. Starting with Chapter 6, each Chapter includes a demonstration, via a measurement of work per degree of freedom, of an optimal solution method, almost always by using a *multigrid* [21, 144] preconditioner. Indeed, PETSc is the first comprehensive answer to the following critical numerical library design question: How can a user compose an optimal or near-optimal multigrid solver at the command line for a large variety of nonlinear PDE problems?

On the other hand, the goal is not really to cover the *theory* of preconditioned Newton–Krylov methods. Rather it is to illustrate their use in the PETSc environment, with special attention given to the composition (construction) of solution algorithms through the use of run-time options. (PETSc could stand for “Portable, Extensible Toolkit for *Solver composition*” [109].) Furthermore, while theory is needed to understand the key goals of optimal solver complexity



**Figure 1.** Chapters 1–8 form the core of the book: C codes implement finite difference (FD) methods to solve linear and nonlinear PDEs. Time-dependent ODE and PDE problems in the left branch (Chapter 5) lead to a finite volume (FV) approach to advection problems (Chapter 11). The right branch uses the finite element (FE) method (Chapters 9 and 10), including Python codes using the Firedrake library [126] (Chapters 13 and 14). Chapter 12 considers a variational inequality (VI). Chapters 10, 13, and 14 apply unstructured meshes (the ellipses), but the others use structured grids.

(Chapter 7) and parallel scalability (Chapter 8), demonstrating these properties in practice is as important as presenting theory for why they should occur.

Nonlinear elliptic PDEs form the largest class of example problems, but substantial material is outside of this category, including time-dependent parabolic PDEs (Chapter 5), time-dependent advection (Chapter 11), free-boundary problems via inequality constraints (Chapter 12), and the Stokes equations (Chapter 14). However, the reader interested in hyperbolic (wave-type) PDEs will be disappointed. Purely hyperbolic time-dependent problems are treated lightly because explicit time-stepping schemes [84, 103] are often effective and, in a sense, already have optimal complexity (Chapter 11), without needing advanced equation solvers.

A chapter progression is shown in Figure 1. Note that only the FD method is used through Chapter 8, but then the FE method is introduced in Chapters 9, 10, and 13. Applications of the FV method are confined to Chapter 11. The majority of examples in the book use 1D, 2D, and 3D structured grids, but Chapters 10, 13, and 14 apply 2D unstructured meshes.

## Level of abstraction and philosophy

The first 12 chapters of this book choose a certain software level of abstraction, namely direct use of PETSc’s C API. The example codes involve relatively low-level loops and arithmetic on individual (scalar) array entries when implementing a PDE discretization. The following criticism is therefore valid [126]:

...[key among the drawbacks of coding at a lower level] is a premature loss of mathematical abstraction: the symbolic structure of differential equations, function spaces, and integrals is replaced by loops over arrays of coefficient values. . . . This has many deleterious effects. First, choices are committed to far too early: deciding to change discretization or the equations to be solved requires the implementation to be recoded. Second, the [application] developer must deal with the mixture of equations, discretization, and implementation all at once.

Though Chapters 3–12 choose a concrete PDE discretization and code it at a low level in C, on the other hand they apply a modern, sophisticated, and flexible view of solver composition at the command line. Additional software layers can remedy the above “deleterious effects.” To illustrate, in Chapters 13 and 14 we apply the `petsc4py` [39] and Firedrake [126] Python libraries to solve the Poisson and Stokes equations, thereby escaping the low level of abstraction while retaining command-line control of solver composition.

Our C codes can therefore be easily defended. Understanding the PETSc API, and the relationship between PETSc options and the underlying mathematical ideas, is necessary for effective application of higher-level tools like Firedrake. Higher-level tools necessarily expose solver choices to the user because nonlinear PDE problems are not easy. Indeed, PETSc’s solver-composition design is motivated by the well-known fact that there is no single Krylov solver that is best for all nonsymmetric linear problems [68, 117], and the situation is no simpler when the problems are nonlinear [29]. Effective use of any higher-level software built on PETSc depends on facility with a large collection of mathematical concepts about solvers, many of which are covered in this book.

It has been said that a computational environment is unlikely to lead to real progress unless the software environment is convenient enough to encourage playing around.<sup>4</sup> Despite using C, a compiled language, PETSc is designed to encourage such experimentation through extensive command-line support for solver composition. A next layer, such as Firedrake with Python, can encourage problem-level play.

One might describe our example codes by these slogans for the PDE solution process:

1. Every problem is nonlinear,
2. every numerical computation is parallel,
3. solver choices should never be hard-wired, and
4. performance comes from careful preconditioner choice.

Starting with Chapter 4, each example follows this philosophy, including both well-documented discretizations *and* guidance on solver options. The examples collectively use a large number of PETSc run-time options, but nonetheless they only scratch the surface of solver possibilities.

Finally, in this vein, if this book (and PETSc itself) has a single guiding philosophy, then it might be this fundamental law of computer science<sup>5</sup>: *As machines become more powerful, the efficiency of algorithms grows more important, not less.*

---

<sup>4</sup>This is one “maxim” at <https://people.maths.ox.ac.uk/trefethen/maxims.html>.

<sup>5</sup>This is another maxim: <https://people.maths.ox.ac.uk/trefethen/maxims.html>.

## Some things this book does not do

This book does NOT

- really help you install PETSc;
- replace the PDF *PETSc Users Manual* [10], or its HTML manual pages, for understanding the PETSc API;
- even mention most of the packages PETSc links to;
- use the PETSc Fortran API;
- consider spatial dimensions beyond three; or
- prove anything of consequence.

Regarding the last point, theorems are stated precisely when appropriate, and some exercises ask for proofs. In computational examples, however, evidence for convergence at desired rates is presented, while *a priori* proofs of convergence are de-emphasized or left to the references.

The above is an incomplete list of what the book does not do. Indeed, the modern applications of PETSc go far beyond the PDE problems which are our focus. Solving a PDE is often merely the “inner loop,” surrounded by an “outer loop” to do data assimilation or inverse modeling, for example. That is, the entire PDE solution process may be a step inside an optimization or sensitivity-analysis algorithm. This book omits outer-loop-related, and other, advanced features of PETSc, including but not limited to

- TAO, the Toolkit for Advanced Optimization, which includes parallel-scalable, constrained and unconstrained optimization algorithms, including for PDE-constrained problems;
- the TSAdjoint methods of the TS type (Chapter 5), which support the sensitivity analysis of differential equation solutions with respect to initial conditions and/or parameters;
- the DMForest, DMStag, DMSwarm, and DMNetwork types of DM; these are mesh, particle, and network management systems which go beyond the structured grids (DMDA; Chapter 3) and unstructured meshes (DMPlex; Chapter 13) which we do cover; and
- support for accelerator and GPU architectures [48].

Furthermore, in Chapters 1–12 we do not use any optional external packages, which PETSc can easily download and install, such as the Hypre multigrid and preconditioner library [77], MUMPS and SuperLU sparse direct solvers libraries, or the HDF5 data model and file format. See

[www.mcs.anl.gov/petsc/miscellaneous/external.html](http://www.mcs.anl.gov/petsc/miscellaneous/external.html)

Besides Firedrake, there are many FE method toolkits and libraries which use PETSc for their underlying parallel numerics, including FEniCS, libMesh, MOOSE, DEAL.II, and others. We do not use these, and we also do not solve PDE eigenvalue problems using the Scalable Library for Eigenvalue Problem computations (SLEPc) [78].

It goes without saying that exploration of all these tools is recommended.

## Future releases

PETSc, an open source project with an active and global developer community, will not stand still. Thus the author commits to maintaining this book's example programs for future PETSc (and Firedrake) releases. See the "Releases" feature at the repository

[github.com/bueler/p4pdes](https://github.com/bueler/p4pdes)

Versioned releases of p4pdes will follow the PETSc version number, which is 3.13 at the time of publication.

Furthermore it is unlikely that everything in this book, or in the example programs, is correct. The reader is encouraged to report corrections and issues through the "Issues" or "Pull requests" features at the p4pdes repository. An up-to-date list of errata is in the ERRATA.md file at the same repository.

## Acknowledgments

This book integrates the influence of many generous colleagues and students, but I take full responsibility for all of the inadequacies in the result.

Jed Brown showed me PETSC for the first time, and I have been following him ever since. Constantine Khrulev is my guru for simulations. Barry Smith said good words about this project at a stage where that was highly speculative, and has supported it since. As the book was nearing completion I received essential feedback from Mark Adams, Matt Knepley, Lois Curfman McInnes, and Richard Mills. Dave May made sure I stayed on the straight and narrow, but fluidly. Important feedback and support came from Celso Alvizuri, Satish Balay, David Ham, Max Heldmann, David Keyes, David Maxwell, Lawrence Mitchell, Will Mitchell, and Patrick Sanan. Thanks to the energetic PETSC developer community for creating an endlessly interesting world of mathematics and computation.

Elizabeth Greenspan of SIAM kept the faith, which helped me do the same, and the everyday professionalism of Matthew Bernard, Mellisa Pascale, Lisa Briggeman, Kathleen LeBlanc, and Cheryl Hufnagle made my first pass through book production easy.

I owe my love of books to my parents, Lois and Bill Bueler. Jill, Thomas, and Vera are everything.

## Chapter 1

# Getting started with PETSc

### A code that does almost nothing, but in parallel

The purpose of the PETSc library is to help solve scientific and engineering problems, especially on multiprocessor computers. As PETSc is built on top of the Message Passing Interface (MPI) library [72], some of the flavor of MPI comes through. We start with an introductory PETSc code which calls MPI directly for some basic tasks.

Our program `e.c`, shown in its entirety in Code 1.1, approximates Euler's constant

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} \approx 2.718281828459045 \quad (1.1)$$

by summing the series in a distributed manner, one term on each process. Thus we get a better estimate of  $e$  when we run on more MPI processes. This silly use of PETSc and MPI is an easy-to-understand parallel computation.

```
#include <petsc.h>

int main(int argc, char **argv) {
  PetscErrorCode ierr;
  PetscMPIInt rank;
  PetscInt i;
  PetscReal localval, globalsum;

  PetscInitialize(&argc,&argv,NULL,
    "Compute e in parallel with PETSc.\n\n");
  ierr = MPI_Comm_rank(PETSC_COMM_WORLD,&rank); CHKERRQ(ierr);

  // compute 1/n! where n = (rank of process) + 1
  localval = 1.0;
  for (i = 2; i < rank+1; i++)
    localval /= i;

  // sum the contributions over all processes
  ierr = MPI_Allreduce(&localval,&globalsum,1,MPIU_REAL,MPIU_SUM,
    PETSC_COMM_WORLD); CHKERRQ(ierr);

  // output estimate of e and report on work from each process
  ierr = PetscPrintf(PETSC_COMM_WORLD,
    "e is about %17.15f\n",globalsum); CHKERRQ(ierr);
```

```

ierr = PetscPrintf(PETSC_COMM_SELF,
    "rank %d did %d flops\n", rank, (rank > 0) ? rank-1 : 0);
CHKERRQ(ierr);
return PetscFinalize();
}

```

**Code 1.1.** *c/ch1/e.c. Compute e with PETSC.*

What does `e.c` actually do? First it declares some variables, then it does a computation on each process, and then it communicates the results between processes to get the final result, namely an estimate of  $e$ . As with most C programs, it starts by including headers, but only `petsc.h` is needed because that includes MPI too. Like any C program, `e.c` has a `main()` function which takes inputs `argc` and `argv` from the command line; the former is an `int` holding the argument count, and the latter is an array of strings. In all of our codes we will simply pass these arguments into `PetscInitialize()` so that PETSc can extract the command-line options; such options are used extensively in this book.

The `main()` function has an `int` output which is zero if the program succeeds. If the last call to `PetscFinalize()` succeeds then `main()` will report success. The error-catching macros “`CHKERRQ(ierr)`,” addressed momentarily, can also return nonzeros from `main()`.

The data types used in the code may catch the reader’s eye. Types `PetscErrorCode`, `PetscMPIInt`, `PetscInt`, and `PetscReal` are all aliases of basic arithmetic types in C. For example, `PetscInt=int` and `PetscReal=double` in most cases. However, PETSc’s configuration system (below) allows these aliases to be assigned to other types such as 64-bit integers for `PetscInt` and quadruple-precision (128-bit) floating-point numbers for `PetscReal`. The former capability is essential for problems with billions of degrees of freedom because the maximum `int` is  $2^{31}$ , approximately two billion. While these extended capabilities are not pursued in this book, good style requires using PETSc types for portability. In fact, PETSc can also use complex scalars, i.e., the type `PetscScalar` can be either real or complex, but here our PDE problems are always real.

## Compiling and running the first code

Before you can compile and run `e.c`, PETSc must be installed. If it is not already installed on your machine, follow the instructions at

[www.mcs.anl.gov/petsc/documentation/installation.html](http://www.mcs.anl.gov/petsc/documentation/installation.html)

to download, configure, and compile PETSc. You will do steps in the PETSc directory that look like the following:

```

$ export PETSC_DIR=/home/bueler/petsc          # the PETSc directory path
$ export PETSC_ARCH=linux-c-dbg                 # you choose the name
$ ./configure --download-mpich --with-debugging=1 # typical basic configuration
$ make all
$ make check

```

You may also want to run `make streams` at this point to get a sense of the memory bandwidth of your machine; see Chapter 8 for more information.

The specific configure options may be different in your environment, but the ones above are recommended in the absence of other advice. They download the MPICH ([www.mpich.org](http://www.mpich.org)) package, which is at least appropriate on any machine lacking an existing MPI installation. We also configure a version of PETSc with debugging symbols to enhance the trace-back mechanism (below) for catching run-time errors.

The PETSC developer team can help with failed installation attempts. Send configure and installation questions, including the `configure.log` and `make.log` files generated during the failed attempt, to `petsc-maint@mcs.anl.gov`.

Once PETSc is correctly installed, the environment variables `PETSC_DIR` and `PETSC_ARCH` will point to a valid PETSc installation. These variables need to be set every time you compile PETSc programs, so it makes sense to set them in your `.bashrc` file, for example. Note that the MPI command `mpiexec`, used below, must be from the same MPI installation as the one used in configuring PETSc. Type “`which mpiexec`” to find which one you are running. You may need to modify your `PATH` environment variable to ensure the correct `mpiexec`, for example:

```
| $ export PATH=$PETSC_DIR/$PETSC_ARCH/bin:$PATH
```

Now we can compile and run the first example `e.c` from this book. Use Git (`git-scm.com`) to get the example codes:

```
| $ git clone https://github.com/bueler/p4pdes.git
```

This will generate directory `p4pdes/`, and codes from subdirectory `p4pdes/c/chN/` are described in Chapter  $N$  of this book.

```
| $ cd p4pdes/c/ch1/
| $ make e
```

This uses the `makefile` in the `c/ch1/` subdirectory; an extract is shown in Code 1.2. All `makefiles` in this book have this PETSc-recommended form; we also add to `CFLAGS` to enforce compliance with the C99 standard [90, 94].

```
include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules
CFLAGS += -pedantic -std=c99

e: e.o
    -${CLINKER} -o e e.o ${PETSC_LIB}
    ${RM} e.o
```

**Code 1.2.** *c/ch1/makefile*. PETSc C codes are built using a `makefile` like this.

Run the code in serial like this:

```
| $ ./e
| e is about 1.0000000000000000
| rank 0 did 1 flops
```

Here “rank” refers to the index of an MPI process, and “flops” stands for *floating-point operations*, a count of arithmetic operations. Of course, the value 1.0 is a very poor estimate of  $e$ , but the code does better with more MPI processes:

```
| $ mpiexec -n 5 ./e
| e is about 2.708333333333333
| rank 0 did 0 flops
| rank 4 did 3 flops
| rank 2 did 1 flops
| rank 3 did 2 flops
| rank 1 did 0 flops
```

With  $N = 20$  processes, and thus  $N = 20$  terms in series (1.1), we get a good estimate, accurate to all digits shown:

```
| $ mpiexec -n 20 ./e
|   e is about 2.718281828459045
|   ...
|
```

Note we are using a 64-bit floating-point representation of real numbers, with about 16 decimal digits of accuracy [143]; in this default PETSc configuration, the type `PetscReal` is the same as `double` in C.

Based on the above run, perhaps the reader is worried that examples will only work on a cluster with at least 20 physical processors. In fact, these runs work just fine on the author's 4-core laptop. MPI processes are created as needed using an old feature of operating systems, namely multitasking. However, achieving actual speedup from parallelism is another matter entirely; it requires multiple physical processors and adequate memory bandwidth (see Chapter 8).

Any MPI computation is a collection of processes, each with a separate address space, communicating by messages. `PETSC_COMM_WORLD` in Code 1.1, the MPI communicator [72] for our computation, consists of the set of ranks generated by using “`mpiexec`” at the command line. Each MPI process in the communicator computes a term  $1/n!$  in the sum, where  $n$  is the value returned by `MPI_Comm_rank()`. When run with  $N$  processes, the call to `MPI_Allreduce()` computes the  $N$ th partial sum of the series in (1.1) and then sends the result back to each process.

Because PETSc generally avoids duplicating MPI functionality, such direct uses of the MPI API, including calls to `MPI_Comm_rank()` and `MPI_Allreduce()` here, are occasionally needed. On the other hand, the vast majority of library calls in the rest of this book are to PETSc functions.

In Code 1.1 we print the computed estimate of  $e$  only once, but each process also prints its rank and the flops it used. The formatted print command `PetscPrintf()`, similar to `fprintf()` from the C standard library, is thus used twice: first with MPI communicator `PETSC_COMM_WORLD` and then with `PETSC_COMM_SELF`. The first only prints on rank zero, so only one line of output is produced, but the second print is individual to each rank. However, the `PETSC_COMM_SELF` outputs appear in an unpredictable order because printing occurs as soon as that rank reaches the print command. Orderly output of these lines would use `PetscSynchronizedPrintf()`, requiring additional interprocess communication.

## Every PETSc program

The `main()` function should always start and end with the commands `PetscInitialize()` and `PetscFinalize()`:

```
PetscInitialize(&argc,&args,NULL,help);
< everything else >
return PetscFinalize();
```

`PetscInitialize()` calls `MPI_Init()` to create the `PETSC_COMM_WORLD` communicator, and it sets up the options database and debugging/logging facilities.

In the last argument to `PetscInitialize()` we supply a help string. To see it at run time, plus PETSc version information, do

```
| $ ./program -help intro
```

A potentially very long list of allowed PETSc options—thus often piped into a pager like `less`—is generated by the simpler option `-help`:

```
| $ ./program -help | less
```

By piping the output of `-help` through a regular-expression search like `grep`, every PETSc program has a run-time option documentation system that is both lightweight and surprisingly effective. For example, to see options related to logging performance, do

```
| $ ./program -help | grep log_
```

See Exercise 1.1 for an example of how to add a new option to our example code `e.c`.

Unfortunately, with respect to aesthetics, all PETSc codes in C also have error-checking clutter. We are stuck with ugly lines like

```
ierr = PetscCommand(...); CHKERRQ(ierr);
```

This is because PETSc functions return a code, for error checking, of type `PetscErrorCode`, with value zero if the function was successful. Here it is passed into the `CHKERRQ()` macro which does nothing when it is zero. If `ierr` is nonzero, then the program is stopped with a “trace-back”: a list of function calls, line numbers, and source file names, thereby identifying where the error occurred. Trace-back, which is a first line of defense when debugging run-time errors, is most effective if PETSc is configured with debugging symbols (thus `--with-debugging=1` above). In later chapters we will remove the error-checking clutter from the *displayed* version of codes, but it is always present.

The PDF *PETSc Users Manual* [10] is the best introduction to the PETSc API itself, and searching in the HTML manual pages is also essential when resolving errors in a new code. See

[www.mcs.anl.gov/petsc/documentation/index.html](http://www.mcs.anl.gov/petsc/documentation/index.html)

## Exploring the example codes

This book is based on a substantial collection of C codes<sup>6</sup> in the directory `p4pdes/c/`, a good place from which to explore. For example, to find usages of `PetscPrintf()` in the examples one might run one of these commands:

```
| $ git grep -i petscprintf
| $ grep -iR petscprintf
```

The option `-i` ignores case when matching, so no (human) memory is wasted on the capitalization patterns in PETSc API names. The `git grep` command searches the whole repository tree, but only in Git repositories, while `grep` needs `-R` to search recursively.

This book’s testing system for example codes is also based at `p4pdes/c/`. The following commands clear out old executables (if present) from each chapter subdirectory (i.e., `chN/`), and then compile and run regression tests on all of the example codes:

```
| $ make distclean      # note PETSc takes over the usual "clean" target
| $ make test
```

You can always ask a PETSc code for copious performance information by using `-log_view`. For example,

```
| $ cd ch6/
| $ make fish
| $ mpexec -n 4 ./fish -da_refine 6 -log_view
```

reports information on the performance, including timing, flops, and load balance, from a solution of the Poisson equation (see Chapter 6). However, the recommended way to time a PETSc

---

<sup>6</sup>The Python codes in Chapters 13 and 14 use PETSc through the Firedrake library.

code, whether in serial or parallel, requires a configuration *without* debugging symbols. Each configuration has a different name, the value of PETSC\_ARCH, and you can just add a new one:

```
$ cd $PETSC_DIR
$ export PETSC_ARCH=linux-c-opt
$ ./configure --download-mpich --with-debugging=0
$ make all
```

Then rebuild your code and grep for the timing information from -log\_view:

```
$ cd ~/p4pdes/c/ch6/
$ make fish
$ mpixexec -n 4 ./fish -da_refine 6 -log_view | grep "Time (sec):"
Time (sec):      4.517e-02      1.005      4.507e-02
```

The first number is the maximum time for the processes, the second is the load imbalance (max/min), and the third is the average time over all processes. Performance measurements and parallel scaling are the topics of Chapter 8.

## Exercises

- 1.1. Modify `e.c` to create a new code `expx.c` which, for a real number  $x$ , approximates  $e^x$  by its  $N$ -term Maclaurin series. Read  $x$  at the command line using the PETSC options mechanism:

```
PetscOptionsBegin(PETSC_COMM_WORLD,"","options for expx","");
PetscOptionsReal("-x","input to exp(x) function",NULL,x,&x,NULL);
PetscOptionsEnd();
```

Find  $N$  so that the run

```
| $ mpixexec -n N ./expx -x -20.0
```

gives an approximation of  $e^{-20}$  accurate to six digits. You will also need to add lines to the `makefile`, to compile `expx.c`.

*Most readers will now want to move on to linear algebra and solving PDEs in Chapters 2 and 3. Readers interested in understanding more about MPI might try the following exercises.*

- 1.2. Program `e.c` does redundant work and a terrible job of load balancing: the rank  $n$  process does  $n - 1$  flops. Modify `e.c` to a new code `balanced.c` which balances the load almost perfectly, namely one divide operation on each process. Use blocking send and receive operations (`MPI_Send()`, `MPI_Recv()`) to pass the result of the last factorial to the next rank. (Now the code does a great deal of unnecessary communication and waiting!)
- 1.3. One does not need PETSC to do a sum in parallel. Convert `e.c` into an MPI-only code `cpi.c` which approximates  $\pi$  via the simplest rectangular approximation of the integral

$$\int_0^1 \frac{1}{1+x^2} dx = \arctan(1) = \frac{\pi}{4}.$$

Explicitly include `mpi.h`, use `MPI_Bcast()` and `MPI_Reduce()` to put the sum on the rank 0 process, and print the result there. Now you have written the introductory example in the MPI book [72].

## Chapter 2

# Finite-dimensional linear systems

At the core of numerical methods for elliptic PDEs is the solution of finite-dimensional linear systems. As a grid is refined, these systems become increasingly accurate representations of the PDE, and typically one wants to solve the largest linear systems which the computer can handle. Therefore, methods from linear algebra that have the potential to scale to large sizes—so large that the vector solution of the linear system must be distributed across compute nodes to even fit in memory—form the core technology in PETSc.

This chapter starts by reviewing the basic ideas of numerical linear algebra. We move fast and finish the review by page 23; compare textbook-length treatments in [64, 66, 143]. Our goal, besides recalling definitions, is to identify the constraints which make solving large linear systems challenging. Main threads of numerical linear algebra are recalled, including iteration to reduce the residual, the connection between Krylov spaces and polynomial approximation, and preconditioning to improve the convergence of Krylov-type iterations. We then introduce PETSc types for vectors, matrices, and linear systems, including sparse storage of matrices and parallel distribution of vectors and matrices. Then we use examples, including linear systems extracted from later chapters, to compare run-time options for Krylov iterations and preconditioners. At that point we are well prepared to solve the Poisson equation in Chapter 3.

## Vectors, matrices, and norms

Vectors, matrices, and norms are assumed to be familiar to the reader, but we set notation and recall the basics.

The unqualified word “vector” will mean a real<sup>7</sup> column vector of finite length. The space of real vectors of length  $N$  is written  $\mathbb{R}^N$ . (In later chapters we also regard functions, defined on domains in  $\mathbb{R}^d$ , as vectors in infinite-dimensional function spaces, but these are generally called “functions” or “solutions” and there will be no confusion.) We will use bold for vectors and square brackets for indexed entries; thus  $\mathbf{v} \in \mathbb{R}^N$  has entries  $v[j] \in \mathbb{R}$ , a convention which frees the subscript to be used for sequences of vectors, e.g., in iteration. Note also that we break tradition, but facilitate C programming, by numbering vector entries starting from zero.

A *matrix* is a linear map between finite-dimensional vector spaces. In polite company it is more accurate to say that a matrix is the representation of a linear map from one vector space to another, given a choice of basis in each space. We write  $\mathbb{R}^{M \times N}$  for the space of all  $M$ -row and  $N$ -column matrices that map from  $\mathbb{R}^N$  to  $\mathbb{R}^M$ , and we denote the entries of  $A \in \mathbb{R}^{M \times N}$  as  $a_{ij}$ . (This use of subscripts causes no conflicts because we do not consider sequences of matrices other than powers of matrices.)

---

<sup>7</sup>PETSc can be configured to handle complex numbers, but we use only reals.

Given  $A \in \mathbb{R}^{M \times N}$ , there are two related matrices which map in the other direction. If the matrix is square and invertible, then the *inverse*  $A^{-1} \in \mathbb{R}^{N \times N}$  is the unique matrix satisfying  $A^{-1}A = AA^{-1} = I$ . Even if  $A$  is nonsquare, one may always construct the *transpose*  $A^\top \in \mathbb{R}^{N \times M}$  with entries  $(A^\top)_{ij} = a_{ji}$ .

A *vector norm* is a function  $\|\cdot\| : \mathbb{R}^N \rightarrow [0, \infty)$  which assigns a finite length to a vector. Axioms including the triangle inequality must be satisfied [143]. For  $1 \leq p < \infty$  the following formula defines a norm:

$$\|\mathbf{v}\|_p = \left( \sum_{i=0}^{N-1} |v[i]|^p \right)^{1/p}. \quad (2.1)$$

The  $p \rightarrow \infty$  limit of (2.1), the maximum absolute value, also defines a norm:

$$\|\mathbf{v}\|_\infty = \max_{i=0, \dots, N-1} |v[i]|. \quad (2.2)$$

All of these vector norms can be computed with  $O(N)$  work, but the  $p = 1, 2, \infty$  cases are especially common in practical use. As properly introduced later in this chapter, PETSc computes these vector norms by applying `VecNorm()` to a `Vec` object.

Choosing vector norms on the input and output vector spaces *induces* a *matrix norm* on  $A \in \mathbb{R}^{M \times N}$  as follows:

$$\|A\| = \sup_{\mathbf{v} \neq 0} \frac{\|A\mathbf{v}\|}{\|\mathbf{v}\|} = \sup_{\|\mathbf{v}\|=1} \|A\mathbf{v}\|. \quad (2.3)$$

It follows from (2.3) that

$$\|A\mathbf{v}\| \leq \|A\| \|\mathbf{v}\|. \quad (2.4)$$

Induced matrix norms are generally expensive to compute because (2.3) is an optimization problem whose solution may take much more work than the  $O(MN)$  operations needed to inspect all the entries of  $A$ . However, the matrix norms  $\|A\|_1$  and  $\|A\|_\infty$ , induced by the  $p = 1$  and  $p = \infty$  vector norms, respectively, are computable by  $O(MN)$  formulas [143]. The *Frobenius* matrix norm, which satisfies (2.4) even though it is not induced, is also widely used. It can be computed by the  $O(MN)$  formula  $\|A\|_{\text{Fro}}^2 = \sum_{i,j} a_{ij}^2$ . Note PETSc can compute  $\|A\|_1$ ,  $\|A\|_\infty$ , or  $\|A\|_{\text{Fro}}$  by applying `MatNorm()` to a `Mat` object.

We will use vector norms routinely in both linear and nonlinear solver algorithms, and in measuring numerical errors in solving PDEs, but what good are matrix norms? In this book we use them mostly when measuring the *conditioning* of linear systems [143]. By definition, if  $\|\cdot\|$  denotes a matrix norm and  $A \in \mathbb{R}^{N \times N}$  is a square and invertible matrix, then

$$\kappa(A) = \|A\| \|A^{-1}\| \quad (2.5)$$

is the *condition number* of  $A$ . By definition  $\kappa(A) = +\infty$  if  $A$  is not invertible. The condition number measures how sensitive the (exact) solution of a linear system  $A\mathbf{v} = \mathbf{b}$  is to perturbations of either  $A$  or  $\mathbf{b}$  [143].

## Linear systems, and some facts of (numerical) life

Suppose  $\mathbf{b} \in \mathbb{R}^N$  is a column vector and  $A \in \mathbb{R}^{N \times N}$  is a square matrix. When  $A$  is invertible, the linear system

$$A\mathbf{u} = \mathbf{b} \quad (2.6)$$

has a unique solution

$$\mathbf{u} = A^{-1}\mathbf{b} \in \mathbb{R}^N. \quad (2.7)$$

This is simple in theory.

It is not so simple in practice, however, to solve large linear systems on a computer. Here are three facts to keep in mind while numerically solving linear systems:

Fact 1. On a digital computer there are unavoidable limitations to numerical accuracy. *If real numbers are represented in floating point with machine precision  $\epsilon$ , then the solution of  $A\mathbf{u} = \mathbf{b}$  can only be computed within an error  $O(\kappa(A)\epsilon)$ , where  $\kappa(A)$  is the condition number.*

Fact 2. Dense direct linear algebra has a high cost. *For a dense matrix  $A \in \mathbb{R}^{N \times N}$ , computation of the solution to  $A\mathbf{u} = \mathbf{b}$ , by a direct method such as Gauss elimination (LU decomposition), whether forming  $A^{-1}$  or not, requires  $O(N^3)$  operations.<sup>8</sup>*

Fact 3. You should not assemble the inverse. *The matrices  $A$  arising from discretized PDEs are often sparse, but their inverses  $A^{-1}$  are usually dense and may not even fit in memory.*

Numerical facts of life like these appear in several places in this book, and they are collected in the appendix.

Fact 1 is about conditioning and not about methods. There are matrices  $A$  and  $\tilde{A}$  that are the same<sup>9</sup> to within machine precision  $\epsilon$  but for which the corresponding exact solutions to (2.6) differ by an amount  $\kappa(A)\epsilon$ . Rounding errors effectively perturb  $A$  by  $O(\epsilon)$  with each operation, and thus  $O(\kappa(A)\epsilon)$ -size errors will appear in the solution of the linear system. *Backward stable* methods [143], for example the QR direct method, can be shown to achieve this minimal level of inaccuracy.

The machine precision of the C double type, the default 64-bit representation of real numbers, is  $\epsilon = 2.2 \times 10^{-16}$ . (The double type is aliased to PetscReal in most PETSC configurations.) Thus by Fact 1 a linear system with  $\kappa(A) \approx 10^{10}$ , for example, can only be solved to five or six decimal digits of precision. Though such a matrix would be regarded as “poorly conditioned,” it is possible for a condition number of that size to arise from discretizing a PDE. While it would be wise to inquire whether a different discretization generates a matrix with a smaller condition number, expectations about solution accuracy should, at least, be informed by estimates of condition numbers.

By Fact 2, a generic linear system with  $N = 10^6$  equations requires  $10^{18}$  or so operations for a solution by Gauss elimination. This is impractical; even modern supercomputers take a while to do a quintillion operations. Furthermore, direct methods generally solve linear systems via the construction of matrix factors which are closely related to the inverse of  $A$ . These matrix factors are often dense and may take up as much memory as the inverse (see Fact 3). Thus naive application of direct methods—treating direct solvers for (2.6) as black boxes—will break down.

Though direct methods may be impractical for generic, dense  $N = 10^6$  systems, we will successfully solve PDE-generated linear systems of that size and larger on a single processor in a few seconds, and in  $O(N)$  operations, using iterative methods. A key idea is that discretized PDEs generate linear systems with exploitable structure, especially *sparsity*, which means that there are only a few nonzero entries in each matrix row and column.

A simple example illustrating Fact 3 is the  $N$ -dimensional tridiagonal matrix  $A$  assembled later in this section that uses  $24N$  bytes of memory. (There are three nonzeros per row, and each allocated real entry uses 8 bytes in double precision.) The inverse  $A^{-1}$ , however, is dense. For this matrix in dimension  $N = 10^7$ , storing the inverse in the obvious way would occupy  $8N^2 = 8 \times 10^{14}$  bytes (800 terabytes), but we will solve (2.6) for a matrix of this size using less than a gigabyte ( $10^9$  bytes) of memory in the entire solution process.<sup>10</sup>

---

<sup>8</sup>See Lecture 32 in [143] for some caveats with respect to “ $O(N^3)$ .”

<sup>9</sup>That is,  $\|A - \tilde{A}\|/\|A\| < \epsilon$ .

<sup>10</sup>See `tri.c` later in this chapter: `./tri -tri_m 10000000 -ksp_type cg`

## Residuals and errors

By definition, the *residual* of a given vector  $\mathbf{v}$  in linear system (2.6) is the vector

$$\mathbf{r} = \mathbf{b} - A\mathbf{v}. \quad (2.8)$$

Typically  $\mathbf{v}$  is an approximation of the solution  $\mathbf{u}$  of (2.6). If  $\mathbf{r} = 0$ , then  $\mathbf{v}$  is a solution, but if  $\mathbf{r}$  is nonzero, then its entries and norm encode information about how far  $\mathbf{v}$  is from being a solution. Evaluating the residual for a known vector  $\mathbf{v}$  requires applying  $A$  to it, an  $O(N^2)$  operation at worst. Because most discretization schemes for PDEs generate matrices  $A$  that are sparse, with the nonzeros per row typically small and independent of  $N$ , the residual can often be computed in  $O(N)$  operations.

The idea that the residual norm  $\|\mathbf{r}\| = \|\mathbf{b} - A\mathbf{v}\|$  measures the “wrongness” of  $\mathbf{v}$  as an approximation to the solution of (2.6) is reasonable, but it needs exploration. We prefer to measure the *error*

$$\mathbf{e} = \mathbf{v} - \mathbf{u}, \quad (2.9)$$

and/or its norm  $\|\mathbf{e}\|$ , but exact knowledge of  $\mathbf{e}$  is equivalent to exact knowledge of the solution  $\mathbf{u}$  itself, so only bounds on  $\|\mathbf{e}\|$  can be expected.

The residual and error are, however, related by a version of system (2.6):

$$A\mathbf{e} = -\mathbf{r}. \quad (2.10)$$

By (2.4), this equation immediately gives a bound  $\|\mathbf{e}\| \leq \|A^{-1}\| \|\mathbf{r}\|$ . We can say more by noting that error norms are most meaningful if they are relative. (For instance, “ $\|\mathbf{e}\| \leq 10^{-6}$ ” does not tell us that  $\mathbf{v}$  is an accurate solution to the system  $A\mathbf{u} = \mathbf{b}$  in a case where  $\|A^{-1}\| = 1$  and  $\|\mathbf{b}\| = 10^{-7}$ . We would know that  $\|\mathbf{u}\| \leq 10^{-7}$  anyway, and thus any  $\mathbf{v}$  with small norm will yield  $\|\mathbf{e}\| \leq 10^{-6}$ .) Furthermore, relative error norms relate to the conditioning of  $A$ . In fact, the relative quantity we can compute, namely  $\|\mathbf{r}\|/\|\mathbf{b}\|$ , and the ratio we want to control, namely  $\|\mathbf{e}\|/\|\mathbf{u}\|$ , are related as follows (Exercise 2.2):

$$\frac{1}{\kappa(A)} \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|} \leq \frac{\|\mathbf{e}\|}{\|\mathbf{u}\|} \leq \kappa(A) \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}. \quad (2.11)$$

Looking ahead to the process of solving PDEs, we caution the reader about the phrase “numerical error.” The error  $\mathbf{e} = \mathbf{v} - \mathbf{u}$  of an approximation  $\mathbf{v}$  to a linear system (2.6) is *not* the error we care about if we want the total numerical error be small in some PDE solution process. That total error is the difference between the computed finite-dimensional approximation  $\mathbf{v}$  and the usually unknown (exact) continuum solution of the PDE problem, which lives in some function space, not in  $\mathbb{R}^N$ .

**Fact 4.** The linear system solver error is not the numerical error of the PDE method. *Though solving a linear system  $A\mathbf{u} = \mathbf{b}$  may be part of your method, making  $\|\mathbf{e}\| = \|\mathbf{v} - \mathbf{u}\|$  small for this system does not control the discretization error or the total numerical error.*

## Richardson iteration

It is an old idea that equations can be solved by starting with a guess and improving it. For example, Newton had this idea; see Chapter 4. For linear system (2.6), the *Richardson iteration* [128] is the simplest such idea. At each iteration it adds a multiple  $\alpha > 0$  of the last residual:

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \alpha(\mathbf{b} - A\mathbf{u}_k). \quad (2.12)$$

If, for example, the matrix-vector product  $A\mathbf{v}$  costs  $O(N)$  operations and if significantly fewer than  $O(N^2)$  steps are needed to make  $\mathbf{u}_k$  an adequate approximation of the solution  $\mathbf{u}$  of (2.6) then such an iteration could improve on  $O(N^3)$  Gauss elimination.

Is it reasonable to just add the residual as a correction? Can we even hope that (2.12) converges? In the context of optimization, at least, the answer is “yes” to both questions. If  $A$  is symmetric and positive-definite (SPD) [143] then system (2.6) describes the minimum of the quadratic function

$$g(\mathbf{v}) = \frac{1}{2}\mathbf{v}^\top A\mathbf{v} - \mathbf{b}^\top \mathbf{v}. \quad (2.13)$$

Note the gradient is  $\nabla g(\mathbf{v}) = A\mathbf{v} - \mathbf{b}$ , so linear system (2.6) is just  $\nabla g(\mathbf{u}) = 0$ , the critical-point condition from calculus. A natural idea in optimization is to start with a guess and go directly downhill (*steepest descent*) toward the minimum. Steepest descent takes steps down the gradient,

$$\mathbf{u}_{k+1} = \mathbf{u}_k - \alpha \nabla g(\mathbf{u}_k),$$

where  $\alpha$  can be chosen by (approximately) solving one-dimensional optimization problems; see “line search” in Chapter 4. Observing that  $\nabla g(\mathbf{v}) = -\mathbf{r} = -(\mathbf{b} - A\mathbf{v})$ , we see that Richardson iteration (2.12) is just the steepest-descent algorithm in this context. Steepest descent will converge, though often slowly, when appropriate step sizes  $\alpha$  are used [118]. However, for some  $\alpha$  the Richardson iteration may not converge even when  $A$  is SPD.

**Example 2.1.** Consider the SPD linear system

$$A\mathbf{u} = \begin{bmatrix} 10 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 8 \\ 1 \end{bmatrix} = \mathbf{b} \quad (2.14)$$

with solution  $\mathbf{u} = [1 \ 2]^\top$ . If we start with estimate  $\mathbf{u}_0 = [0 \ 0]^\top$  then the  $\alpha = 1$  Richardson iteration (2.12) gives a sequence

$$\mathbf{u}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \mathbf{u}_1 = \begin{bmatrix} 8 \\ 1 \end{bmatrix}, \quad \mathbf{u}_2 = \begin{bmatrix} -63 \\ 9 \end{bmatrix}, \quad \mathbf{u}_3 = \begin{bmatrix} 584 \\ -62 \end{bmatrix}, \dots \quad (2.15)$$

This sequence is not heading toward the solution  $\mathbf{u}$ .

Such examples motivate analysis. If we rewrite (2.12) as

$$\mathbf{u}_{k+1} = (I - \alpha A)\mathbf{u}_k + \alpha \mathbf{b}, \quad (2.16)$$

then it is easy to believe that the “size” of the matrix  $I - \alpha A$  determines whether  $\lim_{k \rightarrow \infty} \mathbf{u}_k$  exists. Specifically,  $\mathbf{u}_k = (I - \alpha A)^k \mathbf{u}_0$  if  $\mathbf{b} = 0$ , so convergence relates to the behavior of powers of the matrix  $I - \alpha A$ . We pursue this idea via well-known definitions as follows.

A complex number  $\lambda \in \mathbb{C}$  is an *eigenvalue* of a square matrix  $B \in \mathbb{R}^{N \times N}$  if there is a nonzero vector  $\mathbf{v} \in \mathbb{C}^N$ , an *eigenvector*, so that  $B\mathbf{v} = \lambda\mathbf{v}$ . (Recall that  $\lambda$  and  $\mathbf{v}$  may be complex even if  $B$  is real.) The set of all eigenvalues of  $B$  is the *spectrum*  $\sigma(B)$  of  $B$ . The *spectral radius*  $\rho(B)$  is the maximum magnitude of the eigenvalues of  $B$ . The matrix  $B^\top B$  is symmetric and positive-semidefinite [64] for any real  $B$ , and thus it has nonnegative eigenvalues. The *singular values* of  $B$  are the square roots of the eigenvalues of  $B^\top B$ . They are defined geometrically as the lengths of semi-axes of the hyperellipsoid in  $\mathbb{R}^N$  that results from applying  $B$  to all vectors in the unit sphere [143].

Matrix properties described in terms of eigenvalues or singular values are generically called “spectral properties.” For example, if  $\|\cdot\|_2$  denotes the (induced) matrix 2-norm then  $\|B\|_2$  is the largest singular value of  $B$ , and  $\|B^{-1}\|_2$  is the inverse of the smallest singular value of  $B$ , so

these norms are spectral properties of  $B$ . The 2-norm condition number  $\kappa(B) = \|B\|_2\|B^{-1}\|_2$  is thus also a spectral property. (It is the eccentricity of the above-mentioned hyperellipsoid.)

Returning to (2.12), we can write all such iterations in the form

$$\mathbf{u}_{k+1} = B\mathbf{u}_k + \mathbf{c} \quad (2.17)$$

for  $B \in \mathbb{R}^{N \times N}$  and  $\mathbf{c} \in \mathbb{R}^N$ . Exercise 2.3 asks you to show that (2.17) will converge for all  $\mathbf{c}$  and all initial iterates  $\mathbf{u}_0$  if and only if the spectrum of  $B$  is inside the unit circle:

(2.17) converges for all data if and only if  $\rho(B) < 1$ . (2.18)

Richardson iteration (2.12) therefore converges for all  $\mathbf{u}_0$  if and only if  $\rho(I - \alpha A) < 1$ . Because one can also show that  $\rho(B) \leq \|B\|$  in any induced matrix norm, (2.12) converges if  $\|I - \alpha A\| < 1$ .

## A first look at preconditioning

There are many linear systems which are equivalent to (2.6). In particular, if  $M \in \mathbb{R}^{N \times N}$  is an invertible matrix then the systems

$$(M^{-1}A)\mathbf{u} = M^{-1}\mathbf{b} \quad (2.19)$$

and

$$(AM^{-1})(M\mathbf{u}) = \mathbf{b} \quad (2.20)$$

each has the same solution(s) as (2.6). These systems are the *left- and right-preconditioned* versions of  $A\mathbf{u} = \mathbf{b}$ , respectively.

The *preconditioned matrices*  $M^{-1}A$  and  $AM^{-1}$  generally have different spectral properties—different eigenvalues, condition numbers, and so on—from  $A$ . In fact, though matrices  $M^{-1}A$  and  $AM^{-1}$  are similar to each other, they are generally not similar to  $A$ . (Recall that matrices  $B$  and  $C$  are *similar* to each other if there is an invertible matrix  $S$  so that  $C = S^{-1}BS$ . Furthermore, recall that similar matrices have the same eigenvalues.) See Exercise 2.4.

Algorithms may take advantage of superior spectral properties of the preconditioned matrices, should they possess such, to approximately solve  $A\mathbf{u} = \mathbf{b}$  in fewer iterations. However, a preconditioning approach to solving (2.6) is only effective if

- (i)  $M^{-1}$  is easy to apply, and
- (ii) the action of  $M^{-1}$  approximately inverts  $A$ .

The meaning of (i) is that solving  $M\mathbf{v} = \mathbf{c}$  must require far fewer operations than solving (2.6). As we will see soon, the initial meaning of (ii) is that the preconditioned matrix spectrum is sufficiently close to the spectrum of the identity matrix so that the iteration using  $M^{-1}A$  or  $AM^{-1}$  converges rapidly.

Evidently, these specifications for effective preconditioners  $M$  are only loosely defined. Choosing a good preconditioner is not easy, but PETSc is designed to facilitate rapid exploration of a space of powerful preconditioners.

In practice one does not form  $M^{-1}$  when using (2.19) or (2.20). Instead the system  $M\mathbf{v} = \mathbf{c}$  is solved. For example, when applying  $M^{-1}A$  to a vector  $\mathbf{v}$  one does the matrix-vector product with  $A$  first, and then one solves  $M\mathbf{w} = A\mathbf{v}$  to find  $\mathbf{w} = M^{-1}A\mathbf{v}$ . These steps are reversed when applying  $AM^{-1}$  to  $\mathbf{v}$ .

The next example, where the diagonal of  $A$  is used as  $M$ , shows how such *Jacobi* preconditioning can make the Richardson iteration converge.

**Example 2.2. (Continued from 2.1.)** Suppose we extract the diagonal of  $A$  from (2.14):

$$M = \begin{bmatrix} 10 & 0 \\ 0 & 1 \end{bmatrix}. \quad (2.21)$$

Being diagonal,  $M$  is easy to invert and apply. With  $\mathbf{u}_0 = [0 \ 0]^\top$ , the left-preconditioned  $\alpha = 1$  Richardson iteration using  $M$ , namely  $\mathbf{u}_{k+1} = \mathbf{u}_k + M^{-1}\mathbf{b} - M^{-1}A\mathbf{u}_k$ , generates a sequence

$$\mathbf{u}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \mathbf{u}_1 = \begin{bmatrix} 0.8 \\ 1.0 \end{bmatrix}, \quad \mathbf{u}_2 = \begin{bmatrix} 0.9 \\ 1.8 \end{bmatrix}, \quad \mathbf{u}_3 = \begin{bmatrix} 0.98 \\ 1.90 \end{bmatrix}, \dots \quad (2.22)$$

This sequence converges to the solution  $\mathbf{u} = [1 \ 2]^\top$ . A justification of convergence is via (2.16) and (2.18); compare  $\rho(I - A) = 9.1$  and  $\rho(I - M^{-1}A) = 0.32$ .

The “extreme” preconditioning cases are  $M = I$  and  $M = A$ . Using  $M = I$ , equations (2.19) and (2.20) revert to the original system  $A\mathbf{u} = \mathbf{b}$ , while using  $M = A$  in (2.19) gives  $\mathbf{u} = A^{-1}\mathbf{b}$ , so “preconditioning” means solving the original system. In fact, in PETSc a direct solution method like Gaussian elimination is regarded as a preconditioner application.

The most powerful preconditioning methods fall between these extremes, providing rapid but imperfect inversion of  $A$ . Stating this general idea in terms of the entries of  $A$  is impossible, but for the Richardson iteration the idea that  $M^{-1}$  should approximately invert  $A$  can be interpreted as choosing  $M$  so that the spectrum of  $M^{-1}A$  is clustered around  $1 \in \mathbb{C}$ , the single eigenvalue of the identity matrix. Because of the properties of Krylov methods (next section), the practical preconditioning goal becomes that the spectrum of  $M^{-1}A$  is clustered into a few small areas of the complex plane away from the origin.

Even if both  $A$  and  $M$  are symmetric,  $M^{-1}A$  and  $AM^{-1}$  may not be symmetric. This would be a significant issue because certain iterations (next section) only work for symmetric matrices, but it turns out not to be an impediment. Exercises 2.7 and 2.9 explore *symmetric preconditioning* in the case where  $M$  is SPD. Iterations which expect a symmetric  $A$  can be implemented for SPD  $M$  without losing the benefits of symmetry; see the next section.

Preconditioner possibilities are listed in Table 2.1. Each one has a PETSc name used with the option `-pc_type` at run time. The table attempts to describe the  $M$  used in (2.19) or (2.20), but these are slogans not formulas.

Factorization types `cholesky`, `lu`, and `svd`, which have  $M = A$  in the table, are “direct methods” [64]. They would exactly solve a linear system in exact arithmetic. They may be used without an iteration, i.e., with `-ksp_type preonly`; see the next section. The `cholesky` and `lu` types allow variable reorderings (row and column permutations); the option `-pc_factor_mat_ordering_type` is illustrated in Chapter 3. However, the global exchanges of information needed by these methods imply that these PETSc-native implementations are not available in parallel.<sup>11</sup>

Incomplete versions of the LU and Cholesky direct methods, namely the `icc` and `ilu` types, generate factors which approximate the true factors but which maintain sparsity of the original matrix. Thus these methods, which are now traditional choices for preconditioners, reduce fill-in in sparse matrix calculations [66, 112]. Their effectiveness as preconditioners depends in nontrivial ways on the spectrum and sparsity of the original matrix and on the ordering of the variables.

Now, for any invertible  $M$  and scalar  $\alpha > 0$ , we call

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \alpha M^{-1}(\mathbf{b} - A\mathbf{u}_k) \quad (2.23)$$

<sup>11</sup>Parallel sparse direct solver are available in PETSc via external packages including MUMPS, `mumps.enseeht.fr`, and SuperLU [105]. See [10] and [www.mcs.anl.gov/petsc/miscellaneous/external.html](http://www.mcs.anl.gov/petsc/miscellaneous/external.html).

**Table 2.1.** Commonly used preconditioning methods from PETSc. For `jacobi` and `sor` note  $A = D + L + U$ , where  $D$  is diagonal and  $L, U$  are triangular. See Chapters 6 and 7 for more on `asm`, `fieldsplit`, `mg` and Chapter 10 for `gamg`.

	PC type	Operator $M$
additive Schwarz	<code>asm</code>	overlapping diagonal blocks of $A$
block Jacobi	<code>bjacobi</code>	diagonal blocks of $A$
Cholesky decomposition	<code>cholesky</code>	$A = L^\top L$ (SPD)
vector-component split	<code>fieldsplit</code>	component blocks of $A$
algebraic multigrid	<code>gamg</code>	$\approx A$
incomplete Cholesky	<code>icc</code>	$\tilde{L}^\top \tilde{L}$ (SPD)
incomplete LU	<code>ilu</code>	$\tilde{L}\tilde{U}$
Jacobi	<code>jacobi</code>	$D$
Gauss elimination (LU decomposition)	<code>lu</code>	$A = LU$
geometric multigrid	<code>mg</code>	$\approx A$
none	<code>none</code>	$I$
successive over-relaxation (SOR) and Gauss-Seidel	<code>sor</code>	$\omega^{-1}D + L$
singular-value decomposition	<code>svd</code>	$A = U^\top \Sigma V$

simple iteration [66]. That is, simple iteration is Richardson iteration applied to the left-preconditioned system (2.19).

Classical Jacobi (`jacobi`), successive over-relaxation (`sor`), and Gauss-Seidel (also `sor`) iterations are cases of simple iteration which extract parts of  $A$  to give  $M$  [66]; see Exercise 2.1. These well-known methods split  $A$  into diagonal, lower-triangular, and upper-triangular parts:  $A = D + L + U$ .

To understand the literature about these methods one must observe that a so-called *matrix splitting*  $A = M - N$  gives a matrix iteration

$$M\mathbf{u}_{k+1} = N\mathbf{u}_k + \mathbf{b}, \quad (2.24)$$

which is actually equivalent to simple iteration (2.23) when  $\alpha = 1$ . To see this replace  $N = M - A$  on the right side of (2.24) and then multiply both sides by  $M^{-1}$ :

$$\mathbf{u}_{k+1} = \mathbf{u}_k + M^{-1}(\mathbf{b} - A\mathbf{u}_k) \stackrel{A=M-N}{\iff} M\mathbf{u}_{k+1} = N\mathbf{u}_k + \mathbf{b}. \quad (2.25)$$

Looking forward, Chapter 6 describes how the classical matrix-splitting iterations are used as “smoothers” in multigrid methods [26, 144] and in block decompositions. These classical iterations do not, however, yield effective solver methods for PDE problems when used on their own in simple iteration.

The two multigrid preconditioning methods described as “ $\approx A$ ” in the table are critically important for efficient solution of PDE problems. Algebraic (`gamg`) and geometric (`mg`) multigrid

have complicated details which we begin to address in Chapter 6. Geometric multigrid is a focus of Chapter 6, the algebraic form of multigrid is introduced in Chapter 10, and multigrid methods are used extensively in Chapters 7–14.

The block approximations of  $A$  implied by the additive Schwarz method (asm) [134] and block Jacobi (bjacobi) preconditioners are also covered in Chapter 6. Such block structure arises naturally from a parallel decomposition of a PDE problem into subgrids or submeshes.

The `fieldsplit` preconditioning framework [27], first demonstrated in Chapter 7, gets its block decomposition from a vector-valued PDE problem. That is, `fieldsplit` applies to systems of PDEs, the best-known case being Schur decomposition preconditioning of the Stokes equations; this is demonstrated in Chapter 14.

Finally, two preconditioner types are not listed in the table because they do not correspond to an operator  $M$ . The `redundant` (Chapter 7) and `telescope` (Chapter 8) PC types are *metapreconditioners* which determine on which MPI processes the preconditioning action occurs. That is, they determine the parallel layout of the data which defines the preconditioner.

For further information on preconditioning see references [16, 64, 66, 152].

## Krylov space methods

**Iterative methods** are well suited to the linear systems which arise from discretizing PDEs. Such methods approximate the solution by linear combinations of vectors  $\mathbf{v}$ ,  $A\mathbf{v}$ ,  $A^2\mathbf{v}$ , ...,  $A^{k-1}\mathbf{v}$ , where  $\mathbf{v}$  is an initial residual (e.g.,  $\mathbf{v} = \mathbf{b} - A\mathbf{u}_0$  or  $\mathbf{v} = \mathbf{b}$ ). The matrix  $A$  may be the original one in system (2.6) or a preconditioned version. The effectiveness of such *Krylov space methods* in solving a particular linear system depends on the spectral properties of the matrix. Because there is no single Krylov method which is best for all cases, many such methods are supported in PETSc.

The following overview of Krylov methods is brief, and references [66, 131, 143] are recommended for algorithms and convergence theory. By definition, for a square matrix  $A \in \mathbb{R}^{N \times N}$ , a vector  $\mathbf{v} \in \mathbb{R}^N$ , and an integer  $n \geq 0$ , the *Krylov (sub)space* [99] is

$$\mathcal{K}_n(A, \mathbf{v}) = \text{span}\{\mathbf{v}, A\mathbf{v}, A^2\mathbf{v}, \dots, A^{n-1}\mathbf{v}\} \subseteq \mathbb{R}^N. \quad (2.26)$$

This subspace is of dimension at most  $\min\{n, N\}$ , but it can be of lower dimension. (For example, suppose  $\mathbf{v}$  is an eigenvector of  $A$ .)

Suppose  $\mathbf{w} \in \mathcal{K}_n(A, \mathbf{v})$ . Then

$$\mathbf{w} = c_0\mathbf{v} + c_1A\mathbf{v} + c_2A^2\mathbf{v} + \cdots + c_{n-1}A^{n-1}\mathbf{v}$$

for some coefficients  $c_j$  or, equivalently,

$$\mathbf{w} = p(A)\mathbf{v},$$

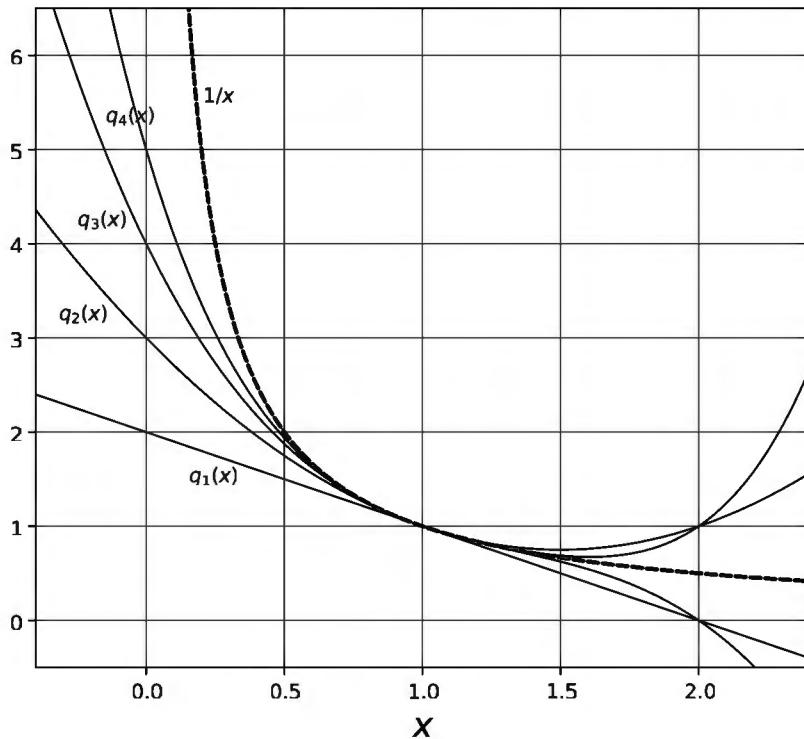
where  $p(x) = c_0 + c_1x + \cdots + c_{n-1}x^{n-1}$  is a polynomial of degree at most  $n - 1$ . This essential observation allows us to use the properties of polynomials to analyze the approximation abilities of vectors in  $\mathcal{K}_n(A, \mathbf{v})$ .

To approximate  $\mathbf{u} = A^{-1}\mathbf{b}$  solving (2.6) we may seek a polynomial  $p_{n-1}$  so that

$$p_{n-1}(A) \approx A^{-1}, \quad (2.27)$$

so  $\mathbf{w} = p_{n-1}(A)\mathbf{b} \in \mathcal{K}_n(A, \mathbf{b})$  approximates  $\mathbf{u} = A^{-1}\mathbf{b}$ . Note  $\mathbf{w}$  is formed using only  $n - 1$  applications of  $A$  to vectors, plus operations to form the linear combination.

One can prove that if a scalar polynomial  $p_{n-1}(z)$  is close to the function  $1/z$  on the finite set of eigenvalues of  $A$ , i.e., on the spectrum  $\sigma(A)$  in the complex plane  $\mathbb{C}$ , then (2.27) holds in



**Figure 2.1.** The Richardson iteration (2.12) approximates  $\mathbf{u} = A^{-1}\mathbf{b}$  by  $\mathbf{u}_k = q_k(A)\mathbf{b}$  for polynomials  $q_k(x)$  which approximate  $1/x$  on the interval  $(0, 2)$ .

a norm sense (Exercise 2.10). Specifically, the size of the error  $\|\mathbf{w} - \mathbf{u}\|$  is determined by the maximum of  $|p_{n-1}(z) - z^{-1}|$  for  $z \in \sigma(A)$  and by the conditioning of the eigenvalue problem for  $A$ . Thus, whether  $p_{n-1}$  is a “good” for approximately inverting  $A$  is a spectral question about  $A$ . While  $p_{n-1}(z) \approx 1/z$  is impossible on the whole of  $\mathbb{C}$ , it can be achieved on finite and/or compact subsets of  $\mathbb{C}$  which exclude zero, and the spectrum  $\sigma(A)$  of an invertible matrix  $A$  is such a set.

The construction of a polynomial  $p_{n-1}$  satisfying (2.27) is thus a question of approximation theory on the finite subset  $\sigma(A) \subset \mathbb{C}$ . While precise knowledge of the spectrum  $\sigma(A)$  is asking too much, and accurately computing  $\sigma(A)$  is as difficult as solving (2.6), the context in which  $A$  was generated might supply bounds on the eigenvalues or condition number of  $A$ . For example,  $A$  might come from discretizing a differential operator with known spectral properties.

Consider the Richardson iteration (2.12) with  $\alpha = 1$ . A straightforward calculation (Exercise 2.11) starting with  $\mathbf{u}_0 = \mathbf{b}$  generates  $\mathbf{u}_k = q_k(A)\mathbf{b}$  where  $q_0(x) = 1$ ,

$$q_{k+1}(x) = 1 + (1-x)q_k(x), \quad (2.28)$$

and one can show that  $q_k(z) \rightarrow 1/z$  for  $z$  in a disk  $D$  of radius one around  $z_0 = 1$ . Figure 2.1 shows these polynomials  $q_k$ . On the other hand, (2.18) says  $\mathbf{u}_k = q_k(A)\mathbf{b}$  converges if  $\rho(I - A) < 1$ , that is, if all eigenvalues of  $A$  are within distance one of  $1 \in \mathbb{C}$ . In summary, the ( $\alpha = 1$ ) Richardson iteration generates iterates  $\mathbf{u}_k$  which are from the Krylov space  $\mathcal{K}_{k+1}(A, \mathbf{b})$ . It converges if the matrices  $q_k(A)$  approximate  $A^{-1}$ , which can only happen if  $\sigma(A)$  is inside  $D$ .

The polynomials generated by the Richardson iteration are not, however, the closest polynomials of the given degree to the function  $1/z$ . Furthermore the Richardson iteration only converges when  $\sigma(A)$  lies inside a fixed disk in the complex plane, a rather rigid requirement on  $A$ . Other Krylov methods are more flexible. We now summarize three well-known methods. Each

generates polynomial approximations which are best in specific senses; the errors or residuals are as small as possible among the vectors in certain Krylov spaces.

To proceed we define a norm and certain affine subspaces. First observe that if  $A$  is SPD then it can be used to define an *A-inner product*  $\mathbf{v}^\top A \mathbf{w}$  and *A-norm*

$$\|\mathbf{v}\|_A = \sqrt{\mathbf{v}^\top A \mathbf{v}}. \quad (2.29)$$

Next let  $\mathcal{P}_n^1$  be the set of all real polynomials  $p$  of degree at most  $n$  such that  $p(0) = 1$ . For any real square matrix  $A \in \mathbb{R}^{N \times N}$ , and a vector  $\mathbf{v} \in \mathbb{R}^N$ , we also define affine Krylov spaces

$$\mathcal{K}_n^1(A, \mathbf{v}) = \mathbf{v} + \text{span}\{A\mathbf{v}, A^2\mathbf{v}, \dots, A^{n-1}\mathbf{v}\} \subset \mathcal{K}_n(A, \mathbf{v}). \quad (2.30)$$

The affine subspaces are closely related to the polynomials:

$$\mathbf{w} \in \mathcal{K}_n^1(A, \mathbf{v}) \iff \text{there exists } p(z) \in \mathcal{P}_{n-1}^1 \text{ so that } \mathbf{w} = p(A)\mathbf{v}.$$

The *conjugate gradient* (CG) algorithm [79] generates iterates which minimize the *A-norm* of the error over all vectors in the Krylov space  $\mathcal{K}_k(A, \mathbf{b})$  [143]. That is, if  $\mathbf{u}_0 = 0$  then the iterate  $\mathbf{u}_k \in \mathcal{K}_k(A, \mathbf{b})$  satisfies

$$\|\mathbf{u}_k - \mathbf{u}\|_A \leq \|\mathbf{w} - \mathbf{u}\|_A \quad \text{for all } \mathbf{w} \in \mathcal{K}_k(A, \mathbf{b}), \quad (2.31)$$

where  $\mathbf{u} = A^{-1}\mathbf{b}$ . Equivalently the errors  $\mathbf{e}_k = \mathbf{u}_k - \mathbf{u}$  solve a minimization problem:

$$\mathbf{e}_k \text{ minimizes } \|\mathbf{v}\|_A \text{ over } \mathbf{v} \in \mathcal{K}_k^1(A, \mathbf{e}_0).$$

Thus the error reduction  $\|\mathbf{e}_k\|_A / \|\mathbf{e}_0\|_A$  achieved by CG arises from the spectral properties of  $A$ , namely

$$\frac{\|\mathbf{e}_k\|_A}{\|\mathbf{e}_0\|_A} \text{ is small if there exists } p \in \mathcal{P}_{k-1}^1 \text{ which is small on } \sigma(A) \subset (0, \infty).$$

The availability of polynomials with small magnitude on the spectrum of  $A$  determines how small is the error in the CG iterates.

If  $A$  is SPD it follows that  $\sigma(A) \subset (0, \infty)$  and that the condition number is the ratio of eigenvalues:  $\kappa_2(A) = \lambda_{\max}/\lambda_{\min}$ . The well-known construction of Chebyshev polynomials, in this case shifted and scaled so as to have small magnitude on the interval  $[\lambda_{\min}, \lambda_{\max}] \subset \mathbb{R}$ , gives an estimate of the error reduction in terms of the condition number of  $A$  [66, 143]. We will return to these ideas, and their consequences for numerical solutions of PDEs, in Chapter 3.

One way of deriving the CG algorithm—and also the manner of its discovery—is to regard it as an improvement of steepest descent for minimizing the objective function  $g(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top A \mathbf{x} - \mathbf{b}^\top \mathbf{x}$ . (Recall that if  $A$  is SPD then the unique minimum is  $\mathbf{u} = A^{-1}\mathbf{b}$ .) If one maintains orthogonality of the search directions in the *A-inner product* (i.e., *conjugacy*) then one gets the CG algorithm [118].

The CG algorithm is a system of simple recurrence formulas (Exercise 2.8). It uses only one application of  $A$  per iteration, and the additional operations require  $O(N)$  work per iteration with a small constant. Only three stored vectors are needed. Thus work-per-iteration and storage are  $O(N)$ , and each application of  $A$  is  $O(N)$ . Also, because each iterate minimizes the *A-norm* of the error on a larger space than the last, the error norms are monotonic nonincreasing. For these reasons, CG is often the preferred Krylov iteration when it is available, namely when the matrix is SPD. Note that Krylov iterations are sometimes effective even outside the cases where they provably work, and so CG may also work for certain “nearly” SPD matrices. While CG is

routinely used in parallel, its implementation includes two inner products per iteration. These global reduction operations require parallel communication (Chapter 8).

If  $A$  is not SPD then there is a different norm to consider. For square matrices  $A$ , recall that  $A^\top A$  is SPD if  $A$  is invertible [143]. We may thus define a norm  $\|\mathbf{v}\|_{A^\top A} = \sqrt{\mathbf{v}^\top A^\top A \mathbf{v}}$ . Using this norm on the error  $\mathbf{e}$  is useful, but it goes by a different name: it is a *residual norm*. In fact, if  $\mathbf{u}$  solves  $A\mathbf{u} = \mathbf{b}$  then

$$\begin{aligned}\|\mathbf{b} - A\mathbf{v}\|_2^2 &= (\mathbf{b} - A\mathbf{v})^\top (\mathbf{b} - A\mathbf{v}) = (A\mathbf{u} - A\mathbf{v})^\top (A\mathbf{u} - A\mathbf{v}) \\ &= (\mathbf{u} - \mathbf{v})^\top A^\top A (\mathbf{u} - \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\|_{A^\top A}^2.\end{aligned}$$

That is, for given  $\mathbf{v}$  the residual  $\mathbf{r} = \mathbf{b} - A\mathbf{v}$  and the error  $\mathbf{e} = \mathbf{v} - \mathbf{u}$  satisfy

$$\|\mathbf{r}\|_2 = \|\mathbf{e}\|_{A^\top A}. \quad (2.32)$$

Changing the goal from minimizing the  $A$ -norm of the error to minimizing the 2-norm of the residual yields effective Krylov methods for many matrices  $A$ . For example, the *minimum residuals* (MINRES) [49, 122] method is designed for indefinite but symmetric matrices, i.e., symmetric matrices with both positive and negative eigenvalues. In exact arithmetic MINRES generates  $\mathbf{u}_k$  which minimize  $\|\mathbf{r}_k\|_2$  over the Krylov space. It exploits the symmetry of  $A$  to retain the short recurrence and  $O(N)$  storage of CG. While the convergence rate is again connected to the spectral properties of  $A$ , convergence bounds are not as simple as for CG. That is, a reduction in norm is tied to the existence of polynomials  $p$  satisfying  $p(0) = 1$  which are small on  $\sigma(A)$ , so the polynomial must be small on a set which straddles the origin [152].

For nonsymmetric matrices, Krylov methods face a choice of either keeping a short recurrence and small memory usage or keeping a strong norm-minimizing property [66]. Both types of methods are implemented in PETSc. In the former short-recurrence category a partial list of KSP types is `cgne`, `bcgs`, `cgs`, `tfqmr`, and `bicg` [66, 10]. About these we say nothing more in this short introduction, but `bcgs` is applied to an advection-diffusion problem in Chapter 11.

The *generalized minimum residuals* (GMRES) [132] method `gmres` is in the latter category. The iterates minimize residual norms, but the memory usage and work-per-iteration are no longer  $O(N)$ . It is a “brute force” approach which stores a representation of the growing Krylov space. While this is slower than the short recurrences in CG and MINRES, the computations at each step of GMRES tend to be quite stable [143]. However, for practical software it would be unacceptable to swamp the computer’s memory with the Krylov space storage from GMRES. In fact, the number of iterations, and thus the size of the storage, depends in a hard-to-predict way on the entries of  $A$ . Instead the method is implemented with *restarts*, namely iterations at which the current Krylov space representation is thrown away and the iteration is reinitialized using the current iterate as  $\mathbf{u}_0$ . This algorithm, often written as “GMRES( $R$ )” where  $R$  is the number of steps between restarts, is critical to making GMRES an effective tool, but the details [131] are beyond our scope.

Though pseudocodes for the CG and preconditioned-CG algorithms appear in the exercises, only the references supply the details of other algorithms. In particular, see [49] for MINRES and [64, 66, 131] for GMRES.

It is worth noting that no Krylov method is “best.” Nachtigal and others [117] showed that for each nonsymmetric method one can design an example matrix  $A$  for which that method is distinctly superior to all others. In a similar spirit, [68] proves that examples can be constructed so that GMRES exhibits any desired convergence rate.

The classical Jacobi and Gauss-Seidel iterations are not pure Krylov space methods because they involve extracting parts of (entries of) the matrix  $A$ . They are, however, simple iterations (2.23) combining a particular Krylov method (Richardson) with left preconditioning (2.19) based

on matrix splitting. Though these classical iterations are slow to converge, their smoothing properties are used in multigrid methods (Chapter 6).

Krylov methods in PETSC live in KSP objects, and the particular method used in a solver can be chosen at the command line using `-ksp_type`. We will demonstrate their usage, and how they are combined with PC preconditioner objects, later in this chapter. The default KSP type in PETSC is GMRES(30), i.e., `-ksp_type gmres -ksp_gmres_restart 30`.

Recall that direct linear algebra methods, such as the LU, Cholesky, or SVD decompositions, represent “extreme preconditioning” with  $M = A$ . On the other hand, if  $M^{-1} = A^{-1}$  then simple iteration (2.23) solves (2.6) in one iteration,

$$\mathbf{u}_1 = \mathbf{u}_0 + A^{-1}(\mathbf{b} - A\mathbf{u}_0) = A^{-1}\mathbf{b},$$

and the same is true of any Krylov method if computed in exact arithmetic. In PETSC these direct methods may be combined, as preconditioners, with any Krylov method, but if only the action of the direct method  $\mathbf{X}$  is desired then the solver combination is `-ksp_type preonly -pc_type X`.

Note that the residual norm after one application of the direct method will generally not be exactly zero because of rounding error. A Krylov iteration may then play an “iterative refinement” role, “mopping up” accumulated rounding error from preconditioner arithmetic. Using a Krylov solver with a convergence test, which is normal PETSC usage, means not having to worry as much about rounding errors because additional iterations are automatically added if rounding error has built up. On the other hand, actual usage of a direct method as the preconditioner is normally limited to small problems because of poor algorithmic scaling with  $N$ , which is  $O(N^3)$  in worst cases.

For any nontrivial linear system, experimentation with Krylov methods is appropriate, and with preconditioners even more so. PETSC is designed to facilitate such experimentation. For example, we will see that the CG method with an incomplete factorization preconditioner seems acceptable for the Poisson problem in Chapter 3. However, multigrid preconditioners (Chapter 6) actually yield a profound improvement. Our recurring focus on preconditioning is another numerical fact of life:

Fact 5. The rate of residual reduction in a Krylov iteration is at the mercy of the spectrum of your preconditioned matrix. *Whatever Krylov iteration you choose, whether norm-minimizing or not, good performance depends on the spectral properties of your matrix. A fast solver for a PDE problem must somehow make the spectral properties of the preconditioned matrix so good that the Krylov choice becomes almost unimportant.*

Table 2.2 is a short list of five Krylov methods which we will use repeatedly in this book. These KSP types are chosen by option `-ksp_type`; also recall  $\mathbf{e}_k = \mathbf{u}_k - \mathbf{u}$  and  $\mathbf{r}_k = \mathbf{b} - A\mathbf{u}_k$ . Note we introduce the Chebyshev iterative method in the next section.

## Chebyshev iteration

The *Chebyshev iterative method* [3, 64] is not norm-minimizing in the same sense as CG, MINRES, and GMRES. Unlike those Krylov methods it requires explicit bounds on the spectrum of  $A$ , thus it is sometimes called a *semi-iterative method* [64]. The iteration itself—not just its analysis—is based on the construction of small polynomials on the spectrum of  $A$ .

The method may be regarded as a simple iteration (2.23) wherein  $M^{-1}$  is a polynomial in  $A$ :

$$\mathbf{u}_{k+1} = \mathbf{u}_k + p(A)(\mathbf{b} - A\mathbf{u}_k). \quad (2.33)$$

It is easy to show (Exercise 2.12) that the errors  $\mathbf{e}_k = \mathbf{u}_k - \mathbf{u}$  then satisfy

$$\mathbf{e}_{k+1} = q(A)\mathbf{e}_k, \quad (2.34)$$

**Table 2.2.** Some common Krylov methods in PETSC for a linear system  $A\mathbf{u} = \mathbf{b}$ .

KSP type	Symmetry	Norm-minimizing?	Good if
cg	SPD	$\mathbf{e}_k$ minimizes $\ \mathbf{v}\ _A$ over $\mathbf{v} \in \mathcal{K}_k^1(A, \mathbf{e}_0)$	exists $p \in \mathcal{P}_n^1$ small on $\sigma(A) \subset (0, \infty)$
chebyshev	SPD	no	known bounds for $\sigma(A)$
gmres	any	$\mathbf{r}_k$ minimizes $\ \mathbf{v}\ _2$ over $\mathbf{v} \in \mathcal{K}_k(A, \mathbf{b})$	exists $p \in \mathcal{P}_n^1$ small on $\sigma(A) \subset \mathbb{C} \setminus \{0\}$
minres	symmetric	$\mathbf{r}_k$ minimizes $\ \mathbf{v}\ _2$ over $\mathbf{v} \in \mathcal{K}_k(A, \mathbf{b})$	exists $p \in \mathcal{P}_n^1$ small on $\sigma(A) \subset \mathbb{R} \setminus \{0\}$
richardson	any	no	$\rho(I - \alpha A) \ll 1$

where  $q(x) = 1 - p(x)x$  is a new polynomial satisfying  $q(0) = 1$ . It follows that for diagonalizable matrices  $A$  we have

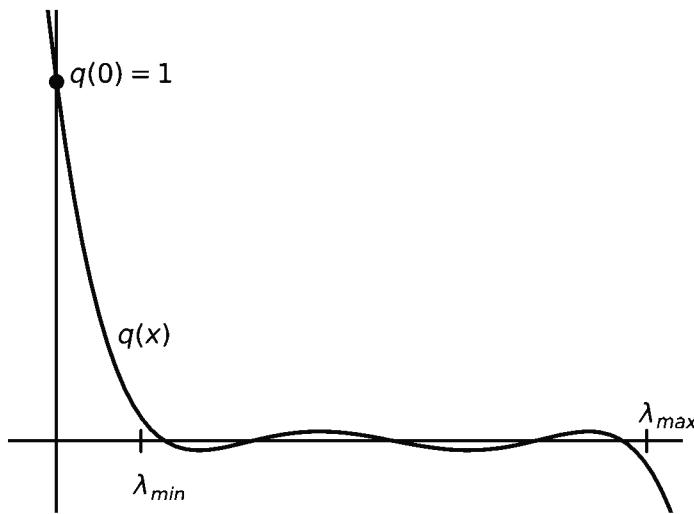
$$\|\mathbf{e}_{k+1}\| \leq \|q(A)\| \|\mathbf{e}_k\| \leq C \max_{\lambda \in \sigma(A)} |q(\lambda)| \|\mathbf{e}_k\|, \quad (2.35)$$

where  $C > 0$  depends only on  $A$ . (The proof of (2.35) follows the same idea as in Exercise 2.10. The constant, which depends on the conditioning of the eigenvalue problem for  $A$ , is one when the matrix is symmetric.) The error norms are thus rapidly reduced when polynomial  $q(x)$  is small on  $\sigma(A)$ .

On the other hand, the *Chebyshev polynomials*  $\{T_n(x)\}$  are a particular orthogonal family in a certain inner product [66, 142]. Their well-known properties include that they are constructed by a short (second-order) recurrence formula and that they grow exponentially large (with  $n$ ) outside of the interval  $[-1, +1]$ , while being bounded by one on that interval (i.e.,  $|T_n(x)| \leq 1$  if  $x \in [-1, +1]$ ). The idea of the *Chebyshev iteration* is that by shifting and scaling Chebyshev polynomials one can build  $q(x)$  with very small magnitudes on  $\sigma(A)$  (Figure 2.2), and thus estimate (2.35) implies rapid reduction in error.

However, implementing this idea requires knowing an interval (or ellipse in  $\mathbb{C}$  [142]) containing  $\sigma(A)$ . For simplicity assume  $A$  is SPD so  $\sigma(A) \subset [\lambda_{\min}, \lambda_{\max}] \subset (0, +\infty)$ . (There exist inexpensive iterations, used in the PETSC implementation of the Chebyshev iteration, which can approximate  $\lambda_{\min}$  and  $\lambda_{\max}$ .) The polynomials  $p(z)$  and  $q(z)$  are built as the iteration proceeds, using the spectral bounds  $\lambda_{\min}$ ,  $\lambda_{\max}$  and a recurrence derived from the recurrence for Chebyshev polynomials [64]. Because of the rapid growth of Chebyshev polynomials outside of  $[-1, +1]$ , Chebyshev iteration is most effective if the interval  $[\lambda_{\min}, \lambda_{\max}]$  is both far from the origin and not too long.

However, we should reveal the reason to include the Chebyshev iteration as an important KSP type (Table 2.2). If the goal of the iteration is changed from reducing the full norm  $\|\mathbf{e}_k\|$  of the  $k$ th-iterate error to instead reducing the high frequencies in  $\mathbf{e}_k$  then the lower bound  $\lambda_{\min}$  can be increased so that the rate of reduction for the high frequencies gets better. An iteration which reduces the high-frequency portion of the error is a *smoother*, and, as addressed in Chapter 6, a multigrid preconditioner uses such a smoother. The primary role of the Chebyshev iteration in our PDE solution methods will be as a component of multigrid methods. The Chebyshev iteration shares the smoother role with the classical Jacobi and Gauss-Seidel iterations, but it is more suitable in parallel [3].



**Figure 2.2.** The Chebyshev iterative method builds polynomials  $q(x)$ , such that  $q(0) = 1$ , which are small on an identified interval  $[\lambda_{\min}, \lambda_{\max}]$ .

Our brief introduction to iterative linear algebra is complete. We turn now to PETSC objects, linear system examples, and example programs.

## PETSc objects

Our first program to solve a linear system will use PETSC's `Vec` and `Mat` data types which represent vectors and matrices, respectively. These data types are, essentially, objects. Although written in C, not an officially object-oriented language like C++, PETSC is a relentlessly object-oriented library. Consider the operations which create and configure a variable of type `Mat` which represents a matrix  $A \in \mathbb{R}^{N \times N}$ :

```
Mat A;
MatCreate(COMM,&A);
MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,N,N);
MatSetFromOptions(A);
MatSetUp(A);
... fill entries of (i.e. assemble) A ...
... solve system with A ...
MatDestroy(&A);
```

These calls, essentially methods of the `Mat` object `A`, control its hidden internal representation. Often such `Mat` methods manipulate the entries of the matrix, but in fact a `Mat` need not even *have* entries. It might instead contain code that applies a linear operator to a vector. Furthermore the data structure inside a `Mat` depends on run-time choices through the call to `MatSetFromOptions()`. For example, option `-mat_type sbaij` chooses a symmetric block sparse representation—often a good idea if the matrix is symmetric.

Once `Mat A` is created and configured, then additional methods become valid. For example, the `MatSetValues()` function sets entries in `A`—see below for examples. Also, setting option `-mat_view` or calling the function `MatView()` will print out the entries of `A`.

The `Mat` type is just one example of the principle that certain operations apply to all PETSC types (“Object” is a placeholder for an actual PETSC type name):

```

Object X;
ObjectCreate(COMM,&X);
... code which sets properties of X ...
ObjectSetFromOptions(X); // allow run-time resetting of properties
... code which uses X ...
ObjectDestroy(&X);

```

Because PETSC types are generally distributed across, and accessible from, multiple MPI processes, the first argument of an `ObjectCreate()` method is an MPI communicator (“COMM”). We will usually use `PETSC_COMM_WORLD`, the communicator generated from all  $P$  processes when we run with “`mpiexec -n P`.`”` Operations like `ObjectCreate()`, `ObjectSetFromOptions()`, and `ObjectDestroy()` are *collective* [72]; they must be called by all processes in COMM.

## Parallel layout of Vec and Mat objects

A `Vec` object stores its entries in parallel across the processes in the MPI communicator used to create it. For example, the create-set-assemble sequence of a `Vec` with four entries might look like this:

```

Vec      x;
PetscInt i[4] = {0, 1, 2, 3};
PetscReal v[4] = {11.0, 7.0, 5.0, 3.0};

VecCreate(PETSC_COMM_WORLD,&x);
VecSetSizes(x,PETSC_DECIDE,4);
VecSetFromOptions(x);
VecSetValues(x,4,i,v,INSERT_VALUES);
VecAssemblyBegin(x);
VecAssemblyEnd(x);

```

The four entries of `x` are set by the call to `VecSetValues()`, putting values from array `v` at the indices given by `i`.

One is allowed to think of a PETSC `Vec` as a one-dimensional C array with its contents split across the processes in the MPI communicator. For example, if the above code is run sequentially as

```
| $ ./mycode
```

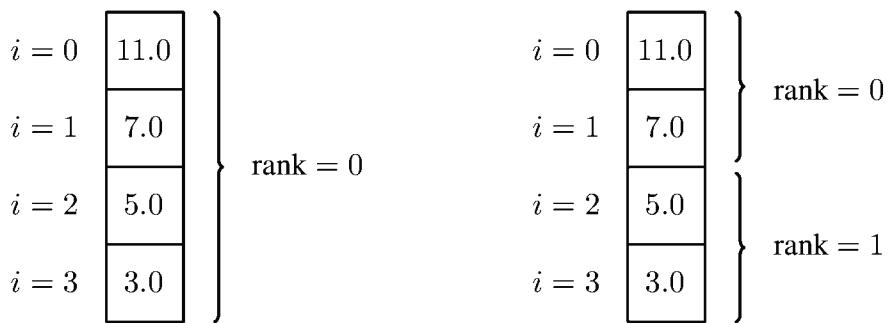
then, at the end of the above sequence, the storage of `x` looks like Figure 2.3 (left). However, if run as

```
| $ mpiexec -n 2 ./mycode
```

then the layout looks like Figure 2.3 (right). The argument `PETSC_DECIDE` in `VecSetSizes()` allows PETSC to split the entries as evenly as possible between the processes.

`Setting values in a Vec may require communication between processes because entries which are to be stored on one process could be set by another process. Such communication can occur between the VecAssemblyBegin() and VecAssemblyEnd() commands.` However, in the above code each process has a copy of all data and simply ignores data in rows it does not own.

PETSC `Mat` objects generally require additional choices regarding storage and parallel distribution. A common storage format is *compressed sparse row storage*, the `MATAIJ` type. In this format only the specifically allocated entries are stored (sparse). These nonzero entries are stored contiguously in memory using an additional index array (compressed). (Allocated entries



**Figure 2.3.** A sequential *Vec* layout (left) and a parallel layout (right) on two processes.

are often referred to as “nonzeros” in sparse representations even though they may be zeros!) In parallel a range of rows is owned by each process (type `MATMPIAIJ`).

Because `Mat` objects are linear operators, their purpose is to multiply `Vecs`. The result `Vec` from a `Mat`-`Vec` product is a linear combination of the columns of the `Mat`. For a `Mat` of type `MATMPIAIJ` the rows are usually distributed the same way as the entries of the intended *output* (i.e., column) `Vec`. Specifically, this is the outcome when `PETSC_DECIDE` is used in setting both the `Vec` and `Mat` local sizes (see below for a `Mat` case) and when the output `Vec` global size is the same as the number of rows of the `Mat`. (As PETSc computes the `Mat`-`Vec` product, it communicates (“scatters” using MPI calls) the input `Vec` to the processes. After the scatter the `Mat`-`Vec` product is a local operation, requiring no further communication.)

## Assemble and view a Mat

Assembling a `Mat` one row at a time is a common usage. In PDE-type applications a matrix row generally represents the discrete version of the PDE at some location in the part of the grid or mesh owned by the processor. (We present much more on this in later chapters.) In such applications there are usually only a few nonzero entries per row.

However, one doesn’t need to know how a `Mat` is stored in order to assemble it. For an example of assembling a small `Mat`, consider the  $4 \times 4$  matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 2 & 1 & -2 & -3 \\ -1 & 1 & 1 & 0 \\ 0 & 1 & 1 & -1 \end{bmatrix}.$$

The following code stores  $A$  in a `Mat`, one row at a time:

```

Mat      A;
PetscInt i, j[4] = {0, 1, 2, 3};
PetscReal aA[4][4] = {{1.0, 2.0, 3.0, 0.0},
                      {2.0, 1.0, -2.0, -3.0},
                      {-1.0, 1.0, 1.0, 0.0},
                      {0.0, 1.0, 1.0, -1.0}};

MatCreate(PETSC_COMM_WORLD,&A);
MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,4,4);
MatSetFromOptions(A);
MatSetUp(A);
for (i=0; i<4; i++) {

```

```

        MatSetValues(A,1,&i,4,j,aA[i],INSERT_VALUES);
    }
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);

```

The function `MatSetValues()` can set multiple entries as long as they form a *block*, namely a product of lists of row indices and column indices. In the above code a row is a  $1 \times 4$  block. The “`1,&i`” arguments set one row with index  $i$  while the “`4, j`” arguments indicate the integer array  $j$  with four column indices. Note that “`aA[i]`” is a pointer to the  $i$ th row of  $A$ .

The approach above treats the matrix as dense—every entry is allocated including those with value 0.0. However, in the usual usage for a sparse matrix one inserts only the nonzero entries. As a demonstration we assemble the same matrix by breaking it up into three blocks of nonzeros: a  $3 \times 3$  block, a  $1 \times 3$  block, and a single entry. The `for` loop is gone, along with any elegance:

```

PetscInt   i1[3] = {0, 1, 2},
           j1[3] = {0, 1, 2},
           i2 = 3,
           j2[3] = {1, 2, 3},
           i3 = 1,
           j3 = 3;
PetscReal  aA1[9] = { 1.0,  2.0,  3.0,
                      2.0,  1.0, -2.0,
                     -1.0,  1.0,  1.0 },
           aA2[3] = { 1.0,  1.0, -1.0 },
           aA3 = -3.0;
...
MatSetValues(A,3,i1,3,j1,aA1,INSERT_VALUES);
MatSetValues(A,1,&i2,3,j2,aA2,INSERT_VALUES);
MatSetValue(A,i3,j3,aA3,INSERT_VALUES);
...

```

Once a `Mat` is assembled its entries can be viewed at run time in various formats. For instance, the `Mat` was assembled using the second method above in a code called `sparsemat.c` (not shown). After building it, we see the following outputs in serial:

```

$ cd c/ch2/
$ make sparsemat
...
$ ./sparsemat -mat_view
Mat Object: 1 MPI processes
  type: seqaij
row 0: (0, 1.) (1, 2.) (2, 3.)
row 1: (0, 2.) (1, 1.) (2, -2.) (3, -3.)
row 2: (0, -1.) (1, 1.) (2, 1.)
row 3: (1, 1.) (2, 1.) (3, -1.)
$ ./sparsemat -mat_view ::ascii_dense
Mat Object: 1 MPI processes
  type: seqaij
  1.00000e+00  2.00000e+00  3.00000e+00  0.00000e+00
  2.00000e+00  1.00000e+00  -2.00000e+00  -3.00000e+00
 -1.00000e+00  1.00000e+00  1.00000e+00  0.00000e+00
  0.00000e+00  1.00000e+00  1.00000e+00  -1.00000e+00

```

The first view shows the sparse storage format, with values as pairs with column index and value. The second view is “dense”: all values are shown, whether they occupy memory or not. Note

	$j = 0$	$1$	$2$	$3$	
$i = 0$	1.0	2.0	3.0		rank = 0
$i = 1$	2.0	1.0	-2.0	-3.0	
$i = 2$	-1.0	1.0	1.0		rank = 1
$i = 3$		1.0	1.0	-1.0	

**Figure 2.4.** A parallel Mat layout, on two processes, of a  $4 \times 4$  matrix with 13 nonzero entries.

that the output from option `-mat_view` occurs at completion of the `MatAssemblyBegin()` and `MatAssemblyEnd()` calls. In a one-process run, by default the Mat is in serial compressed sparse row storage, namely the `MATSEQAIJ` type, and thus “type: seqaij” is reported as above. If the run is parallel then `-mat_view` reports “type: mpiaij” corresponding to `MATMPIAIJ`. For two processes the layout is as shown in Figure 2.4.

Here are some additional possibilities for viewing a Mat:

- (i) show the sparsity pattern graphically using the X window system

```
-mat_view draw -draw_pause 1
```

- (ii) save the matrix to file `mat.dat` in PETSC’s binary format

```
-mat_view binary:mat.dat
```

- (iii) save to text file `mat.m` in a MATLAB text format

```
-mat_view ascii:mat.m:ascii_matlab
```

The file format specification is

```
-mat_view TYPE:FILENAME:FORMAT
```

One may omit TYPE when ascii is desired and FILENAME when destination `stdout` is desired.

## A small linear system

We now know how to create, fill, assemble, view, and destroy Vec and Mat objects. Code 2.1 shows `vecmatksp.c` which does these steps for a small linear system:

$$\begin{bmatrix} 1 & 2 & 3 & 0 \\ 2 & 1 & -2 & -3 \\ -1 & 1 & 1 & 0 \\ 0 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 1 \\ 1 \\ 3 \end{bmatrix}. \quad (2.36)$$

In this code a KSP (Krylov space method) object solves the linear system, but the particular solution algorithm can be chosen at run time. Note that the KSP also has the expected create-set-use-destroy sequence.

```

static char help[] = "Solve a 4x4 linear system using KSP.\n";

#include <petsc.h>

int main(int argc,char **args) {
    Vec      x, b;
    Mat     A;
    KSP     ksp;
    PetscInt i, j[4] = {0, 1, 2, 3};           // j = column index
    PetscReal ab[4] = {7.0, 1.0, 1.0, 3.0},       // vector entries
              aA[4][4] = {{ 1.0,  2.0,  3.0,  0.0}, // matrix entries
                            { 2.0,  1.0, -2.0, -3.0},
                            {-1.0,  1.0,  1.0,  0.0},
                            { 0.0,  1.0,  1.0, -1.0}};

    PetscInitialize(&argc,&args,NULL,help);

    VecCreate(PETSC_COMM_WORLD,&b);
    VecSetSizes(b,PETSC_DECIDE,4);
    VecSetFromOptions(b);
    VecSetValues(b,4,j,ab,INSERT_VALUES);
    VecAssemblyBegin(b);
    VecAssemblyEnd(b);

    MatCreate(PETSC_COMM_WORLD,&A);
    MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,4,4);
    MatSetFromOptions(A);
    MatSetUp(A);
    for (i=0; i<4; i++) { // set entries one row at a time
        MatSetValues(A,1,&i,4,j,aA[i],INSERT_VALUES);
    }
    MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);

    KSPCreate(PETSC_COMM_WORLD,&ksp);
    KSPSetOperators(ksp,A,A);
    KSPSetFromOptions(ksp);
    VecDuplicate(b,&x);
    KSPSolve(ksp,b,x);
    VecView(x,PETSC_VIEWER_STDOUT_WORLD);

    KSPDestroy(&ksp); MatDestroy(&A);
    VecDestroy(&x); VecDestroy(&b);
    return PetscFinalize();
}

```

**Code 2.1.** *c/ch2/vecmatksp.c. Solve a small linear system.*

At the setup stage we tell the KSP about the matrix  $A$  via the command

```
KSPSetOperators(ksp,A,A);
```

But why list  $A$  twice? The first  $A$  defines the linear operator. An iterative method uses this **Mat** when applying the matrix to a vector.

However, recalling left- and right-preconditioning equations (2.19) and (2.20), the reason for the second argument  $A$  is that at run time we may choose a preconditioning method which builds  $M^{-1}$  from  $A$ , or from an approximation of  $A$ . The second matrix argument to **KSPSetOperators()** is the **Mat** from which  $M^{-1}$  is built. We do not supply  $M$  itself because doing so would obstruct easy choice among preconditioners at run time. (The user might need to

write extra code for different preconditioners.) Instead we supply “material” from which  $M^{-1}$  is built.

The most common preconditioner material is  $A$  itself. In Jacobi preconditioning, for example, PETSC forms  $M$  by extracting the diagonal from this material. In the LU decomposition the matrix entries are copied from the material and then reordered (e.g., with partial pivoting) and completely transformed into the factors  $L$  and  $U$ . In later chapters we will give examples in which only an approximation of  $A$  is supplied as the material.

To actually solve the system we call

```
KSPSolve(ksp,b,x);
```

This supplies the right-hand side of the system, an allocated and assembled Vec  $b$ , and space for the solution, namely a Vec  $x$  which is often allocated by duplicating the storage layout of  $b$ ; here we use `VecDuplicate()`.

The rest of Code 2.1 is self-explanatory, so we give it a try:

```
$ make vecmatksp
...
$ ./vecmatksp
Vec Object: 1 MPI processes
  type: seq
1.
0.
2.
-1.
```

The reader should, of course, check the correctness of this solution to (2.36).

## Revealing solvers at run time

To see more of what happens when we run `vecmatksp.c`, we could start by viewing the Vec and Mat objects. However, the result of

```
| $ ./vecmatksp -vec_view -mat_view
```

is as expected (and not shown). In fact, output of a Vec or a Mat tells us nothing about the solution process, nor does it give any hints of alternative ways of solving the equations. On the other hand, the following idea cannot be overemphasized.

**Fact 6.** Learning PETSC requires viewing *solver* objects at run time. *If you did not view the solver with `-ksp_view`, `-snes_view`, or `-ts_view` then you probably do not know what it did, even if it succeeded. Viewing solvers is the first step to understanding their composition.*

We will consider the SNES and TS solver types in Chapters 4 and 5, respectively. For now we use `-ksp_view`, with output (slightly) clipped for clarity:

```
$ ./vecmatksp -ksp_view
KSP Object: 1 MPI processes
  type: gmres
    restart=30, using Classical (unmodified) Gram-Schmidt ...
    happy breakdown tolerance 1e-30
  maximum iterations=10000, initial guess is zero
  tolerances: relative=1e-05, absolute=1e-50, divergence=10000.
  left preconditioning
  using PRECONDITIONED norm type for convergence test
```

```

PC Object: 1 MPI processes
  type: ilu
    out-of-place factorization
  0 levels of fill
  tolerance for zero pivot 2.22045e-14
  matrix ordering: natural
  factor fill ratio given 1., needed 1.
  Factored matrix follows:
    Mat Object: 1 MPI processes
      type: seqaij
      rows=4, cols=4
      package used to perform factorization: petsc
      total: nonzeros=16, allocated nonzeros=16
      ...
  linear system matrix = precond matrix:
  Mat Object: 1 MPI processes
    type: seqaij
    rows=4, cols=4
    total: nonzeros=16, allocated nonzeros=16
    ...
  ...
Vec Object: 1 MPI processes
  type: seq
  1.
  0.
  2.
  -1.

```

Here is some of what we learn:

- The default KSP solver is GMRES(30) or `-ksp_gmres_restart 30`, that is, it restarts after 30 iterations so as to limit memory usage.
- The default convergence tolerances are `-ksp_rtol 1.0e-5` and `-ksp_atol 1.0e-50`. (Here the iterations stop when the residual norm has been reduced by  $10^5$ , which happens on the first iteration; see below.)
- Inside every KSP is a PC preconditioner object. (We did not need to ask for one!)
- The PC object has a matrix  $A$  (“`Mat Object:`”), supplied as the second argument to `KSPSetOperators()`.
- We see “`linear system matrix = precond matrix`” because we supplied the same `Mat` as both arguments to `KSPSetOperators()`.
- The serial default PC is left preconditioning using incomplete LU factorization [64, 112], and “`0 levels of fill`”, thus it is ILU(0).

While option `-ksp_view` tells us what the solver *is*, option `-ksp_monitor` tracks what it *does*. In this case, the iteration is short:

```

$ ./vecmatksp -ksp_monitor
0 KSP Residual norm 2.449489742783e+00
1 KSP Residual norm 1.520235486122e-15

```

The residual drops by 15 orders of magnitude in one iteration because GMRES sees a preconditioned system (2.19) with  $M^{-1}A = I$ . That is, the ILU operation on this particular  $A$  is actually

a full LU factorization:  $M = LU = A$ . Every entry is allocated so the exact LU factorization requires no further “fill-in.” (By definition, “fill-in” occurs in a sparse matrix operation if previously un-allocated zeros are replaced by nonzeros, thus using more memory.) Examples below, and in later chapters, are more representative because they involve actually sparse matrices.

Now, a direct solver can be chosen more deliberately. First do

```
| $ ./vecmatksp -help | grep ksp_type
|   -ksp_type <gmres>: Krylov method (one of) cg ... preonly ... (KSPSetType)
```

and

```
| $ ./vecmatksp -help | grep pc_type
|   -pc_type <ilu>: Preconditioner (one of) none jacobi ... lu ... (PCSetType)
```

For direct solvers we typically *do not* want iterations, and one of the KSP options is “`preonly`.” Of the many available PC types, one preconditioner is “`lu`,” a full LU decomposition. Thus a direct solver combination is

```
| $ ./vecmatksp -ksp_type preonly -pc_type lu
```

One could also use `-pc_type svd`.

When solving on multiple MPI processes, using ILU as the default preconditioner makes no sense. Even when fill-in is avoided, a LU factorization algorithm could involve a great deal of interprocess communication.<sup>12</sup> However, the ILU method can be applied to diagonal blocks of  $A$ , i.e., the block of entries corresponding to Vec indices owned by a process, and the result treated as an approximation  $M^{-1} \approx A^{-1}$ , a block-diagonal or *block Jacobi* preconditioner. This defines the default parallel KSP/PC composition:

```
-ksp_type gmres -pc_type bjacobi -sub_pc_type ilu
```

Note that inside the `bjacobi` PC is a `sub_` PC object, which defaults to `ILU(0)` applied to each diagonal block without interprocess communication. (There is also a `sub_` KSP object inside `bjacobi`, and it is `preonly` by default.) The reader can confirm this situation by running, for example,

```
| $ mpiexec -n 2 ./vecmatksp -ksp_view
```

## A tridiagonal linear system

The next example `tri.c` (Code 2.2) introduces the following concepts:

- Creating an integer option, using `PetscOptionsInt()`, to determine the size of the linear system.
- Assembling a `Mat` of arbitrary size across an arbitrary number of processes, using `MatGetOwnershipRange()` to only fill locally owned rows.
- Setting up a known exact solution to the linear system, and then computing and displaying the numerical error.

<sup>12</sup>Advanced sparse direct solvers like MUMPS and SuperLU are available through external packages. See [10] and [www.mcs.anl.gov/petsc/miscellaneous/external.html](http://www.mcs.anl.gov/petsc/miscellaneous/external.html).

```

static char help[] = "Solve a tridiagonal system of arbitrary size.\n"
"Option prefix = tri_.\n";

#include <petsc.h>

int main(int argc,char **args) {
  Vec x, b, xexact;
  Mat A;
  KSP ksp;
  PetscInt m = 4, i, lstart, lend, j[3];
  PetscReal v[3], xval, errnorm;

  PetscInitialize(&argc,&args,NULL,help);

  PetscOptionsBegin(PETSC_COMM_WORLD,"tri_","options for tri","");
  PetscOptionsInt("-m","dimension of linear system","tri.c",m,&m,NULL);
  PetscOptionsEnd();

  VecCreate(PETSC_COMM_WORLD,&x);
  VecSetSizes(x,PETSC_DECIDE,m);
  VecSetFromOptions(x);
  VecDuplicate(x,&b);
  VecDuplicate(x,&xexact);

  MatCreate(PETSC_COMM_WORLD,&A);
  MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,m,m);
  MatSetOptionsPrefix(A,"a_");
  MatSetFromOptions(A);
  MatSetUp(A);
  MatGetOwnershipRange(A,&lstart,&lend);
  for (i=lstart; i<lend; i++) {
    if (i == 0) {
      v[0] = 3.0; v[1] = -1.0;
      j[0] = 0; j[1] = 1;
      MatSetValues(A,1,&i,2,j,v,INSERT_VALUES);
    } else {
      v[0] = -1.0; v[1] = 3.0; v[2] = -1.0;
      j[0] = i-1; j[1] = i; j[2] = i+1;
      if (i == m-1) {
        MatSetValues(A,1,&i,2,j,v,INSERT_VALUES);
      } else {
        MatSetValues(A,1,&i,3,j,v,INSERT_VALUES);
      }
    }
    xval = PetscExpReal(PetscCosReal(i));
    VecSetValues(xexact,1,&i,&xval,INSERT_VALUES);
  }
  MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
  VecAssemblyBegin(xexact);
  VecAssemblyEnd(xexact);
  MatMult(A,xexact,b);

  KSPCreate(PETSC_COMM_WORLD,&ksp);
  KSPSetOperators(ksp,A,A);
  KSPSetFromOptions(ksp);
  KSPSolve(ksp,b,x);

  VecAXPY(x,-1.0,xexact);
  VecNorm(x,NORM_2,&errnorm);
  PetscPrintf(PETSC_COMM_WORLD,
"error for m = %d system is |x-xexact|_2 = %.1e\n",m,errnorm);
}

```

```

KSPDestroy(&ksp); MatDestroy(&A);
VecDestroy(&x); VecDestroy(&b); VecDestroy(&xexact);
return PetscFinalize();
}

```

**Code 2.2.** *c/ch2/tri.c*. Solve a tridiagonal linear system.

The first new tool used in Code 2.2, bracketed by `PetscOptionsBegin()` and `PetscOptionsEnd()`, is `PetscOptionsInt()`. The new option `-tri_m` sets the number of equations  $m$  in the linear system. The `Begin` method takes a prefix to distinguish our option from the built-in PETSC options which start with prefixes `-ksp_`, `-vec_`, etc., and the default value  $m = 4$ . Knowing the prefix allows us to find the option and its default value in `-help` output by using `grep`:

```

$ make tri
$ ./tri -help | grep tri_
Option prefix = tri_.
  -tri_m <now 4 : formerly 4>: dimension of linear system (tri.c)
$ ./tri -tri_m 7 -help | grep tri_
Option prefix = tri_.
  -tri_m <now 7 : formerly 4>: dimension of linear system (tri.c)
$ ./tri -ksp_monitor -a_mat_view ::ascii_dense
Mat Object: (a_) 1 MPI processes
  type: seqaij
  3.00000e+00  -1.00000e+00   0.00000e+00   0.00000e+00
  -1.00000e+00   3.00000e+00  -1.00000e+00   0.00000e+00
   0.00000e+00  -1.00000e+00   3.00000e+00  -1.00000e+00
   0.00000e+00   0.00000e+00  -1.00000e+00   3.00000e+00
  0 KSP Residual norm 3.302822756884e+00
  1 KSP Residual norm 5.519370044893e-16
error for m = 4 system is |x-xexact|_2 = 5.1e-16

```

In Code 2.2 the numerical solution `Vec x` is created and configured just as we did in `vecmatksp.c` (Code 2.1). We also allocate `Vecs b` and `xexact` just like `x` by calling `VecDuplicate()`. Note there is a different method for copying the *contents* of `Vecs`, namely `VecCopy()`, which requires that the source and destination `Vecs` are already allocated and have the same layout.

Next we assemble the matrix `A`. It is a boring tridiagonal matrix with 3 on the diagonal and  $-1$  in the super- and subdiagonals, but it is assembled in parallel. Entries are set by the process on which they will be stored using `MatGetOwnershipRange()` which tells our program, running on a particular process (rank), which rows it owns. That is,

```
MatGetOwnershipRange(A,&Istart,&Iend)
```

obtains the starting and ending row indices for the local process, to be used as limits in a `for` loop. We then use `MatSetValues()` to actually set the entries of `A`, and `MatAssemblyBegin/End()` to complete the assembly.

We also need to define the right-hand side `b` and the exact solution `xexact`. The easiest way, at least for demonstration purposes here, is to choose `xexact` and then compute `b` by multiplication:  $\mathbf{b} = \mathbf{A}\mathbf{x}_{\text{exact}}$ . Thus, we set some values for `xexact`—the details are not important—and call `VecAssemblyBegin/End()` on it. Then we compute `b` by `MatMult(A,xexact,b)`. As in `vecmatksp.c` we then set up a `KSP` and call `KSPSolve()` to approximately solve  $\mathbf{Ax} = \mathbf{b}$ .

Though option `-ksp_monitor` prints the residual norm  $\|\mathbf{r}_k\|_2 = \|\mathbf{b} - A\mathbf{x}_k\|_2$ , we also want to see that the error  $\|\mathbf{x} - \mathbf{x}_{\text{exact}}\|_2$  is small when the solver finishes. So, after getting  $\mathbf{x}$  back from `KSPSolve()` we compute the error:

$$\begin{array}{lll} \text{VecAXPY}(\mathbf{x}, -1.0, \mathbf{x}_{\text{exact}}) & | & \mathbf{x} \leftarrow -1 \mathbf{x}_{\text{exact}} + \mathbf{x} \\ \text{VecNorm}(\mathbf{x}, \text{NORM\_2}, \&\text{errnorm}) & | & \text{errnorm} \leftarrow \|\mathbf{x}\|_2. \end{array}$$

Then we print `errnorm` with `PetscPrintf()`.

The linear system assembled by `tri.c` is easy to solve. It is tridiagonal, symmetric positive-definite (SPD), and diagonally dominant—see Exercises 2.16 and 2.17—so almost any Krylov method will be acceptable.

We start by timing a serial solution with the KSP and PC defaults `-ksp_type gmres -pc_type ilu`:

```
$ time ./tri -tri_m 10000
error for m = 10000 system is |x-xexact|_2 = 1.6e-13
real 0.03
user 0.10
sys 0.10
```

(The `time` command used above should be present in any Unix system, and from now on we only list the “real” time when assessing performance.<sup>13</sup>) A time of 0.03 seconds means that the  $m = 10^4$  system here is too small for the comparisons which follow. A time between one and 10 seconds is more useful, and a bit of experimentation with powers of 10 finds that size  $m = 10^7$  takes a few seconds on a laptop, which is about right for our purposes.

We can try a direct solve with Gauss elimination (`-ksp_type preonly -pc_type lu`) or the default parallel solver (`-ksp_type gmres -pc_type bjacobi -sub_pc_type ilu`). We can turn off preconditioning (`-pc_type none`) or switch to Jacobi preconditioning (`-pc_type jacobi`). Furthermore, because our matrix is SPD, we can try the CG iteration (`-ksp_type cg`), the Cholesky direct method [143] (`-ksp_type preonly -pc_type cholesky`), or combine these using the incomplete form of Cholesky [112] as a preconditioner (`-ksp_type cg -pc_type icc`).

Thus Table 2.3 was built from 16 runs, all with  $m = 2 \times 10^7$ :

```
$ timer mpiexec -n P ./tri -tri_m 20000000 -ksp_rtol 1.0e-10 \
-ksp_type X -pc_type Y
```

There are  $P = 1$  or  $P = 4$  processes, and several KSP/PC choices; compare Tables 2.1 and 2.2. Fair comparison of iterative and direct methods requires a tight tolerance (`-ksp_rtol 1.0e-10`) on the former methods. PC notation “`bjacobi+ilu`” corresponds, of course, to options `-pc_type bjacobi -sub_pc_type ilu`.

Looking at Table 2.3, note that this problem is very well behaved for direct methods. No fill-in or pivoting occurs when LU or Cholesky are applied to such a tridiagonal and (strongly) diagonally dominant matrix, so these methods need only  $O(m)$  operations (Exercise 2.16). These results are *not* representative for direct solvers applied to generic systems with  $m = 10^7$  rows. The iterative methods all succeed, and CG is somewhat faster than the general (nonsymmetric) method GMRES.

There is some speedup from  $P = 1$  to  $P = 4$  processes on this desktop, and perhaps a rough factor-of-two speedup applies across methods. Using four cores does not guarantee anywhere near four times speedup; see Chapter 8 for more discussion.

<sup>13</sup>The author uses an alias called `timer` which prints only the `real` time.

**Table 2.3.** Solution times for a tridiagonal SPD system, from program `tri.c`, of dimension  $m = 2 \times 10^7$ .

KSP	PC	P=1	time (s)	P=4	time (s)
preonly	lu		13.75		
	cholesky		14.89		
richardson	jacobi		17.58		8.87
gmres	none		15.50		6.66
	jacobi		15.78		7.13
	ilu		7.28		
	bjacobi+ilu				3.44
cg	none		10.15		4.55
	jacobi		10.48		4.84
	icc		6.95		
	bjacobi+icc				3.04

## Saving and loading linear systems from files

The matrix from `tri.c` is too easy as a test case, and so, to get more realistic and relevant problems, we extract linear systems from examples later in the book.

Any KSP-based code can save its final Mat and right-hand-side Vec with options `-ksp_view_(mat|rhs)`. We apply this technique to generate two additional test matrices from Chapters 7 and 10. A simple code `c/ch2/loadsolve.c` then loads a linear system  $Ax = b$  from a file and solves it with KSP. More precisely, it creates `PetscViewer` objects and does `MatLoad()` for  $A$  from one file and then `VecLoad()` for  $b$  from another. Then it creates a KSP and calls `KSPSolve()`. The key parts of the code look like this:

```

Vec      x, b;
Mat      A;
KSP      ksp;
PetscViewer fileA, fileb;

MatCreate(PETSC_COMM_WORLD,&A);
MatSetFromOptions(A);
PetscViewerBinaryOpen(PETSC_COMM_WORLD,nameA,FILE_MODE_READ,&fileA);
MatLoad(A,fileA);

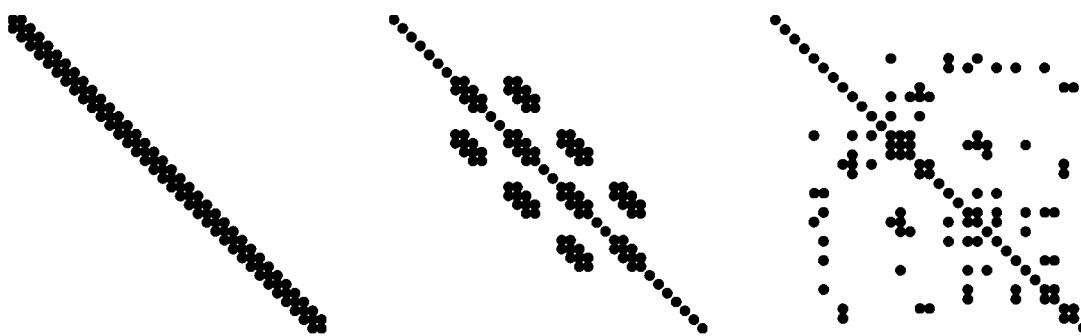
VecCreate(PETSC_COMM_WORLD,&b);
VecSetFromOptions(b);
PetscViewerBinaryOpen(PETSC_COMM_WORLD,nameb,FILE_MODE_READ,&fileb);
VecLoad(b,fileb);

KSPCreate(PETSC_COMM_WORLD,&ksp);
KSPSetOperators(ksp,A,A);
KSPSetFromOptions(ksp);

VecDuplicate(b,&x);
VecSet(x,0.0);
KSPSolve(ksp,b,x);

```

Our first example matrix comes from a structured-grid nonlinear PDE problem, the minimal surface equation solver `minimal.c` in Chapter 7. For a realistic but convenient size we use a  $257 \times 257$  grid, thus the linear system has dimension  $N = 257^2 = 66049$ . To generate this matrix yourself do



**Figure 2.5.** Sparsity patterns of matrices from `tri.c` in this Chapter (left), `minimal.c` in Chapter 7 (middle), and `unfem.c` in Chapter 10 (right).

**Table 2.4.** KSP iterations for a nonsymmetric banded system of dimension  $N = 66049$  from `minimal.c` (Chapter 7).

KSP	PC	Iterations
preonly	lu	1
richardson	jacobi	DIVERGED_ITS
gmres	none	8321
	jacobi	4240
	ilu	265

```
$ cd ..ch7/
$ make minimal
$ ./minimal -snes_fd_color -snes_grid_sequence 7 \
  -ksp_view_mat binary:A.dat -ksp_view_rhs binary:b.dat
$ cp A.dat b.dat ..ch2/
$ cd ..ch2/
```

(The `-snes_` options are explained in Chapters 4 and 7.) Files `A.dat` and `b.dat` now contain a linear system in PETSc binary format.

This matrix  $A$  is banded and contains at most nine nonzeros in each row. Though structurally symmetric ( $a_{ij} \neq 0 \iff a_{ji} \neq 0$ ) it is *not* symmetric. The bandwidth is roughly twice the grid size, namely  $2(257) + 3 = 517$  in this case. The sparsity structure of a small version of the matrix (a  $6 \times 6$  grid giving  $N = 36$ ) is shown in the middle of Figure 2.5.

For each KSP/PC combination, in Table 2.4 we show the number of KSP iterations needed to achieve  $10^{-10}$  reduction in residual norm:

```
$ make loadsolve
$ ./loadsolve -fA A.dat -fb b.dat -ksp_rtol 1.0e-10 \
  -ksp_converged_reason -ksp_type X -pc_type Y
```

Table 2.4 is interesting because of the DIVERGED\_ITS failure and the large number of iterations for other combinations. Though the default maximum KSP iterations (`-ksp_max_it 10000`) is exceeded in the failure case, all of the iterative cases other than `gmres+ilu` reflect very slow convergence. (Exercise 2.20 shows how to visualize slowly progressing iterations.) The table suggests that the iterative KSP/PC combinations are all disappointing, but these results are realistic motivation for the introduction of more-advanced preconditioners in Chapter 6, especially multigrid. (We will indeed find a highly efficient solver for this problem!)

**Table 2.5.** Solution times for an unstructured SPD system of dimension  $N = 41409$  from Chapter 10.

KSP	PC	P=1 time (s)	P=4 time (s)
preonly	lu	0.27	
	cholesky	0.32	
	jacobi	DIVERGED_ITS	DIVERGED_ITS
	gmres	7.29	3.47
	jacobi	5.51	2.53
	ilu	1.35	
cg	bjacobi+ilu		2.03
	none	0.49	0.36
	jacobi	0.51	0.35
	icc	0.32	
bjacobi+icc			0.37

Our second example matrix comes from a Chapter 10 Poisson problem discretized by the finite element method on an unstructured mesh. Generating the matrix is a bit more involved than the last example, and so we do not document it here, but it results in a matrix of dimension  $N = 41409$ . The Mat for a lower-dimensional ( $N = 33$ ) version, on the same domain but using a much coarser mesh, is shown in Figure 2.5. Typical of matrices from unstructured meshes, the matrix is not banded, but it is SPD so we may apply CG and Cholesky methods. We build timing Table 2.5 using `loadsolve.c`, as in the last example, running a `-with-debugging=0` PETSC configuration on a 4-core laptop.

Convergence occurs, for the same methods used in Table 2.3, except for the Richardson iteration. Now there is clear evidence that iterative methods exploiting symmetry are faster—compare the CG and GMRES results—but the iterations for these methods are large (not shown).

Tables 2.3–2.5 suggest some basic conclusions about solving linear systems:

- (i) In the absence of good preconditioning, norm-minimizing Krylov iterations like CG and GMRES may not generate fast convergence.
- (ii) Serial direct solvers (LU and Cholesky), used with matrix ordering so as to eliminate fill-in, are competitive on problems of the type and size tested.
- (iii) ILU-preconditioning may or may not be fast—the evidence is mixed—but GMRES/ILU seems to be a robust combination, and we can understand why it is the PETSC default combination.

The  $P = 4$  runs are not four times faster than the serial runs, and it is not even automatic that they are faster. However, the problems under consideration are small. Furthermore, solving sparse linear systems on a 4-core processor is limited by memory bandwidth and contention issues, not just floating-point performance. We will need larger problems, and machines of a different architecture, i.e., cluster-type architectures, to see significant speedup in parallel (Chapter 8).

Furthermore, we have not yet introduced the performance metrics that really matter. We actually want to know how iterations, floating-point operations, and run times scale as the size  $N$  of the discrete linear systems, arising from a given PDE problem, increases. It will turn out that preconditioners which exploit the PDE origin of problems (Chapter 6) can generate major improvements.

## Parallel preconditioning

Another numerical “fact of life” relates to preconditioning and iterative methods in parallel. We demonstrate it using `tri.c` runs:

```
$ mpexec -n P ./tri -tri_m 100 -ksp_type cg -pc_type bjacobi \
    -sub_pc_type icc -ksp_converged_reason
```

Comparing  $P = 2$  and  $P = 20$ , we see convergence to the default tolerance in 3 and 5 iterations, respectively.

**Fact 7.** Parallel preconditioning generally depends on processor count. *When the number of MPI processes changes, a block-matrix or domain-decomposition preconditioner also changes.*

The `bjacobi` and `asm` methods (Table 2.1 and Chapter 6) are examples of preconditioners which act differently depending on the number of MPI processes. The dependence arises because the preconditioning matrices  $M^{-1}$  applied at each Krylov iteration are actually different. For example, on  $P$  processes the block Jacobi matrix from `-pc_type bjacobi -sub_pc_type icc` is

$$M^{-1} = \begin{bmatrix} M_0^{-1} & & & \\ & M_1^{-1} & & \\ & & \ddots & \\ & & & M_{P-1}^{-1} \end{bmatrix}, \quad (2.37)$$

where each block  $M_i$  is the product of the `icc` factors acting on those rows which are owned by the rank  $i$  process.

By contrast, one example of a preconditioner which does not depend on  $P$  is `-pc_type jacobi`, but it is a weaker preconditioner. As the reader may check using `-ksp_monitor`, runs like the above, but using `-pc_type jacobi` instead of `-pc_type bjacobi -sub_pc_type icc`, converge in 12 iterations and have residual norm and numerical error results which are independent of  $P$ .

Good preconditioners, which must both act quickly *and* significantly improve the spectrum, tend to depend on parallel decomposition because they avoid interprocess communication. That is, high-quality spectral effect requires using multirow information, but speed suggests avoiding communicating information between rows on different processors. Looking ahead, this trade-off especially applies to domain-decomposition (`asm`) preconditioners, but it also applies to the smoother components in PETSc’s geometric multigrid implementation (`mg`; see Chapter 6).

## Exercises

- 2.1. Suppose a square matrix  $A$  with nonzero diagonal entries is decomposed into diagonal and lower/upper triangular parts as  $A = D + L + U$ . Show that the Jacobi iteration  $\mathbf{u}_{k+1} = D^{-1}(\mathbf{b} - (L + U)\mathbf{u}_k)$  is the same as the  $\alpha = 1$  Richardson iteration (2.12) applied to the left-preconditioned system (2.19) with  $M = D$ . Make a corresponding statement about the Gauss-Seidel iteration  $\mathbf{u}_{k+1} = (D + L)^{-1}(\mathbf{b} - U\mathbf{u}_k)$ .
- 2.2. Show (2.11).
- 2.3. As stated, (2.18) is more of a slogan than a theorem. Let  $B \in \mathbb{R}^{N \times N}$ , and prove the following more precise statements:
  - (i) If (2.17) converges for all  $\mathbf{u}_0, \mathbf{c} \in \mathbb{R}^N$  then  $\rho(B) < 1$ .
  - (ii) Fix  $\mathbf{u}_0, \mathbf{c} \in \mathbb{R}^N$ . If  $\rho(B) < 1$  then (2.17) converges.

(The proof in part (i) can be done with elementary eigenvalue facts. For part (ii) one may replace (2.17) with a simpler equation for the error  $\mathbf{e}_k = \mathbf{u}_k - \mathbf{u}$ , where  $\mathbf{u}$  solves  $\mathbf{u} = B\mathbf{u} + \mathbf{c}$ , and then use (a)  $\rho(B) < 1$  if and only if  $\lim_{k \rightarrow \infty} \|B^k\| = 0$  in any norm [131], or (b) if  $\rho(B) < 1$  then there is a matrix norm  $\|\cdot\|'$  such that  $\|B\|' < 1$  [66].)

- 2.4. Suppose  $A$  and  $M$  are square matrices of the same size and that  $M$  is invertible. Show that  $M^{-1}A$  and  $AM^{-1}$  are similar. (Thus they have the same eigenvalues.) Show that if  $M = M_1M_2$ , where  $M_i$  are invertible, then  $M_1^{-1}AM_2^{-1}$  is similar to  $M^{-1}A$ .
- 2.5. (i) Show that the error  $\mathbf{e}_k = \mathbf{u}_k - \mathbf{u}$  in simple iteration (2.23) satisfies

$$\mathbf{e}_{k+1} = (I - M^{-1}A)\mathbf{e}_k. \quad (2.38)$$

It follows that simple iteration converges for all  $\mathbf{e}_0$  if and only if  $\rho(I - M^{-1}A) < 1$ .

- (ii) By observing that  $A(I - BA) = (I - AB)A$ , show that the residuals from simple iteration satisfy

$$\mathbf{r}_{k+1} = (I - AM^{-1})\mathbf{r}_k. \quad (2.39)$$

Exercise 2.4 then shows the equivalence of convergence of errors and residuals.

- 2.6. Give examples of matrices  $A$  and  $M$  which are symmetric but for which  $M^{-1}A$  and  $AM^{-1}$  are not symmetric. (Hint. Choose wildly.)
- 2.7. Suppose  $M = LL^\top$  for some square, invertible matrix  $L$ . (This implies  $M$  is SPD.) Show that

$$(L^{-1}AL^{-\top})\mathbf{w} = L^{-1}\mathbf{b}, \quad L^\top\mathbf{u} = \mathbf{w} \quad (2.40)$$

is equivalent to the original system (2.6). (Here  $L^{-\top}$  denotes the matrix  $(L^{-1})^\top = (L^\top)^{-1}$ . Note that  $L^{-1}AL^{-\top}$  is symmetric if  $A$  is symmetric. One might apply (2.40) with a computed Cholesky factorization  $M = LL^\top$ , but actually factoring  $M$  is not needed to apply such symmetric preconditioning; see Exercise 2.9 below.)

- 2.8. The following pseudocode gives the CG algorithm:

```

function CG( $A, \mathbf{b}, \mathbf{u}_0$ )
     $\mathbf{r}_0 = \mathbf{b} - A\mathbf{u}_0$ 
     $\mathbf{p}_0 = \mathbf{r}_0$ 
    for  $k = 1, 2, \dots$ 
         $\alpha_{k-1} = (\mathbf{r}_{k-1}^\top \mathbf{r}_{k-1}) / (\mathbf{p}_{k-1}^\top A \mathbf{p}_{k-1})$ 
         $\mathbf{u}_k = \mathbf{u}_{k-1} + \alpha_{k-1} \mathbf{p}_{k-1}$ 
         $\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_{k-1} A \mathbf{p}_{k-1}$ 
        if  $\|\mathbf{r}_k\|$  is small enough
            return  $\mathbf{u}_k$ 
         $\beta_{k-1} = (\mathbf{r}_k^\top \mathbf{r}_k) / (\mathbf{r}_{k-1}^\top \mathbf{r}_{k-1})$ 
         $\mathbf{p}_k = \mathbf{r}_k + \beta_{k-1} \mathbf{p}_{k-1}$ 

```

This algorithm actually requires one matrix-vector multiply, two inner products, and three *axpy* (i.e.,  $\mathbf{w} = \alpha\mathbf{x} + \mathbf{y}$ ) operations per iteration of the inner loop. Define temporary variables and rewrite the above pseudocode so that this is clear. (Note that parallel CG needs two reductions per iteration, namely the inner products.)

- 2.9. Suppose  $M = LL^\top$  is SPD. Consider the symmetrically preconditioned system (2.40) as  $\hat{A}\mathbf{w} = \hat{\mathbf{b}}$  where  $\hat{A} = L^{-1}AL^{-\top}$  and  $\hat{\mathbf{b}} = L^{-1}\mathbf{b}$ . In these terms we define the symmetrically preconditioned conjugate gradient (PCG) algorithm as

$$\mathbf{u}_k = L^{-\top} \text{CG}(\hat{A}, \hat{\mathbf{b}}, L^\top \mathbf{u}_0). \quad (2.41)$$

Show that (2.41) can be computed by the following algorithm which *does not actually use the factor L*:

```

function PCG( $M, A, \mathbf{b}, \mathbf{u}_0$ )
   $\mathbf{r}_0 = \mathbf{b} - A\mathbf{u}_0$ 
  solve  $M\mathbf{z}_0 = \mathbf{r}_0$ 
   $\mathbf{p}_0 = \mathbf{z}_0$ 
  for  $k = 1, 2, \dots$ 
     $\alpha_{k-1} = (\mathbf{r}_{k-1}^\top \mathbf{z}_{k-1}) / (\mathbf{p}_{k-1}^\top A\mathbf{p}_{k-1})$ 
     $\mathbf{u}_k = \mathbf{u}_{k-1} + \alpha_{k-1}\mathbf{p}_{k-1}$ 
     $\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_{k-1}A\mathbf{p}_{k-1}$ 
    if  $\|\mathbf{r}_k\|$  is small enough
      return  $\mathbf{u}_k$ 
    solve  $M\mathbf{z}_k = \mathbf{r}_k$ 
     $\beta_{k-1} = (\mathbf{r}_k^\top \mathbf{z}_k) / (\mathbf{r}_{k-1}^\top \mathbf{z}_{k-1})$ 
     $\mathbf{p}_k = \mathbf{z}_k + \beta_{k-1}\mathbf{p}_{k-1}$ 

```

- 2.10. On page 17 we noted that an approximation  $p_{n-1}(A)\mathbf{b}$  to the solution  $\mathbf{u} = A^{-1}\mathbf{b}$  of (2.6) is accurate if  $p_{n-1}(z)$  is close to  $1/z$  on the spectrum of  $A$ . Prove this for invertible diagonalizable matrices  $A = S\Lambda S^{-1}$ :

$$\|p_{n-1}(A) - A^{-1}\| \leq \kappa(S) \max_{\lambda \in \sigma(A)} |p_{n-1}(\lambda) - \lambda^{-1}|.$$

(Here  $\|\cdot\|$  is any induced matrix norm. Note that if  $A$  is normal then  $S$  can be chosen to be unitary, in which case  $\kappa(S) = 1$  in the 2-norm.)

- 2.11. Consider polynomials  $q_k$  given by (2.28). Show that if  $\lim_{k \rightarrow \infty} q_k = q_\infty$  exists then  $q_\infty(x) = 1/x$ . On the other hand, by setting  $y = 1 - x$  and defining  $Q_k(y) = q_k(1 - y)$ , show  $Q_k(y)$  is the partial sum of a series with a well-known radius of convergence.  
 2.12. Show that (2.34) follows from (2.33).  
 2.13. In the text describing `vecmatksp.c` we assert that communication might occur during the `VecAssemblyBegin/End()` calls. However, as `vecmatksp.c` is written, each process has all the values it needs. To demonstrate a case where communication occurs, modify `vecmatksp.c` to a new program `vmkrankzero.c` by adding

```
MPI_Comm_rank(PETSC_COMM_WORLD,&rank)
```

(Chapter 1). Then surround the `VecSetValues()` and `MatSetValues()` lines in `vecmatksp.c` with the conditional

```
if (rank == 0) { ... }
```

so that values are only set on a single process, even in parallel. Check that this version of the code gives the same serial and parallel results for the solution of the linear system.

- 2.14. In the following example Richardson iteration seems to converge in one iteration:

```
| $ ./tri -tri_m 100 -ksp_monitor -ksp_type richardson
```

Recall, however, that the PETSc default preconditioner is `-pc_type ilu`. Using ILU is actually a complete LU factorization on this tridiagonal and diagonally dominant  $A$ , and we are really seeing a direct solve. Confirm that, as with the example on page 15, Richardson iteration succeeds with `-pc_type jacobi` but diverges with `-pc_type none`.

- 2.15. The accuracy of direct solves (e.g., `-ksp_type preonly -pc_type cholesky`) in `tri.c`, as measured by the reported error norm  $\|\mathbf{x} - \mathbf{x}_{\text{exact}}\|_2$ , decreases with increasing dimension. Confirm and explain.

- 2.16. There is a well-known  $O(N)$  algorithm for solving SPD tridiagonal linear systems, namely Gauss elimination without pivoting. Write a pseudocode for this algorithm. Find constants  $a, b$  in the work estimate (flops) =  $aN + b$ . (*Pivoting is not needed for stability in the SPD case* [80].)
- 2.17. GMRES without preconditioning solves the linear system in `tri.c` reasonably efficiently; confirm this. We can explain this by asking PETSC to compute the eigenvalues of the unpreconditioned matrix  $A$ :<sup>14</sup>

```
-ksp_view_eigenvalues -pc_type none
```

Try dimensions  $m = 10, 100, 1000$ . Why does the  $m = 1000$  run only show 11 eigenvalues? How do these eigenvalues explain the good behavior of unpreconditioned GMRES?

- 2.18. Table 2.3 includes a number of blanks. For each one, explain why it is blank, experimenting if needed.
- 2.19. Table 2.3 gives run times not KSP iteration counts. Generate the corresponding iteration table by adding option `-ksp_converged_reason` to the run commands. Explain the “coincidences” in iteration count. Which preconditioners have a strong or weak effect?
- 2.20. PETSC can generate X Window line graphics showing KSP convergence:

```
| $ ./tri -tri_m 1000000 -ksp_rtol 1.0e-10 -pc_type jacobi \
|   -ksp_monitor_lg_true_residualnorm -draw_pause 1
```

This shows both the preconditioned and true residual norm logarithms versus the iteration number. Use this visualization to see that the DIVERGED\_ITS case in Table 2.4 is actually making slow progress.

- 2.21. One convergence test in `KSPSolve()` requires that  $\|\mathbf{r}\| < \text{DTOL} \|\mathbf{b}\|$ , with tolerance set by `-ksp_divtol DTOL` and default value  $10^4$ . Oddly, this means that KSP fails on the trivial linear system  $I\mathbf{u} = 0$  if the initial guess is nonzero. Confirm this.
- 2.22. As the reader will undoubtedly experience, segmentation faults and memory leaks occur when developing PETSC codes. A recommended tool for detection and diagnosis is `valgrind`; see [valgrind.org](http://valgrind.org). Run

```
| $ valgrind ./tri
```

to see what `valgrind` shows for a leak-free program. Then comment out a `VecDestroy()` call in `tri.c` to see a common type of memory leak.

---

<sup>14</sup>As stated in [10], option `-ksp_view_eigenvalues` is “intended only for assistance in understanding the convergence of iterative methods, not for eigenanalysis. For accurate computation of eigenvalues we recommend using the excellent package SLEPc.” See [slepc.upv.es](http://slepc.upv.es).

## Chapter 3

# Poisson equation on a structured grid

In this chapter we solve the Poisson equation on a square, a straightforward PDE problem on which to learn key parts of PETSc. We will also return to this problem—it is a cliché—in later chapters. The code here introduces DM objects, specifically a DM type, to build a structured grid. Then it uses a finite difference (FD) discretization to assemble a linear system based on this grid, and it solves the linear system using a KSP solver object. Both the assembly and the solution processes are in parallel, with each process responsible for its portion of the grid.

### Poisson problem on a square domain

Let  $\mathcal{S}$  be the open unit square  $(0, 1) \times (0, 1)$  with boundary  $\partial\mathcal{S}$ . The following *Poisson equation* problem is shown in Figure 3.1:

$$-\nabla^2 u = f \quad \text{on } \mathcal{S}, \tag{3.1}$$

$$u = 0 \quad \text{on } \partial\mathcal{S}. \tag{3.2}$$

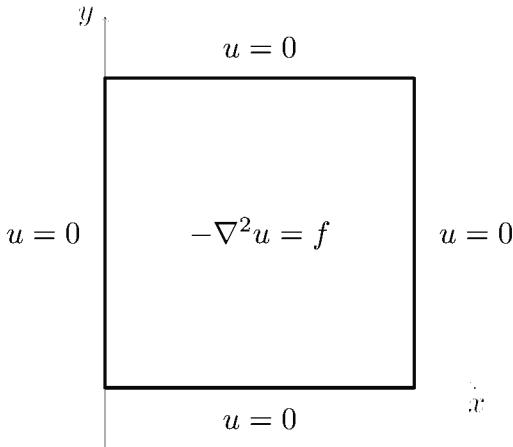
So that we may use its pointwise values, we assume that  $f(x, y)$  is continuous and bounded on  $\mathcal{S}$ .

A few words on the context of this problem are appropriate before we dive into numerical solutions. Equation (3.1) is a problem of inverting the *Laplacian* operator

$$\nabla^2 u = \nabla \cdot (\nabla u) = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{3.3}$$

for  $u$ . This operator appears in mathematical models through an assumption that a flux is proportional to the gradient [119], and because a divergence connects a boundary integral of the flux with an interior integral. (Recall the divergence or Gauss-Green theorem, e.g., as presented in [51, Appendix C].)

The Poisson equation (3.1) may model the electrostatic potential, the equilibrium distribution from random walks, or other physical phenomena. For a thermodynamical example, Fourier's law for heat conduction in solids says that the heat flux is  $\mathbf{q} = -k\nabla u$ , where  $k$  is the conductivity. On the other hand, given an additional heat source  $f$  within the domain, conservation of energy [119] implies  $c\rho\partial u/\partial t = -\nabla \cdot \mathbf{q} + f$ . (The product of the heat capacity  $c$  and the density  $\rho$  parameterizes the ability of the material to hold heat by a gain in temperature.) If  $k$  is constant then, in steady state when  $\partial u/\partial t = 0$ , these equations combine to give Poisson's equation  $0 = k\nabla^2 u + f$ .



**Figure 3.1.** Poisson equation on a square with homogeneous Dirichlet boundary conditions.

Equation (3.2) states *homogeneous Dirichlet boundary conditions*. If  $f$  were zero and the value of  $u$  along  $\partial\mathcal{S}$  were given by some nonzero  $g$  then one would call this the *Dirichlet problem*. However, we call all forms of the same basic problem “Poisson” for simplicity. Various boundary conditions appear later (e.g., Chapter 10), but we always allow nonzero right-hand sides  $f$ . For Dirichlet boundary conditions, standard theory says that a unique solution  $u(x, y)$  exists and is continuous on the closed square  $\bar{\mathcal{S}}$  [51, Theorem 6 in section 5.6].

Instead of  $u$  itself, the normal component of the heat flux  $-k\nabla u$  could be set to known values along the boundary, so-called *Neumann boundary conditions*. In that case the Poisson problem is not well posed because if  $u$  is a solution then  $v = u + c$  is also a solution for any constant  $c$ . Finally, in the absence of any boundary conditions there is an infinite-dimensional space of solutions, i.e., to the unconstrained PDE  $\nabla^2 w = 0$  on  $\mathcal{S}$ , namely the space of all *harmonic functions* [51] on  $\mathcal{S}$ .

## Creating structured grids

We apply a *finite difference* (FD) method as a way to generate a linear system from the problem consisting of (3.1) and (3.2). Our method is based on a 2D grid of  $m_x \times m_y$  points on the unit square. The grid points have spacing  $h_x = 1/(m_x - 1)$  and  $h_y = 1/(m_y - 1)$  and coordinates  $x_i = i h_x$  and  $y_j = j h_y$ , for  $i = 0, 1, \dots, m_x - 1$  and  $j = 0, 1, \dots, m_y - 1$ , respectively (Figure 3.2).

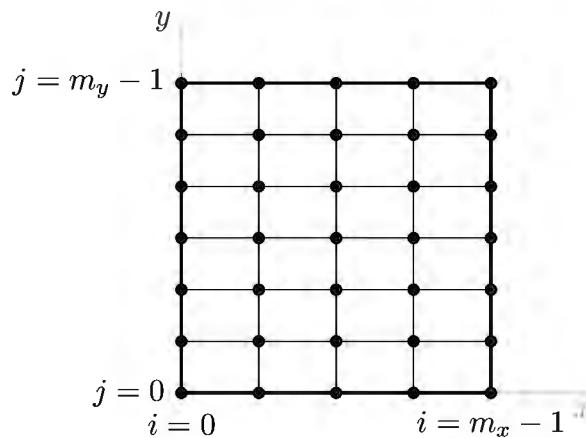
A new PETSc object is used to construct this grid. Consider the following code extract:

```
DM da;
DMDACreate2d(PETSC_COMM_WORLD,
               DM_BOUNDARY_NONE, DM_BOUNDARY_NONE, DMDA_STENCIL_STAR,
               9, 9, PETSC_DECIDE, PETSC_DECIDE, 1, 1, NULL, NULL, &da);
```

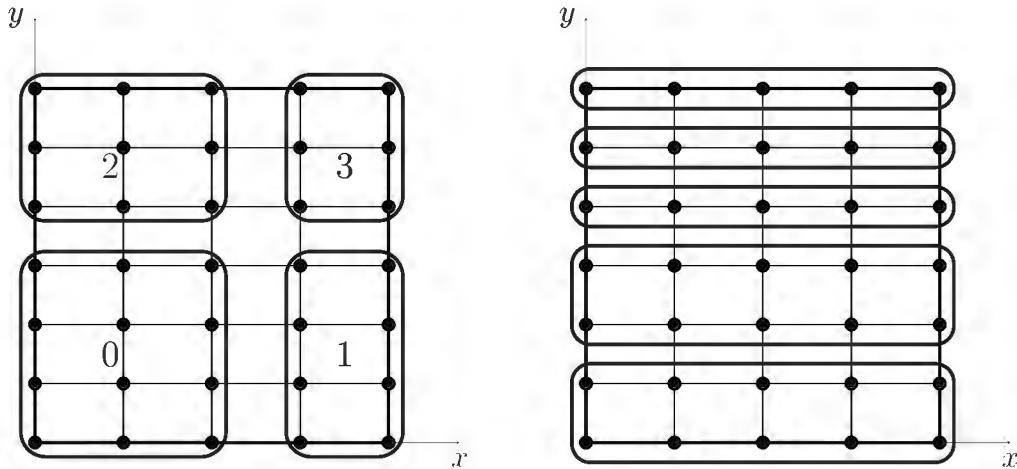
In general, DM types describe the topology (connectedness) of grids and meshes and the manner in which data on these grids is distributed across MPI processes. This specific case is a DMDA type, a structured-grid subclass of DM. The name “DM” variously stands for data management [10], distribution manager [109], or distributed mesh. “DA” is for distributed array.

If we do

```
$ cd c/ch3/
$ make poisson
$ ./poisson -da_grid_x 5 -da_grid_y 7
on 5 x 7 grid: error |u-uexact|_inf = 0.00217606
```



**Figure 3.2.** A grid on the unit square  $S$ , here with  $m_x = 5$  and  $m_y = 7$ .



then the grid in Figure 3.2 is created and the Poisson equation is approximately solved using this grid. In this case all nodes are “owned” by a single MPI process. One may instead use multiple MPI processes:

```
| $ mpiexec -n P ./poisson -da_grid_x MX -da_grid_y MY
```

PETSC then does its best to balance  $MX \times MY$  grid points across  $P$  processes, with the restriction that each process owns a rectangular subgrid. For example, values  $P = 4$ ,  $MX = 5$ , and  $MY = 7$  lead to the distribution in Figure 3.3 (left). Neither 5 nor 7 is divisible by 2, but PETSC distributes the four ranks across the 35 grid points so that the load is somewhat balanced, and on larger grids better balance is expected. However, notice that if the number of processes  $P$  is prime then we get far-from-square domains. For instance, if  $P = 5$  and  $MX, MY$  are the same then the distribution is as in Figure 3.3 (right). Each process’s domain has large perimeter-to-area ratio and so interprocess communication will be substantial.

To explain `DMDACreate2d()` further we paraphrase from the manual [10]:

```
DMDACreate2d(MPI_Comm comm, DMBoundaryType bx, DMBoundaryType by,
  DMDAStencilType stype, PetscInt M, PetscInt N, PetscInt m, PetscInt n,
  PetscInt dof, PetscInt s, const PetscInt lx[], const PetscInt ly[],
  DM *da)
```

`comm` MPI communicator

`bx,by` type of ghost nodes at boundary: `DM_BOUNDARY_NONE|GHOSTED|PERIODIC`

`stype` stencil type: `DMDA_STENCIL_BOX|STAR`

`M,N` global dimension in each direction of the array; this can be set at the command line with a combination of `-da_grid_x M`, `-da_grid_y N`, and `-da_refine`

`m,n` number of processes in each dimension (or `PETSC_DECIDE`)

`dof` number of degrees of freedom per node

`s` stencil width: the number of grid points away from the center of the stencil

`lx,ly` arrays containing the number of nodes in each process' portion of the grid, or `NULL`; if non-null, these must be of length `m` and `n`, respectively, and the sums of `lx[]` and `ly[]` must be `M` and `N`, respectively

`da` the resulting `DMDA` object

In the extract shown earlier (page 44), the second and third arguments are `DM_BOUNDARY_NONE` because Dirichlet boundary conditions do not need extra neighbors or periodic wrapping. In our particular FD method, discussed below, the nodes along the boundary will be unknowns, but no storage is needed *beyond* the boundary points, as might be needed in some schemes. In the fourth argument we choose `DMDA_STENCIL_STAR` because only cardinal neighbors of a grid point are used in our FD scheme (below).

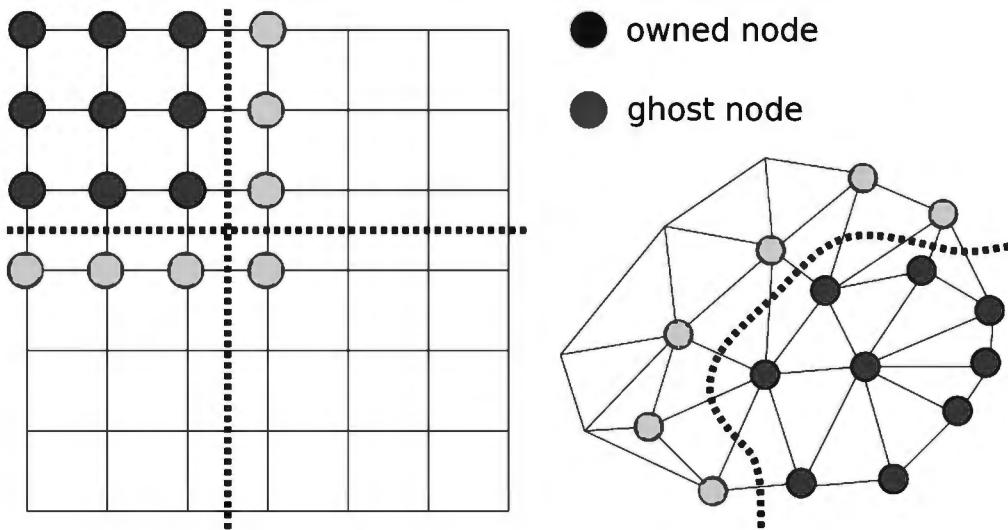
The fifth and sixth arguments used in the extract set a  $9 \times 9$  default grid. These defaults can be overridden by options `-da_grid_x|y` and/or by *refining* by factors of two. More precisely, option `-da_refine` specifies how many times the number of grid subintervals in each dimension will be increased by factors of two. Thus the following option replaces our default grid by a  $17 \times 17$  grid:

```
| $ ./poisson -da_refine 1
on 17 x 17 grid: error |u-uexact|_inf = 0.000196764
```

The next two `PETSC_DECIDE` arguments in the extract (page 44) tell PETSC to use its internal logic to distribute the grid over processes according to the size of the MPI communicator. The two arguments after that identify the PDE as scalar (`dof = 1`) and say that the FD scheme only needs one neighbor in each direction (`s = 1`); see more below on stencils. The next two `NULL` arguments indicate that we are *not* telling PETSC any details about how to distribute processes over the grid; it decides for itself.<sup>15</sup> Finally, the `DMDA` object is returned via a reference (pointer argument).

---

<sup>15</sup>For integer arguments one generally uses `PETSC_DECIDE` to have PETSC select the value, but for array/pointer arguments one uses `NULL` when not specifying a value.



**Figure 3.4.** Parallel decomposition of a structured grid (left) and an unstructured mesh (right), using some type of DM object. One distinguishes between the mesh nodes owned by the process (solid) and those which are allocated locally but have values duplicated from other processes (“ghost nodes”; gray). (Figure modified from [10].)

The standard views of parallel DMs are in Figure 3.4. Our DMDA code in this chapter corresponds to the left part, except that a DMDA\_STENCIL\_BOX stencil is shown in the figure and we use DMDA\_STENCIL\_STAR. On the right is an unstructured mesh, for which one may use type DMPlex; see Chapter 13. The figure shows the nodes owned by a given process, sometimes called “local” nodes in PETSc documentation, and the other nodes that are allocated in the memory space of a process but which need to contain duplicated values from the neighboring processes. These “ghost” nodes allow a process to set up the equation for each owned node.

## A finite difference method

If  $F(x)$  is sufficiently smooth then, by a Taylor’s theorem argument [115],

$$F''(x) = \frac{F(x-h) - 2F(x) + F(x+h)}{h^2} + O(h^2) \quad (3.4)$$

as  $h \rightarrow 0$ . This formula, applied to partial derivatives, approximates the Laplacian in (3.1). In fact, if  $u_{i,j}$  is the gridded approximation to the exact value  $u(x_i, y_j)$ , and if  $f_{i,j} = f(x_i, y_j)$ , then from (3.4) we have this FD approximation to equation (3.1):

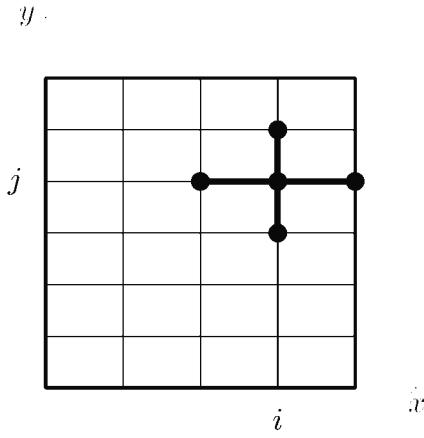
$$-\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h_x^2} - \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_y^2} = f_{i,j}. \quad (3.5)$$

While we compute the values  $u_{i,j}$  from such finite difference equations, in typical PDE problems we will not know the exact values  $u(x_i, y_j)$ . The *numerical error* at each point is the difference  $e_{i,j} = u_{i,j} - u(x_i, y_j)$ .

Equation (3.5) applies at all interior points, i.e., for  $1 \leq i \leq m_x - 2$  and  $1 \leq j \leq m_y - 2$ . The boundary conditions (3.2) become

$$u_{0,j} = 0, \quad u_{m_x-1,j} = 0, \quad u_{i,0} = 0, \quad u_{i,m_y-1} = 0 \quad (3.6)$$

for all  $i, j$ . We regard all values  $u_{i,j}$  as unknowns, whether on the boundary or in the interior.



**Figure 3.5.** The 5-point star stencil for FD scheme (3.5).

At interior grid locations  $(x_i, y_j)$ , equation (3.5) relates  $u_{i,j}$  to its four cardinal neighbors  $u_{i+1,j}$ ,  $u_{i-1,j}$ ,  $u_{i,j+1}$ , and  $u_{i,j-1}$ , in a *star stencil* (Figure 3.5). By contrast, a *box stencil* also includes the four diagonal neighbors, thus nine points.

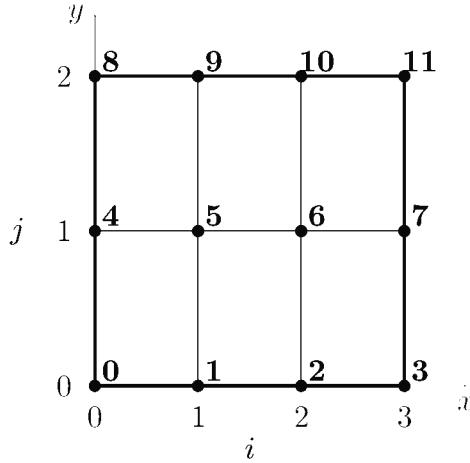
Equations (3.5) and (3.6) form a linear system  $Au = b$  of  $N = m_x m_y$  equations in  $N$  unknowns. To identify entries of the matrix  $A$  and the right-hand side  $b$  in the linear system we must order the unknowns, and the ordering is implemented by the DMDA. Our code (`poisson.c` below) will only use the gridwise coordinates  $(i, j)$ . The ability to assemble `Mats` and `Vecs` using only grid indexing is one reason that structured-grid codes using DMDA can be quite short. The ordering in this case is shown in Figure 3.6, and the global index is  $k = j m_x + i$  for the  $k$ th unknown in the system, but such a formula appears nowhere in `poisson.c`; it is internal to DMDA.

**Example 3.1.** In the  $m_x = 4$  and  $m_y = 3$  case (Figure 3.6) we have grid spacing  $h_x = 1/3$  and  $h_y = 1/2$ . Only the  $k = 5, 6$  equations are not boundary conditions (3.6). Using (3.5) as stated, the linear system (3.8) is

$$\begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 1 & \\ c & & b & a & b & & c \\ & c & & b & a & b & & c \\ & & & & & 1 & & \\ & & & & & & 1 & \\ & & & & & & & 1 \end{bmatrix} \begin{bmatrix} u_{0,0} \\ u_{1,0} \\ u_{2,0} \\ u_{3,0} \\ u_{0,1} \\ u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{0,2} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ f_{1,1} \\ f_{2,1} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

where  $a = 2/h_x^2 + 2/h_y^2 = 26$ ,  $b = -1/h_x^2 = -9$ , and  $c = -1/h_y^2 = -4$ . The matrix is not symmetric, and its 2-norm condition number is  $\kappa(A) = 43.16$ .

Now, before writing code, we make two observations which lead to an equivalent linear system which can be solved faster and more accurately. First, equations (3.5) have very different scaling from equations (3.6). For example, if  $m_x = m_y = 1001$ , so  $h_x = h_y = 0.001$ , then the coefficient of  $u_{i,j}$  in (3.5) is  $4/(.001)^2 = 4 \times 10^6$ , while the coefficients from (3.6) are equal



**Figure 3.6.** Ordering of unknowns on a  $m_x \times m_y = 4 \times 3$  grid.

to 1. To make the equations better scaled, we multiply (3.5) by the grid cell area  $h_x h_y$  to obtain

$$2(a+b)u_{i,j} - a(u_{i-1,j} + u_{i+1,j}) - b(u_{i,j-1} + u_{i,j+1}) = h_x h_y f_{i,j}, \quad (3.7)$$

where  $a = h_y/h_x$  and  $b = h_x/h_y$ . Using (3.7), all the equations in the system will have coefficients of comparable size, at least so long as the cell aspect ratio  $h_y/h_x$  is neither large nor small. For instance, if  $h_x = h_y$  then diagonal entries are 4 and off-diagonal entries are  $-1$ .

Second, the FD equations can be reinterpreted as giving a *symmetric* matrix. For example, at a grid point adjacent to the left-hand boundary, the boundary value  $u_{0,j}$  appears in the equation. The matrix will be symmetric if we move such values to the right-hand side. This converts the off-diagonal entries of  $A$  to zeros in the columns corresponding to boundary values. This way of generating a symmetric matrix, which can be done even if the boundary values are nonzero, opens up a larger range of methods for solving the system efficiently. For instance, we can use the CG iteration (`-ksp_type cg`) and Cholesky preconditioners (`-pc_type cholesky` or `-pc_type icc`).

With these two modifications the linear system associated to our FD scheme is an  $N \times N$  symmetric, positive-definite (SPD) linear system with  $O(1)$  magnitude entries:

$$A_h \mathbf{u} = \mathbf{b}. \quad (3.8)$$

**Example 3.2.** For the grid in Figure 3.6, linear system (3.8) is

$$\begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & \alpha & \beta \\ & & & & \beta & \alpha \\ & & & & & 1 \\ & & & & & & 1 \\ & & & & & & & 1 \\ & & & & & & & & 1 \end{bmatrix} \begin{bmatrix} u_{0,0} \\ u_{1,0} \\ u_{2,0} \\ u_{3,0} \\ u_{0,1} \\ u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{0,2} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ (1/6)f_{1,1} \\ (1/6)f_{2,1} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

where  $\alpha = 2(h_y/h_x) + 2(h_x/h_y) = 13/3$  and  $\beta = -h_y/h_x = -3/2$ . The matrix is SPD and better scaled than before:  $\kappa(A_h) = 5.83$ .

## Structured-grid matrix assembly

Our FD solver `poisson.c` is now presented in three parts. First, function `formMatrix()` in Code 3.1 assembles  $A_h$  in (3.8) from the rescaled FD equations (3.7) on a grid described by an input `DMDA` object. The other argument to `formMatrix()` is the returned (modified) `Mat`.

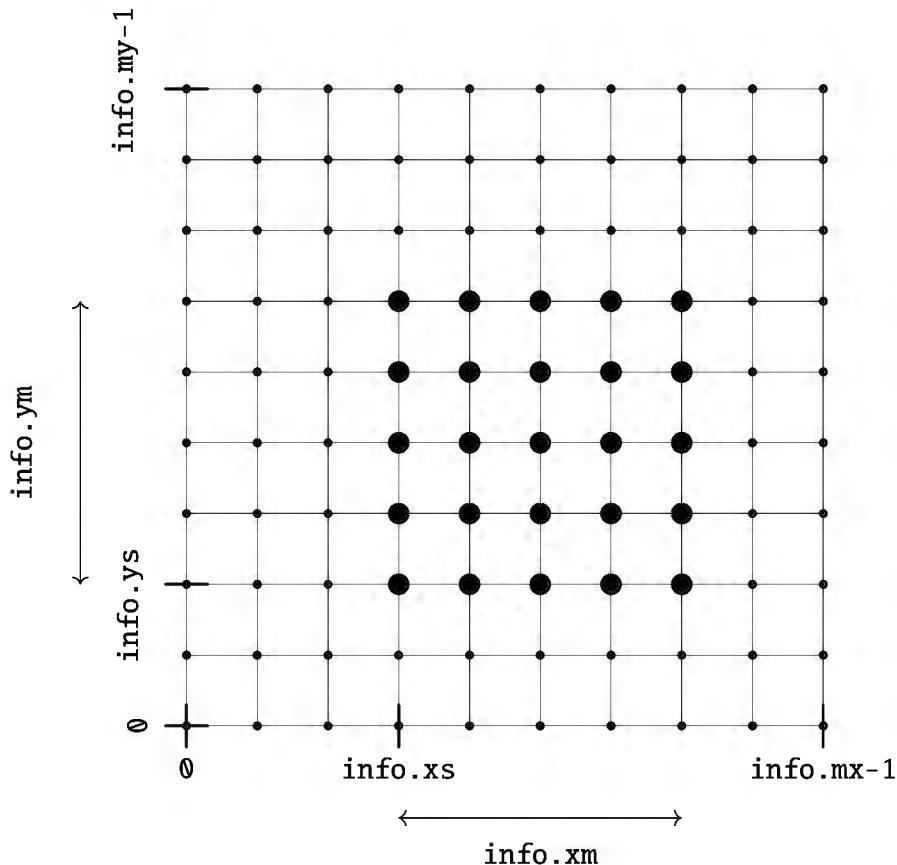
```
PetscErrorCode formMatrix(DM da, Mat A) {
    DMDALocalInfo info;
    MatStencil row, col[5];
    PetscReal hx, hy, v[5];
    PetscInt i, j, ncols;

    DMGetLocalInfo(da,&info);
    hx = 1.0/(info.mx-1); hy = 1.0/(info.my-1);
    for (j = info.ys; j < info.ys+info.ym; j++) {
        for (i = info.xs; i < info.xs+info.xm; i++) {
            row.j = j; // row of A corresponding to (x_i,y_j)
            row.i = i;
            col[0].j = j; // diagonal entry
            col[0].i = i;
            ncols = 1;
            if (i==0 || i==info.mx-1 || j==0 || j==info.my-1) {
                v[0] = 1.0; // on boundary: trivial equation
            } else {
                v[0] = 2*(hy/hx + hx/hy); // interior: build a row
                if (i-1 > 0) {
                    col[ncols].j = j; col[ncols].i = i-1;
                    v[ncols++] = -hy/hx;
                }
                if (i+1 < info.mx-1) {
                    col[ncols].j = j; col[ncols].i = i+1;
                    v[ncols++] = -hx/hx;
                }
                if (j-1 > 0) {
                    col[ncols].j = j-1; col[ncols].i = i;
                    v[ncols++] = -hx/hy;
                }
                if (j+1 < info.my-1) {
                    col[ncols].j = j+1; col[ncols].i = i;
                    v[ncols++] = -hx/hy;
                }
            }
            MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES);
        }
    }
    MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
    return 0;
}
```

**Code 3.1.** *c/ch3/poisson.c, part I. Formulas (3.6) and (3.7) fill matrix  $A_h$ .*

The `DMDALocalInfo` object in `formMatrix()` is a PETSC type, a C structure, which describes both the global grid and locally owned portion (Figure 3.7). The global grid extent is in members `info.mx`, `info.my`. The local process owns a `info.xm`  $\times$  `info.ym` rectangular subgrid:

$$\begin{aligned} \text{info.xs} \leq i \leq \text{info.xs} + \text{info.xm} - 1, \\ \text{info.ys} \leq j \leq \text{info.ys} + \text{info.ym} - 1. \end{aligned}$$



**Figure 3.7.** A DMDALocalInfo struct describes both the global grid sizes and the “local” indices for a given process’s portion (large dots) of a 2D grid.

These ranges appear in for loops when operating on the grid:

```
for (j=info.ys; j<info.ys+info.ym; j++) {
    for (i=info.xs; i<info.xs+info.xm; i++) {
        DO SOMETHING AT GRID POINT (i,j)
    }
}
```

For example, regarding  $x$ -axis indices, the left side of Figure 3.3 shows a grid with  $\text{info.mx} = 5$  where the rank 0 and 2 processes both have  $\text{info.xs} = 0$  and  $\text{info.xm} = 3$ , and the rank 1 and 3 processes both have  $\text{info.xs} = 3$  and  $\text{info.xm} = 2$ .

Considering Code 3.1, note that each rank owns certain ranges of rows of the Mat object  $A$ . However, the DM $\Delta$ A allows us to work with the locally-owned subgrid using  $(i, j)$  indices, ignoring the actual ordering of the equations and unknowns. We need only loop over the portion of the grid owned by the process, and grid indices  $(i, j)$  suffice when inserting entries into Mat  $A$ .

We call `MatSetValuesStencil()` for each locally owned grid point, that is, for each equation in the linear system, to insert coefficients into the matrix. There are five values to insert at interior points, but fewer at the boundary. The key data structure is type `MatStencil`, a simple struct:

```
typedef struct {
    PetscInt k,j,i,c;
} MatStencil;
```

In our 2D case we use only the `i`, `j` members of `MatStencil`, and the matrix entries come from equation (3.7).

Regarding further uses of `MatStencil`, in 3D we would also use index `k`, and an example is in Chapter 6. While Poisson equation (3.1) is a scalar PDE, with only one unknown at a grid point, for a system of PDEs we would set the `dof` argument in `DMDACreateXd()` to a value larger than one, and then the “`c`” member of `MatStencil` would identify a component of the solution vector field. Such examples, with `dof > 1`, appear in Chapters 5 and 7.

## A particular problem

We need a convenient Poisson problem for our example. For this we choose a solution, that is, we *manufacture* a solution [153], taking care that it satisfies homogeneous Dirichlet boundary conditions  $u = 0$  along  $\partial\mathcal{S}$ :

$$u(x, y) = (x^2 - x^4)(y^4 - y^2). \quad (3.9)$$

Then we merely differentiate to get  $f = -\nabla^2 u$ :

$$f(x, y) = 2(1 - 6x^2)y^2(1 - y^2) + 2(1 - 6y^2)x^2(1 - x^2). \quad (3.10)$$

Code 3.2 shows (3.9) implemented as `formExact()`, and (3.10) implemented as `formRHS()`, using only local grid coordinates  $(i, j)$ .

```
PetscErrorCode formExact(DM da, Vec ueexact) {
    PetscInt i, j;
    PetscReal hx, hy, x, y, *auexact;
    DMDALocalInfo info;

    DMGetLocalInfo(da,&info);
    hx = 1.0/(info.mx-1); hy = 1.0/(info.my-1);
    DMDAVecGetArray(da, ueexact, &auexact);
    for (j = info.ys; j < info.ys+info.ym; j++) {
        y = j * hy;
        for (i = info.xs; i < info.xs+info.xm; i++) {
            x = i * hx;
            auexact[j][i] = x*x * (1.0 - x*x) * y*y * (y*y - 1.0);
        }
    }
    DMDAVecRestoreArray(da, ueexact, &auexact);
    return 0;
}

PetscErrorCode formRHS(DM da, Vec b) {
    PetscInt i, j;
    PetscReal hx, hy, x, y, f, **ab;
    DMDALocalInfo info;

    DMGetLocalInfo(da,&info);
    hx = 1.0/(info.mx-1); hy = 1.0/(info.my-1);
    DMDAVecGetArray(da, b, &ab);
    for (j=info.ys; j<info.ys+info.ym; j++) {
        y = j * hy;
        for (i=info.xs; i<info.xs+info.xm; i++) {
            x = i * hx;
            if (i==0 || i==info.mx-1 || j==0 || j==info.my-1) {
                ab[j][i] = 0.0; // on boundary: 1*u = 0
            } else {

```

```

        f = 2.0 * ( (1.0 - 6.0*x*x) * y*y * (1.0 - y*y)
                  + (1.0 - 6.0*y*y) * x*x * (1.0 - x*x) );
        ab[j][i] = hx * hy * f;
    }
}
DMDAVecRestoreArray(da, b, &ab);
return 0;
}

```

**Code 3.2.** *c/ch3/poisson.c, part II. Implementations of formulas (3.9) and (3.10).*

The error term  $O(h^2)$  in equation (3.4) has a coefficient proportional to fourth derivatives [115], and thus our FD method will not be exact on this problem. This is *good* for measuring convergence. For example, we would not want to use the simpler form  $u(x, y) = (x - x^2)(y^2 - y)$ , which has zero fourth derivatives, because then the scheme would be exact and so the generic correct rate of decay of the numerical error, as  $h \rightarrow 0$ , would not be apparent; compare Exercise 3.2.

## Solving the PDE

Code 3.3 shows the `main()` function of `poisson.c`. Here we create the various objects needed to solve the Poisson problem, namely one `DMDA`, one `Mat`, three `Vvecs`, and one `KSP`. Note that the `DMDA` computes matrix and vector sizes from the grid dimensions when we call `DMCreateMatrix()` and `DMCreateGlobalVector()`, respectively, so we do not do this ourselves. (Compare examples in Chapter 2.) We call the functions defined in Codes 3.1 and 3.2, shown earlier, to assemble the matrix and vectors.

```

int main(int argc,char **args) {
    DM          da;
    Mat         A;
    Vec          b,u,uexact;
    KSP         ksp;
    PetscReal    errnorm;
    DMDALocalInfo info;

    PetscInitialize(&argc,&args,(char*)0,help);

    // change default 9x9 size using -da_grid_x M -da_grid_y N
    DMDACreate2d(PETSC_COMM_WORLD,
                  DM_BOUNDARY_NONE, DM_BOUNDARY_NONE, DMDA_STENCIL_STAR,
                  9,9,PETSC_DECIDE,PETSC_DECIDE,1,1,NULL,NULL,&da);

    // create linear system matrix A
    DMSetFromOptions(da);
    DMSetUp(da);
    DMCreateMatrix(da,&A);
    MatSetFromOptions(A);

    // create RHS b, approx solution u, exact solution uexact
    DMCreateGlobalVector(da,&b);
    VecDuplicate(b,&u);
    VecDuplicate(b,&uexact);

    // fill vectors and assemble linear system
    formExact(da,uexact);
    formRHS(da,b);
}

```

```

    formMatrix(da,A);

    // create and solve the linear system
    KSPCreate(PETSC_COMM_WORLD,&ksp);
    KSPSetOperators(ksp,A,A);
    KSPSetFromOptions(ksp);
    KSPSolve(ksp,b,u);

    // report on grid and numerical error
    VecAXPY(u,-1.0,uexact);      // u <- u + (-1.0) uxact
    VecNorm(u,NORM_INFINITY,&errnorm);
    DMDAGetLocalInfo(da,&info);
    PetscPrintf(PETSC_COMM_WORLD,
                "on %d x %d grid: error |u-uexact|_inf = %g\n",
                info.mx,info.my,errnorm);

    VecDestroy(&u);  VecDestroy(&uexact);  VecDestroy(&b);
    MatDestroy(&A);  KSPDestroy(&ksp);   DMDDestroy(&da);
    return PetscFinalize();
}

```

**Code 3.3.** *c/ch3/poisson.c, part III. Function main().*

As in Chapter 2, the linear system is solved by using a KSP linear solver object. Code 3.3 shows how we create the KSP and tell it about Mat A through a call to `KSPSetOperators()`. Recall that there are two ways A can be used, namely as the system matrix and as the “material” from which the preconditioner is built, thus A appears as two arguments of `KSPSetOperators()` (Chapter 2). We then call `KSPSetFromOptions()` so that we may, as illustrated below, change the KSP type at run time. After calling `KSPSolve()` we compute and report the numerical error  $\|u - u_{\text{exact}}\|_\infty$ . Finally we destroy objects and call `PetscFinalize()`.

A first goal might be to see the residual norms decrease:

```
$ ./poisson -ksp_monitor
0 KSP Residual norm 1.020952970432e-01
1 KSP Residual norm 2.656923348626e-02
2 KSP Residual norm 8.679141000397e-03
3 KSP Residual norm 1.557150861763e-03
4 KSP Residual norm 2.239919982542e-04
5 KSP Residual norm 2.519822315367e-05
6 KSP Residual norm 2.152764600588e-06
7 KSP Residual norm 2.650467236964e-07
on 9 x 9 grid: error |u-uexact|_inf = 0.000763959
```

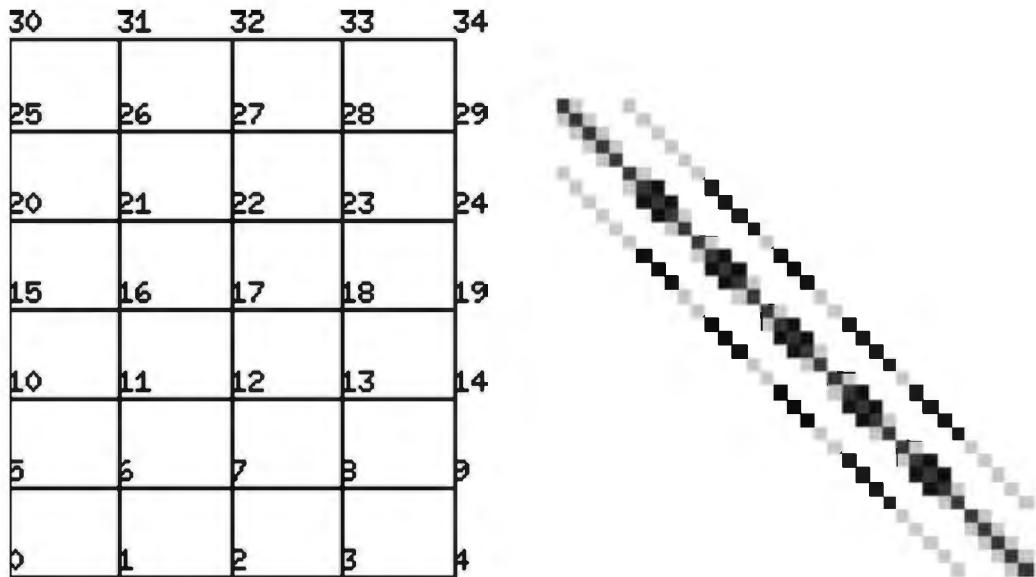
This run sets up a linear system of  $N = 81$  equations and solves it using the default KSP method, namely GMRES+ILU (Chapter 2). This yields a small residual norm after seven iterations, and an apparently small numerical error, so it is reasonable to hope that we have solved the problem. However, further inspection and experimentation are definitely in order.

## Run-time visualization

At run time one may visualize the grid, the assembled Mat, or the solution. For the grid, if the X window system is correctly configured with your PETSC installation then

```
| $ ./poisson -da_grid_x 5 -da_grid_y 7 -dm_view draw -draw_pause 3
```

gives Figure 3.8 (left), the same grid as Figure 3.2, along with the global index.



**Figure 3.8.** PETSc can graphically display a grid (DMDA object; left) or the matrix structure (right) at run time.

As a first check on our matrix assembly, the following view matches the matrix on page 49:

```
| $ ./poisson -da_grid_x 4 -da_grid_y 3 -ksp_view_mat
```

The matrix structure can also be viewed graphically, for example

```
| $ ./poisson -da_grid_x 5 -da_grid_y 7 -ksp_view_mat draw -draw_pause 3
```

See Figure 3.8 (right); note that positive entries, negative entries, and allocated zeros are shown with different colors. As expected, the matrix structure for this Poisson problem is a symmetric sparse matrix with tridiagonal blocks along the diagonal and a banded structure.

To generate a “movie” of the KSP iterates  $u_k$  do

```
| $ ./poisson -da_refine 4 -ksp_monitor -ksp_monitor_solution draw \
    -draw_pause 0.1
```

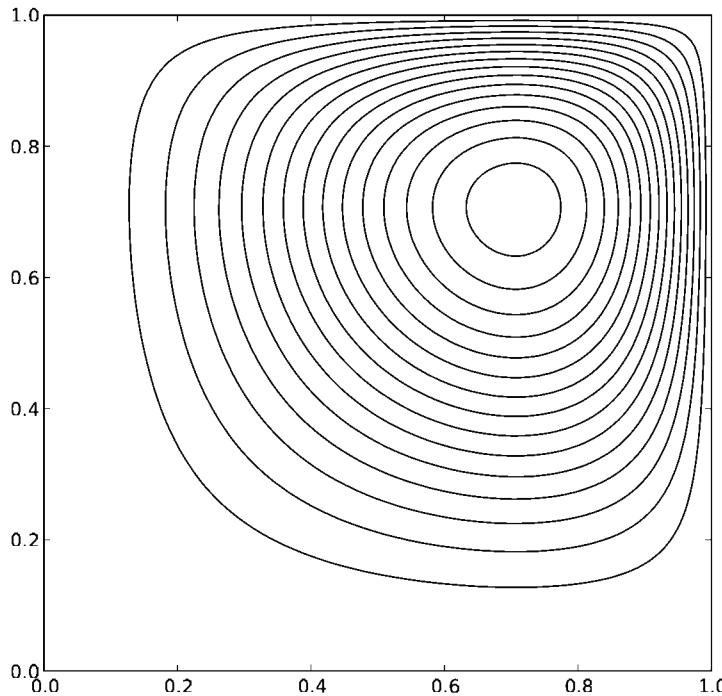
The last frame is shown as a contour map in Figure 3.9. (The figure was actually generated offline using option `-ksp_view_solution binary:u.dat` and a Python script to process `u.dat`.)

We can also visualize KSP convergence. For example, a line graph of the preconditioned and true residual norms comes from `-ksp_monitor_lg_true_residualnorm` (not shown; see Exercise 2.20).

## Convergence in practice and theory

Questions of convergence and performance have been delayed until now, that is, until we can run the code and *see*. We want to know

- is the numerical method correctly implemented? (*convergence*)
- what is going on inside the solver? (*exposure*)
- how do we get the solution quickly? (*performance*)



**Figure 3.9.** Contours of the computed solution  $u(x, y)$  on a  $129 \times 129$  grid.

While the next sections address these questions, our exploration of performance will be quite preliminary; only in Chapters 6, 7, and 8 is this question seriously addressed.

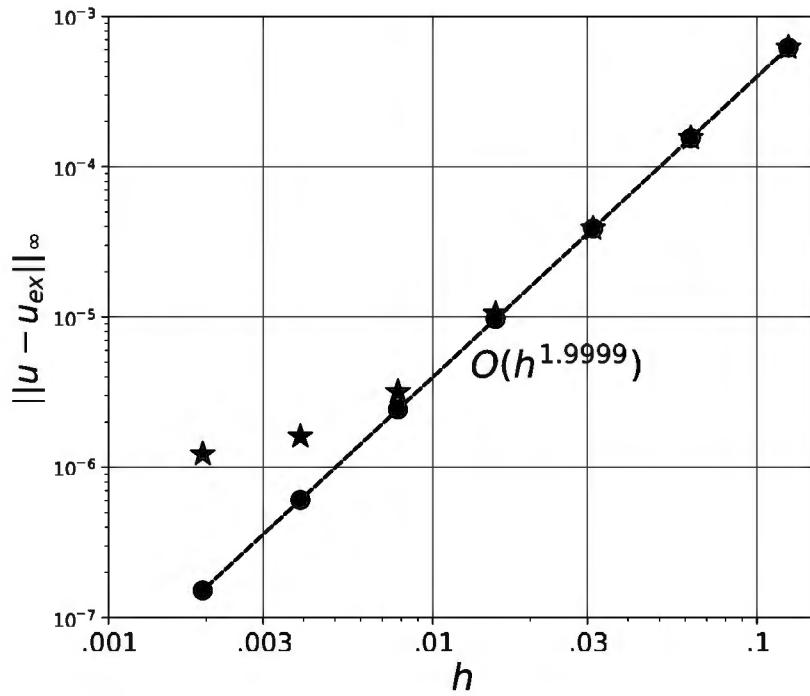
For convergence we want to see that the numerical error, namely a norm of the difference between the computed and exact solutions, decreases as we refine the grid. Such decrease is expected because the finite differences become better approximations of the corresponding derivatives (*consistency* [104]). However, now we want to measure the error to get evidence that our actual implementation is correct. The code is correctly implemented if the error reduction rate matches what we expect from theory.

Recall that `poisson.c` prints the maximum-norm error; see Code 3.3. Here is a refinement study in a one-line Bash loop:

```
$ for K in 0 1 2 3 4 5 6; do ./poisson -da_refine $K; done
on 9 x 9 grid: error |u-uexact|_inf = 0.000763959
on 17 x 17 grid: error |u-uexact|_inf = 0.000196764
on 33 x 33 grid: error |u-uexact|_inf = 4.91557e-05
on 65 x 65 grid: error |u-uexact|_inf = 1.29719e-05
on 129 x 129 grid: error |u-uexact|_inf = 3.76924e-06
on 257 x 257 grid: error |u-uexact|_inf = 1.73086e-06
on 513 x 513 grid: error |u-uexact|_inf = 1.23567e-06
```

This data is shown in Figure 3.10 by stars. For the first four grids, refinement by a factor of two in each dimension reduces the error by a factor of about four, as expected from a  $O(h^2)$  FD scheme.

Unfortunately, the error stops falling for the three finest grids, stagnating around  $10^{-6}$ . This is not from an implementation error, but rather from a default KSP tolerance. We simply need to ask for more accuracy. Rerunning the above loop with a tighter value `-ksp_rtol 1.0e-12`, instead of the default value `1.0e-5`, yields clear convergence at rate  $O(h^2)$  (Figure 3.10).



**Figure 3.10.** To show convergence we refine the DM grid by factors of two (stars), but the errors stagnate. With a stronger linear solver tolerance (-ksp\_rtol 1.0e-12) the errors (dots) decrease at the expected rate.

Backing up slightly, why should we expect  $O(h^2)$  numerical errors from a scheme using FD approximation (3.4), which neglects an  $O(h^2)$  term in the Taylor expansion? The local errors, made when replacing derivatives in the PDE by finite difference quotients, must build up in some manner to yield the global numerical error; why is this still  $O(h^2)$ ?

The well-known theory for FD schemes [104], as follows, directs us to look at quantities which are computable at run time using PETSC, namely eigenvalues and norms of the matrices  $A_h$ . Note that for the rest of this discussion we assume square grid cells  $h = h_x = h_y$ , but the general case is a straightforward extension.

Let  $u(x, y)$  be the exact solution to Poisson problem (3.1), (3.2); the well-posedness of the problem says this function exists and is unique. Let  $\hat{u}_{ij} = u(x_i, y_j)$  denote the gridded values of the exact solution.

By definition, scheme (3.5) has *local truncation error*  $\tau$  satisfying

$$-\frac{\hat{u}_{i-1,j} - 2\hat{u}_{i,j} + \hat{u}_{i+1,j}}{h^2} - \frac{\hat{u}_{i,j-1} - 2\hat{u}_{i,j} + \hat{u}_{i,j+1}}{h^2} = f_{i,j} + \tau_{ij}. \quad (3.11)$$

That is, the local truncation error is the residual from applying the scheme to the exact solution [104, 115]. From (3.4) we know that  $\tau_{ij}$  is  $O(h^2)$  as  $h \rightarrow 0$ .

Now define

$$e_{ij} = u_{ij} - \hat{u}_{ij}, \quad (3.12)$$

the *numerical error* [104]; note that on the boundary,  $e_{ij} = 0$ . Our scheme is *convergent* if these numerical errors  $e_{ij}$  vanish in the limit  $h \rightarrow 0$ .

Subtracting (3.11) from scheme (3.5) gives an equation for the numerical errors,

$$-\frac{e_{i-1,j} - 2e_{i,j} + e_{i+1,j}}{h^2} - \frac{e_{i,j-1} - 2e_{i,j} + e_{i,j+1}}{h^2} = -\tau_{ij}. \quad (3.13)$$

This equation says that values  $e_{ij}$  behave as the solution of a discrete Poisson equation wherein the source term is the truncation error. We may write the error equation (3.13) as a linear system,

$$M_h \mathbf{e} = -\boldsymbol{\tau}, \quad (3.14)$$

where  $M_h = h^{-2} A_h$ . (Recall that we have earlier rescaled  $A_h$  to have  $O(1)$  entries.)

We have arrived at the core reason why FD schemes are effective. If we imagine solving (3.14) for  $\mathbf{e}$ , and take norms, then we see what is needed to prove convergence, and it becomes a definition. By definition, an FD scheme for a linear boundary value problem is *stable* in a given norm if  $M_h$  is invertible and if there is a constant  $C > 0$ , independent of  $h$ , so that

$$\|(M_h)^{-1}\| \leq C. \quad (3.15)$$

Note  $M_h \approx -\nabla^2$  so (3.15) is a uniform-invertibility statement for the approximate Laplacian.

Given stability, convergence will occur at the expected rate:

$$\|\mathbf{e}\| = \|(M_h)^{-1} \boldsymbol{\tau}\| \leq \|(M_h)^{-1}\| \|\boldsymbol{\tau}\| \leq C \|\boldsymbol{\tau}\| = O(h^2). \quad (3.16)$$

This proves the sufficiency of stability in the *Lax-Richtmyer equivalence theorem* [102].

**Theorem 3.3.** *A consistent FD scheme for a linear differential equation is convergent if and only if it is stable.*

A major point now is that the stability of a scheme relates to computable and visualizable quantities. First observe that the 2-norm is a convenient choice because  $M_h$  and  $(M_h)^{-1}$  are symmetric and the 2-norm of a symmetric matrix equals its largest-magnitude eigenvalue [143]. Suppose  $\lambda_h$  denotes an eigenvalue of  $M_h$ . The smallest-magnitude eigenvalue of  $M_h$ , nonzero if and only if  $M_h$  is invertible, supplies the norm in (3.15):

$$\|(M_h)^{-1}\|_2 = \frac{1}{\min |\lambda_h|}. \quad (3.17)$$

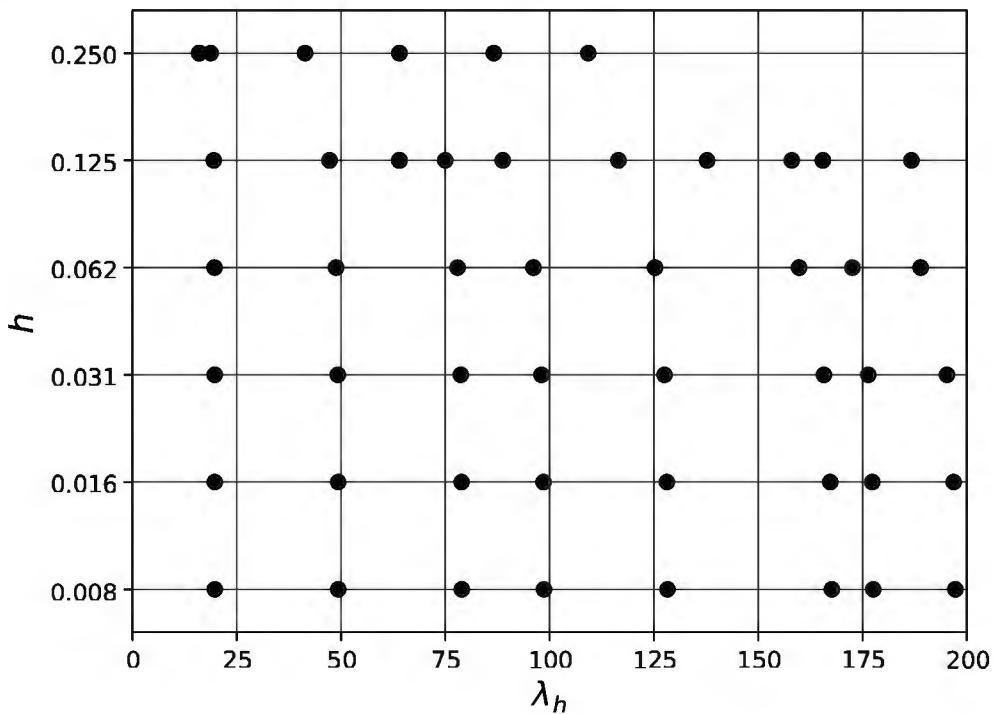
The stability of scheme (3.8) in the 2-norm comes down to whether the smallest-magnitude eigenvalue is bounded below:

$$\text{Is there } \delta > 0 \text{ so that } \min |\lambda_h| \geq \delta \text{ for all small } h > 0? \quad (3.18)$$

Because the matrices  $M_h$  are positive definite, the answer to this question is visualizable by generating matrices and plotting the smaller positive eigenvalues. Figure 3.11 shows the eigenvalues of  $M_h$  for grids with  $m_x = m_y = 5, 9, 17, 33, 65, 129$ ; see Exercise 3.7. The stability of the scheme is reflected by the consistent gap from zero, as  $h \rightarrow 0$ , with  $\min \lambda_h \approx 19.7$  for fine grids. In fact, we see convergence of all small eigenvalues. However, because  $M_h$  approximates the unbounded Laplacian operator [127], the *largest* eigenvalues diverge to infinity.

Historically, the theory of stability of FD schemes took time to develop [115], reflecting another numerical fact of life.

Fact 8. Understanding the theory of convergence and stability for FD schemes requires thinking globally, beyond the local truncation error. *One must consider either the norms or eigenvalues of the family of matrices which are generated as the mesh is refined, and these are global considerations.*



**Figure 3.11.** An FD scheme, like that in `poisson.c`, is stable in 2-norm if the smallest eigenvalue is bounded away from zero. In fact, all small eigenvalues converge to their continuum values as  $h \rightarrow 0$ .

## A first look at performance

The finer-grid calculations above are slow. Why? Are we using the right Krylov method and preconditioner combinations? What will change in parallel?

A first step toward answering these questions, which are fully addressed only in Chapters 6–8, might be to find possible solver options by piping the output from `-help` into a pager like `less` or by grepping for a solver-object prefix:

```
$ ./poisson -help | less
$ ./poisson -help | grep ksp_
```

If the grep result is too long one may pipe it again into `less`.

The latter `-help` usage shows the default values for KSP parameters, such as that the `-ksp_rtol` default is `1e-05`, which explains why convergence “leveled out” on fine grids. One can also grep for prefix `pc_` and thereby list PC types and their options. Also, as in Chapter 2, we may use `-ksp_view` to expose the KSP and PC objects. (Recall that the serial defaults are `KSP = GMRES` and `PC = ILU`, while in parallel they are `GMRES` and `block Jacobi` with each diagonal block preconditioned by `ILU`.) However, merely listing KSP and/or PC types, and associated options, or even inspecting a given solver in detail, does not clarify the relationship between solver choices and performance.

Consider the following run using the serial PETSc defaults:

```
$ ./poisson -da_refine 5 -ksp_converged_reason
Linear solve converged due to CONVERGED_RTOL iterations 506
on 257 x 257 grid: error |u-uexact|_inf = 1.73086e-06
```

The run time, extracted using `-log_view` (Chapter 1), is about 3 seconds. It is not clear if this timing is fast or slow, though we see a concerning number of iterations.

**Table 3.1.** Times and KSP iterations for serial runs of `poisson.c` on a  $257 \times 257$  grid.

KSP	PC	time (s)	iterations
gmres	none	8.44	4705
	ilu	1.35	506
	ilu (+ restart=200)	1.46	174
cg	none	0.52	606
	jacobi	0.57	606
	icc	0.33	177
preonly	icc (+ rtol= $10^{-14}$ )	0.52	314
	cholesky	0.46	1
	none	0.76	579

Experimentation should illuminate better solver choices. Table 3.1 was generated by running

```
$ ./poisson -da_refine 5 -ksp_converged_reason -ksp_rtol 1.0e-10 \
-log_view -ksp_type KSP -pc_type PC
```

For GMRES we see from the table that having a preconditioner is essential. ILU substantially reduces both iteration count and time compared to having no preconditioner. The number of iterations suggests that GMRES(30) went through many restarts, by default every 30 iterations. As memory overflow is no issue in the current problem, we may try to avoid restarts for the PC = ILU case by using option `-ksp_gmres_restart 200`, and the table shows that this reduces the iteration count but not the execution time. (Avoiding restarts this way is not recommended for larger problems, because of memory overflow.)

In this problem the system matrix  $A_h$  is SPD so CG and the Cholesky and ICC (incomplete-Cholesky) preconditioners can be applied (Chapter 2). However, because `-ksp_type preonly -pc_type cholesky` is a direct solver, fair comparison suggests we should solve the equations accurately, which is why we have added a tighter tolerance to all the iterative cases in Table 3.1. We see that Jacobi preconditioning has no benefit over *un*-preconditioned CG, because the diagonal is nearly constant—see Exercise 3.4—while ICC preconditioning for CG is the best so far. The direct Cholesky solver is also fast for this 2D problem.

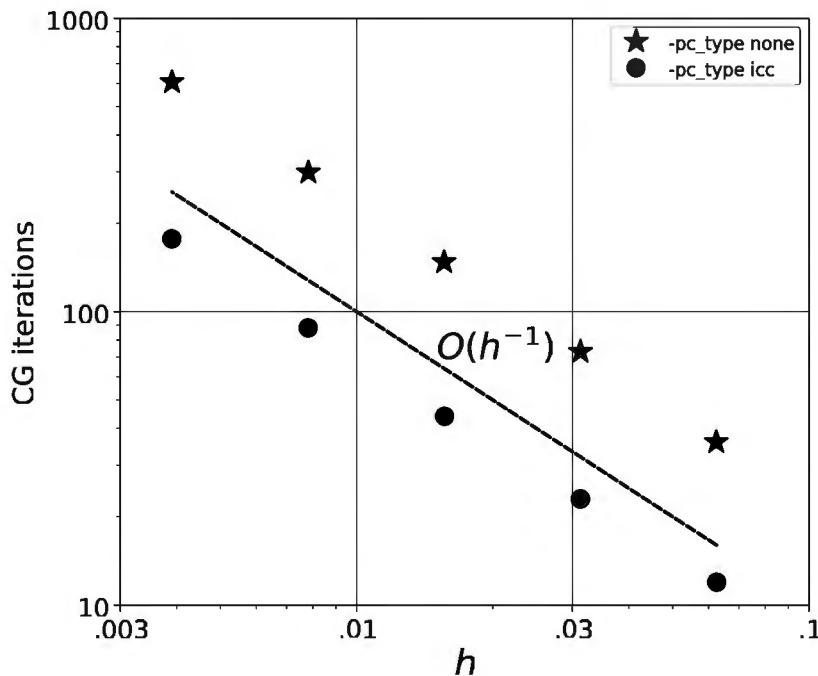
Serial and parallel runs with CG+ICC yield the following results on a grid with  $N \approx 10^6$  points:

```
$ ./poisson -da_refine 7 -ksp_type cg -pc_type icc -ksp_converged_reason
Linear solve converged due to CONVERGED_RTOL iterations 721
on 1025 x 1025 grid: error |u-uxact|_inf = 5.29691e-08
$ mpixexec -n 4 ./poisson -da_refine 7 -ksp_type cg -pc_type bjacobi \
-sub_pc_type icc -ksp_converged_reason
Linear solve converged due to CONVERGED_RTOL iterations 821
on 1025 x 1025 grid: error |u-uxact|_inf = 5.41698e-08
```

As expected, the number of KSP iterations depends on the number of processes. However, are these KSP iteration counts too high anyway? Can we do better?

## Scaling of preconditioned CG iterations

Accepting our best KSP and PC combination so far, namely `cg+icc`, would be premature. In fact, one can demonstrate that all the preconditioned CG solvers in Table 3.1 have the same basic



**Figure 3.12.** As the grid is refined ( $h \rightarrow 0$ ), CG iterations increase on our Poisson problem, even with ICC preconditioning.

flaw. Namely, their iteration counts grow with refinement, so that the number of CG iterations required to meet a fixed tolerance approximately doubles with each refinement  $h \rightarrow h/2$  [49, p. 77].

For example, consider the following Bash loop with unpreconditioned CG:

```
$ for N in 1 2 3 4 5; do ./poisson -da_refine $N -ksp_converged_reason \
    -ksp_type cg -pc_type none; done
```

This generates iteration counts 36, 73, 148, 299, and 606, approximate doubling with each refinement. While `-pc_type jacobi` does no better (Exercise 3.4), Figure 3.12 shows that our favorite method also has the same scaling as the grid is refined and the size of the problem increases. That is, though ICC gives faster solves and lower iteration counts than `none`, the rate at which these counts increase is the same.

A theoretical bound on CG iterations helps us understand. Suppose that  $A$  is SPD so its eigenvalues are positive and equal to its singular values, thus that the 2-norm condition number is the ratio of extreme eigenvalues:  $\kappa_2(A) = \lambda_{\max}/\lambda_{\min}$ . Recall also that in Chapter 2 we defined the  $A$ -norm (2.29), namely  $\|v\|_A = (v^\top Av)^{1/2}$ . Finally, recall that  $e_k = u_k - u$  denotes the error and  $r_k = b - Au_k$  the residual of iterate  $u_k$  as a solution to  $Au = b$ . The following theorem recalls the sense in which CG is norm-minimizing (compare with Table 2.2), and then it supplies an error bound in terms of the condition number.

**Theorem 3.4.** Suppose  $u_j$  are the iterates from CG, and let  $\mathcal{P}_j^1$  be the space of all real polynomials  $p(x)$  of degree at most  $j$  such that  $p(0) = 1$ . Then

$$\|e_j\|_A = \min_{p \in \mathcal{P}_j^1} \|p(A)e_0\|_A.$$

It follows that

- (i) The  $A$ -norm of the error at the  $j$ th iteration is bounded by a function of the 2-norm condition number  $\kappa = \kappa_2(A)$ , namely

$$\frac{\|\mathbf{e}_j\|_A}{\|\mathbf{e}_0\|_A} \leq 2 \left( \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^j.$$

- (ii) The 2-norm of the residual is likewise bounded:

$$\frac{\|\mathbf{r}_j\|_2}{\|\mathbf{r}_0\|_2} \leq 2\sqrt{\kappa} \left( \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^j.$$

The first formula is proved by [66, p. 50]. Part (i) follows by using Chebyshev polynomials to construct a polynomial which is small on the interval  $[\lambda_{\min}, \lambda_{\max}]$  [66, p. 51], and part (ii) follows from (i) by rewriting  $\|\cdot\|_A$  in terms of the 2-norm and the matrix  $\sqrt{A}$  (Exercise 3.6). We state part (ii) because  $\|\mathbf{r}_k\|_2$  is always computable while generally  $\|\mathbf{e}_k\|_A$  is not. Parts (i) and (ii) are not optimal bounds, though such can also be found [66, p. 51].

From the theorem, the number of iterations  $j$  sufficient to reduce  $\|\mathbf{e}_j\|_A$  by a factor of  $\epsilon > 0$  from its initial value  $\|\mathbf{e}_0\|_A$  is given by

$$2 \left( \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^j \leq \epsilon$$

or, equivalently,

$$j \geq \frac{\ln(\epsilon/2)}{\ln\left(1 - \frac{2}{\sqrt{\kappa}+1}\right)} \approx -\frac{1}{2} \ln\left(\frac{\epsilon}{2}\right) (\sqrt{\kappa} + 1)$$

for large  $\kappa$ . (Note  $(x - 1)/(x + 1) = 1 - 2/(x + 1)$  and that  $\ln(1 - x) \approx -x$  for small  $x$ .) This gives a possibly memorable bound:

$$\frac{\|\mathbf{e}_j\|_A}{\|\mathbf{e}_0\|_A} \leq \epsilon \quad \text{if } j = O\left((\ln \epsilon)\sqrt{\kappa_2(A)}\right). \quad (3.19)$$

The same conclusion applies to reducing  $\|\mathbf{r}_j\|_2$  by a factor of  $\epsilon$  (Exercise 3.6).

Theorem 3.4 connects condition number and iteration count. We may ask PETSC to approximate  $\kappa_2(A_h)$  as the Krylov iteration proceeds using option `-ksp_view_singularvalues`, for example as in this un-preconditioned case:

```
$ ./poisson -ksp_type cg -pc_type none -ksp_view_singularvalues
Iteratively computed extreme singular values:
  max 7.69543 min 0.304482 max/min 25.2738
on 9 x 9 grid: error |u-uexact|_inf = 0.00076388
```

Thus, on a  $9 \times 9$  grid with spacing  $h = 1/8$ ,  $\kappa_2(A_{1/8}) \approx 25.3$ . Rerunning with option `-da_refine 1,2` we obtain  $\kappa_2(A_{1/16}) \approx 103.1$  and  $\kappa_2(A_{1/32}) \approx 414.3$ . We now see the issue: apparently  $\kappa_2(A_h) = O(h^2)$  for our FD scheme. Then (3.19) says that, for fixed accuracy goal  $\epsilon$ , we should expect a doubling in iteration count.

The 2-norm condition number of the (un-preconditioned) discrete Poisson matrix  $A_h$  on a uniform rectangular grid with spacing  $h$  is also known in theory to be  $\kappa_2(A_h) = O(h^{-2})$ , as we have just seen experimentally. This can be shown using the eigenvectors of the discrete Poisson problem in the uniform rectangular case [26] or by a finite element method analysis

**Table 3.2.** Condition numbers  $\kappa_2$ , of the symmetrically preconditioned matrix, and iteration counts  $j$ , for `poisson.c` solutions using a CG+ICC solver.

$h$	$\kappa_2$	Ratio	$j$	Ratio
$2^{-3}$	2.982		7	
$2^{-4}$	9.573	3.211	12	1.714
$2^{-5}$	36.35	3.797	23	1.917
$2^{-6}$	147.0	4.045	44	1.913
$2^{-7}$	587.4	3.995	88	2.000
$2^{-8}$	2348	3.998	177	2.011
$2^{-9}$	9391	3.999	357	2.017
$2^{-10}$	37563	4.000	721	2.020

which applies to unstructured grids [49]. Combined with (3.19) this shows that we expect  $j = O(h^{-1})$  iterations to solve our discrete Poisson problem to a given relative tolerance using unpreconditioned CG.

Now, it is true that ICC preconditioning reduces the time and iteration count relative to no preconditioning. However, while detailed analysis is beyond our scope, the asymptotic behavior of the condition number is unchanged:  $\kappa_2(M^{-1}A_h) = O(h^{-2})$  [49, p. 84]. This explains theoretically why ICC does not improve on  $O(h^{-1})$  scaling. On the empirical side, Table 3.2 shows the results of runs

```
$ ./poisson -da_refine X -ksp_type cg -pc_type icc \
-ksp_view_singularvalues -ksp_converged_reason
```

These results confirm that  $\kappa_2 = O(h^{-2})$  and  $j = O(h^{-1})$ ; see Figure 3.12.

## Krylov is not enough: Better preconditioning is needed!

When solving sparse linear systems arising from discretized PDEs, we can hope that preconditioned-Krylov iterations such as CG+ICC are effective solvers. The primary difficulty with such methods has now become clear: the number of iterations can grow as the grid is refined.

However, a second reason to be skeptical of these methods is that on this Poisson problem certain direct linear algebra techniques can also beat CG+ICC. Suppose we compare runs of `poisson` with option `-da_refine 7`, giving a  $1025 \times 1025$  grid and  $N \approx 10^6$  degrees of freedom, a reasonably high resolution 2D solution. Compare the ICC-preconditioned CG method with two direct methods using these runs:

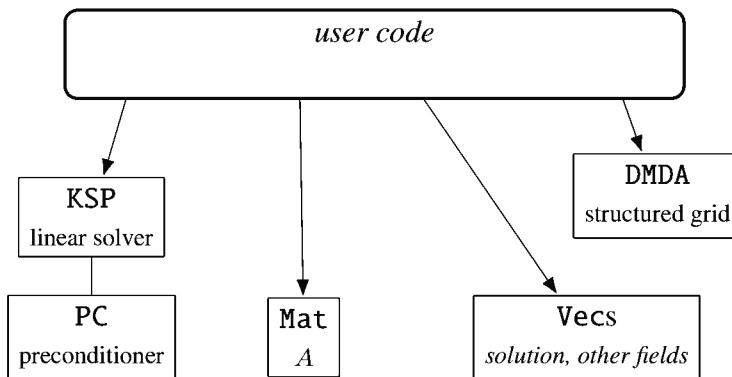
```
$ ./poisson -da_refine 7 -ksp_type cg -pc_type icc \
-ksp_converged_reason -ksp_rtol 1.0e-10
$ ./poisson -da_refine 7 -ksp_type preonly -pc_type cholesky \
-pc_factor_mat_ordering_type nd
$ ./poisson -da_refine 7 -ksp_type preonly -pc_type lu \
-pc_factor_mat_ordering_type nd
```

We obtain the first three rows of Table 3.3.

Option `-pc_factor_mat_ordering_type nd`, which is actually the default for these direct solvers, generates LU and Cholesky factors using the *nested dissection* ordering [58]. While we give no details, the unknowns and equations are reordered so as to reduce the cost of “fill-in” of additional nonzero values in the sparse matrix structure. The two direct solvers in Table 3.3 are faster than ICC-preconditioned CG at this resolution.

**Table 3.3.** Time and iteration count on a  $1025 \times 1025$  grid on a single MPI process on a laptop. The *fish* code is from Chapter 6.

Code	KSP	PC	Time (s)	Iterations
poisson	cg	icc	29.72	1056
	preonly	cholesky + nd	23.82	1
		lu + nd	17.60	1
fish	cg	mg	3.52	8



**Figure 3.13.** An overview of *poisson.c*. Arrows should be read as “user code acts directly on.”

Nonetheless the reader should not give up on Krylov methods yet, though we do need to find better preconditioners if we are going to have scalable solutions of PDE problems. In particular, a geometric multigrid method (Chapter 6), acting as a preconditioner on CG, provides optimal scaling for a structured-grid Poisson problem. Thus we add one more run to Table 3.3:

```
$ cd c/ch6/
$ make fish
$ ./fish -da_refine 9 -pc_type mg -ksp_converged_reason -ksp_rtol 1.0e-10
```

Here *fish.c* is a Chapter 6 program for solving the same Poisson problem, and *-da\_refine 9* gives the same  $1025 \times 1025$  grid.

Switching to multigrid preconditioning yields significant speedup over both the ICC-preconditioned CG and nested-dissection direct methods. The speedup is even larger for further-refined grids because the scaling is greatly improved. (The multigrid method is an order of magnitude faster for  $2049 \times 2049$  and finer 2D grids, and the benefit is even greater for high-resolution 3D grids.) As the reader may check, the condition number of the preconditioned operator, and thus the CG iteration count, is independent of the grid spacing  $h$ .

To exploit the power of geometric multigrid preconditioning (Chapter 6), however, our future codes must do a better job of using PETSC’s tools for assembling the discrete equations. We will use techniques which we want to use anyway, which allow nonlinearity in our PDEs. Thus, as introduced in the next chapter, both nonlinear and linear problems will be solved using a nonlinear solver object.

For now, Figure 3.13 shows an overview of our first PDE-solving PETSC code *poisson.c*. User code directly acts on KSP and DMDA objects and on the Vec and Mat objects which define the linear system. As we develop facility with new PETSC types, this diagram should be compared to similar diagrams in later chapters; compare Figures 4.4, 5.5, and 9.5 in particular.

## Exercises

- 3.1. Use `DMDACreate1d()`, and other appropriate modifications of `poisson.c`, to write a code `poisson1D.c` for the 1D Poisson problem

$$-u'' = f, \quad u(0) = u(1) = 0.$$

For simplicity, choose the exact solution  $u(x) = x^2(1 - x^2)$  to determine  $f(x)$ . Because indexing is easy in 1D, you can either use `MatSetValues()` or `MatSetValuesStencil()`. Show that your results converge at the expected  $O(h^2)$  rate on a sequence of refining grids; compare with Figure 3.10. Find a grid refinement level permitting 11-digit accuracy at every grid point. (*It will never again be this easy to solve an elliptic problem!*)

- 3.2. Modify `poisson.c` to use exact solution  $u(x, y) = (x - x^2)(y^2 - y)$  and the corresponding right-hand side function  $f(x, y)$ . Compute numerical errors as you refine the grid. What rate of convergence do you expect? What do you actually see?
- 3.3. Modify `poisson.c` to allow nonhomogeneous Dirichlet boundary data  $u|_{\partial S} = g$ . That is, solve (3.1) with both  $f$  and  $g$  derived from an exact solution such as  $u(x, y) = 3x + \sin(20xy)$ . Again examine numerical errors under grid refinement; a correct code will show  $O(h^2)$  convergence.
- 3.4. Confirm the nearly identical performance of un-preconditioned and Jacobi-preconditioned CG by doing

```
$ ./poisson -da_refine N -ksp_converged_reason -log_view \
-ksp_type cg -pc_type X
```

for  $N \in \{1, \dots, 6\}$  and  $X \in \{\text{none}, \text{jacobi}\}$ . Explain. Now confirm that ICC preconditioning gives lower iteration counts but the same scaling with refinement.

- 3.5. Add `-log_view` to an un-preconditioned CG run, e.g.,

```
$ ./poisson -ksp_converged_reason -ksp_type cg -pc_type none \
-log_view
```

By looking at the “Count” column, and noting that an iteration count comes from `-ksp_converged_reason` output, confirm that the computational work of one CG iteration consists of one matrix-vector product (`MatMult`), two inner products (`VecTDot`), and three vector updates (`VecAXPY` and `VecAYPX`). Compare Exercise 2.8.

- 3.6. Let  $A$  be an SPD  $N \times N$  matrix. Note  $A$  is diagonalizable and has positive eigenvalues.

(i) Show that (2.29) defines a norm on  $\mathbb{R}^N$ .

(ii) Define the matrix  $\sqrt{A}$  as the unique SPD matrix such that  $(\sqrt{A})^2 = A$ . Show that  $\|v\|_A = \|\sqrt{A}v\|_2$  and that  $\kappa_2(\sqrt{A}) = \sqrt{\kappa_2(A)}$ .

(iii) Suppose  $A\mathbf{e} = -\mathbf{r}$  and show that

$$\frac{1}{\|\sqrt{A}\|_2} \|\mathbf{r}\|_2 \leq \|\mathbf{e}\|_A \leq \|\sqrt{A}\|_2 \|\mathbf{r}\|_2.$$

(iv) Prove part (ii) of Theorem 3.4.

- 3.7. Generate Figure 3.11 using `-ksp_view_eigenvalues`. (*You will need to use the un-preconditioned operator. Remember to scale the computed eigenvalues by  $h^{-2}$ .*)
- 3.8. The minimum residual method (MINRES) [66] applies to symmetric matrices, and so it could be added to Table 3.1. Confirm [49, p. 91]: “when solving discrete Poisson problems the convergence of MINRES is almost identical to that of CG.”

- 3.9. Produce figures similar to Figure 3.12 for GMRES+ILU and MINRES+ICC preconditioning to show that the scaling limitations which apply to CG+ICC also apply to these solvers. (*Only with geometric (Chapter 6) and algebraic (Chapter 10) multigrid preconditioning is this fixed.*)

- 3.10. See what happens when you try

```
| $ ./poisson -da_refine 7 -ksp_type cg -pc_type mg
```

Though there is no error message, and the solution eventually succeeds, the solver is not geometric multigrid at all. See the result of `-ksp_view`. (*The solver view may not make sense. Consider returning to this exercise after reading Chapter 6.*)

- 3.11. If you have difficulty reproducing timings comparable to those in Table 3.3, note that they come from a PETSC configuration using the “optimized” option `--with-debugging=0`. If you have not already done so, see the PETSC installation page

[www.mcs.anl.gov/petsc/documentation/installation.html](http://www.mcs.anl.gov/petsc/documentation/installation.html)

and generate an optimized PETSC configuration with a new PETSC\_ARCH value. Then generate your own version of Table 3.3.

- 3.12. Use `DMDACreate3d()`, and etc., in a code `poisson3D.c` which solves a 3D Poisson problem on the unit cube  $\mathcal{C} = (0, 1)^3$  using the same DMDA and KSP methods as in `poisson.c`. Looking forward, describe how the resulting solver is different from the one in Chapter 6. (*The latter version is recommended.*)

## Chapter 4

# Nonlinear equations by Newton's method

Compared to linear equations, nonlinear equations merely change the functional form of the residual. For a linear system  $A\mathbf{x} = \mathbf{b}$  the residual is the linear function  $\mathbf{r}(\mathbf{x}) = \mathbf{b} - A\mathbf{x}$ , but now we consider cases where the residual is a more general function. That is, suppose  $\mathbf{F} : \mathbb{R}^N \rightarrow \mathbb{R}^N$  is differentiable. Because the input  $\mathbf{x}$  and output  $\mathbf{F}(\mathbf{x})$  are column vectors,  $\mathbf{F}$  has the same domain and range spaces as multiplication by a square matrix  $A$ , i.e.,  $\mathbf{F}(\mathbf{x})$  and  $\mathbf{r}(\mathbf{x})$  are analogous. We hope to solve the general nonlinear equation

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}. \quad (4.1)$$

For the equation  $A\mathbf{x} = \mathbf{b}$ , an iterative linear solver would generate a sequence  $\mathbf{x}_k$  which reduces  $\mathbf{r}(\mathbf{x}_k) = \mathbf{b} - A\mathbf{x}_k$  to zero. Similarly, we will solve nonlinear equation (4.1) by iteration, by generating approximations  $\mathbf{x}_k$  so that  $\mathbf{F}(\mathbf{x}_k)$  goes to zero. As explained below, Newton's method does this by linearizing equation (4.1) around an iterate, solving this linear system for a vector which is a “Newton step,” and computing a new iterate, which we hope is closer to the solution, by adding this vector to the current iterate. This is not the only possible plan for solving (4.1)—compare [29]—but this chapter is devoted to it.

What choices arise in using Newton's method in practice? First, both exact and approximate linearizations are allowed. Second, deciding on a distance to move, even if the search direction is fixed to be along the Newton step vector, is a nontrivial choice. Finally, choices made in solving the linear problem for the Newton step, especially Krylov method and preconditioning choices, are critical for efficiency. Nonlinear solvers in PETSc can, and should, exploit all of the linear-system tools from Chapters 2 and 3, and more.

In this book we are led to Newton's method because large systems of nonlinear equations (4.1) arise as the discretizations of nonlinear PDE models. While much of the following material focuses on fixed-dimension nonlinear systems, i.e., systems not necessarily derived from PDEs, in this chapter we do eventually solve a nonlinear boundary value problem in one dimension. In later chapters Newton's method is routinely used to solve nonlinear PDEs in two or three dimensions.

## Newton's method

Suppose  $\mathbf{x}_k \in \mathbb{R}^N$  is an approximation to the solution of equation (4.1). If  $\mathbf{F} : \mathbb{R}^N \rightarrow \mathbb{R}^N$  is differentiable then for  $\mathbf{s} \in \mathbb{R}^N$  we have

$$\mathbf{F}(\mathbf{x}_k + \mathbf{s}) = \mathbf{F}(\mathbf{x}_k) + J_{\mathbf{F}}(\mathbf{x}_k)\mathbf{s} + o(\|\mathbf{s}\|) \quad (4.2)$$

for some (square) matrix-valued function

$$J_{\mathbf{F}}(\mathbf{x}) = \begin{bmatrix} \frac{\partial F_0}{\partial x_0} & \cdots & \frac{\partial F_0}{\partial x_{N-1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_{N-1}}{\partial x_0} & \cdots & \frac{\partial F_{N-1}}{\partial x_{N-1}} \end{bmatrix} \quad (4.3)$$

and some quantity  $o(\|\mathbf{s}\|)$  which goes to zero as  $\|\mathbf{s}\| \rightarrow 0$ . The matrix  $J_{\mathbf{F}}(\mathbf{x})$ , uniquely defined by (4.2), is called the *Jacobian* of  $\mathbf{F}$  at  $\mathbf{x}$ .

Each iteration of Newton's method approximately solves (4.1) by truncating (4.2) to a linear equation for  $\mathbf{s}$  and finding  $\mathbf{s}$  so that the updated value  $\mathbf{F}(\mathbf{x}_{k+1})$  would be zero if that linear equation were exact. That is, we drop the “ $+o(\|\mathbf{s}\|)$ ” term and define  $\mathbf{s}$  as the solution to  $\mathbf{0} = \mathbf{F}(\mathbf{x}_k) + J_{\mathbf{F}}(\mathbf{x}_k)\mathbf{s}$ . If  $\mathbf{s}$  is the step to take from the current iterate  $\mathbf{x}_k$  then  $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}$  is the next iterate. Thus each iteration of Newton's method requires solving a linear system and doing a vector addition:

$$J_{\mathbf{F}}(\mathbf{x}_k)\mathbf{s} = -\mathbf{F}(\mathbf{x}_k), \quad (4.4a)$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}. \quad (4.4b)$$

**Example 4.1.** Small nonlinear systems are visualized as finding the intersections of curves, surfaces, or hypersurfaces, depending on dimension. For example, given a real parameter  $b$ , the nonlinear equations  $y = \frac{1}{b}e^{bx}$  and  $x^2 + y^2 = 1$  may be intersecting curves in the plane. If  $b \geq 1$ , for instance, it is clear that the curves intersect exactly twice. Put in form (4.1), the nonlinear residual function is

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} \frac{1}{b}e^{bx_0} - x_1 \\ x_0^2 + x_1^2 - 1 \end{bmatrix}, \quad (4.5)$$

and thus the Jacobian is

$$J_{\mathbf{F}}(\mathbf{x}) = \begin{bmatrix} e^{bx_0} & -1 \\ 2x_0 & 2x_1 \end{bmatrix}. \quad (4.6)$$

As shown in Figure 4.1 for the  $b = 2$  case, if we start the Newton iteration with  $\mathbf{x}_0 = [1, 1]^\top$  then the sequence of iterates from (4.4) is

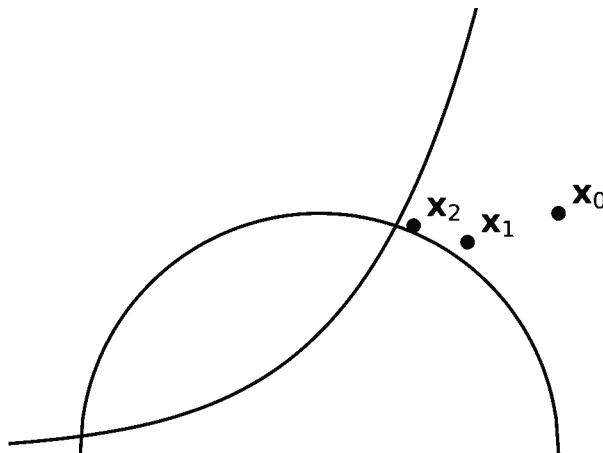
$$\mathbf{x}_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \mathbf{x}_1 = \begin{bmatrix} 0.619203 \\ 0.880797 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 0.394157 \\ 0.948623 \end{bmatrix}, \quad \mathbf{x}_3 = \begin{bmatrix} 0.325199 \\ 0.948157 \end{bmatrix}, \dots$$

At least visually,  $\mathbf{x}_k$  are approaching a solution of  $\mathbf{F}(\mathbf{x}) = 0$ .

Thus Newton's method is simple in theory. Actual practice is also not that complicated with PETSC in hand. In fact, despite the perception that it is fragile or scary, Newton's method works well on many nonlinear systems if one has a good initial iterate  $\mathbf{x}_0$  and one adds some important protections [89]. A *line search*, which sometimes moves a shorter distance than computed in (4.4b), is such a “protection”; see the discussion starting page 90.

## Inside the first SNES code

We will compute the Newton iterates in the above example by using a PETSC object of type SNES, a scalable nonlinear equation solver. Our code provides  $\mathbf{F}$  to the SNES as a *call-back*. That is, the SNES calls the function we provide when it needs a value  $\mathbf{F}(\mathbf{x})$  during the Newton iteration. We may also provide a call-back function which evaluates  $J_{\mathbf{F}}$ , but, because this Jacobian can be



**Figure 4.1.** Newton iterates  $\mathbf{x}_k$  approach a solution of  $\mathbf{F}(\mathbf{x}) = 0$ .

approximated by repeated  $\mathbf{F}$  evaluations using finite differences, as explained below, our first code succeeds without it.

The whole of `expcircle.c`, which solves Example 4.1, is in Code 4.1. The `main()` function starts by allocating a `Vec`  $\mathbf{x}$  of fixed dimension 2 to hold the initial iterate; when the Newton iteration terminates this `Vec` will instead hold the converged solution. A duplicate `Vec`  $\mathbf{r}$  is also needed so that the SNES has space to store the current residual. Note we apply the usual `Create/SetFromOptions/Destroy` sequence to the SNES, and the call to `SNESSetFromOptions()` permits run-time control both on how the Jacobian is calculated and on how the length of the step is determined. After (4.1) is solved by a call to `SNESSolve()` the values in  $\mathbf{x}$ , presumably the converged solution, are printed using `VecView()`.

```

static char help[] = "Newton's method for a two-variable system.\n"
"No analytical Jacobian. Run with -snes_fd or -snes_mf.\n\n";

#include <petsc.h>

extern PetscErrorCode FormFunction(SNES, Vec, Vec, void *);

int main(int argc, char **argv) {
    SNES snes;           // nonlinear solver
    Vec x, r;            // solution, residual vectors

    PetscInitialize(&argc,&argv,NULL,help);
    VecCreate(PETSC_COMM_WORLD,&x);
    VecSetSizes(x,PETSC_DECIDE,2);
    VecSetFromOptions(x);
    VecSet(x,1.0);       // initial iterate
    VecDuplicate(x,&r);

    SNESCreate(PETSC_COMM_WORLD,&snes);
    SNESSetFunction(snes,r,FormFunction,NULL);
    SNESSetFromOptions(snes);
    SNESSolve(snes,NULL,x);
    VecView(x,PETSC_VIEWER_STDOUT_WORLD);

    SNESDestroy(&snes);  VecDestroy(&x);  VecDestroy(&r);
    return PetscFinalize();
}

```

```
PetscErrorCode FormFunction(SNES snes, Vec x, Vec F, void *ctx) {
    const PetscReal b = 2.0, *ax;
    PetscReal *aF;

    VecGetArrayRead(x,&ax);
    VecGetArray(F,&aF);
    aF[0] = (1.0 / b) * PetscExpReal(b * ax[0]) - ax[1];
    aF[1] = ax[0] * ax[0] + ax[1] * ax[1] - 1.0;
    VecRestoreArrayRead(x,&ax);
    VecRestoreArray(F,&aF);
    return 0;
}
```

**Code 4.1.** *c/ch4/expcircle.c*. A SNES code which solves (4.1) for  $\mathbf{F}$  in (4.5).

Formula (4.5) is implemented in method `FormFunction()` and supplied to the SNES using `SNESSetFunction()`. In order to match how SNES calls it, `FormFunction()` must have the signature

```
PetscErrorCode FormFunction(SNES snes, Vec x, Vec F, void *ctx)
```

In particular, `FormFunction()` takes  $\mathbf{x}$  as the first `Vec` argument and it generates output  $\mathbf{F}(\mathbf{x})$  as the second `Vec`. (A `Vec` is actually a *pointer*, so passing a `Vec` by value allows the object to be modified.) There may be additional information, such as parameters, passed to `FormFunction()` in an “application context,” justifying the fourth `void*` argument. (We show how to pass parameters in the next code.)

Looking inside `FormFunction()` in Code 4.1 we see two different ways of accessing the values in a `Vec`. Previously we have used `VecSetValues()` to set values at given indices (Chapter 2), or we have used a DMDA structured grid method of access (Chapter 3). Here we access the C arrays inside the `Vecs`. Because we only need to read the entries of `Vec x`, we use `VecGetArrayRead()` which returns a read-only pointer `const PetscReal *ax`,<sup>16</sup> while for `Vec F` we use `VecGetArray()` because we are setting its entries. Note that `Get` calls are matched by `Restore` calls; the latter “free” the `Vecs` so that other parts of the code may work on them, but they do not deallocate as would `VecDestroy()`.

It is time to run this example:

```
$ cd c/ch4/
$ make expcircle
$ ./expcircle -snes_fd -snes_monitor
0 SNES Function norm 2.874105323289e+00
1 SNES Function norm 8.591393113962e-01
2 SNES Function norm 1.609958353862e-01
3 SNES Function norm 1.106891696425e-02
4 SNES Function norm 6.618141730691e-05
5 SNES Function norm 2.420782802130e-09
Vec Object: 1 MPI processes
  type: seq
0.319632
0.947542
```

Option `-snes_monitor` counts iterations and shows residual norms  $\|\mathbf{F}(\mathbf{x}_k)\|_2$ . After five iterations the Newton method has reduced the residual norm by a factor of  $10^9$ —the default for `-snes_rtol` is  $10^{-8}$ —so the iteration stops with solution  $\mathbf{x}_5 = [0.319632, 0.947542]^\top$ .

<sup>16</sup>Because of the `const` qualifier, the C compiler will stop us from altering `ax[0]`.

The above run also uses option `-snes_fd`, the purpose of which the reader may already see. Clearly the Newton iteration (4.4) requires the Jacobian, but we have only supplied a function  $\mathbf{F}(\mathbf{x})$ . The entries of the Jacobian are, however, derivatives which can be approximated by finite differences, as follows. Let  $\mathbf{u}_j \in \mathbb{R}^N$  denote the standard unit vector with entry one in the  $j$ th position and zeros otherwise. If  $\delta \neq 0$  then an entry in matrix  $J_{\mathbf{F}}(\mathbf{x})$  is approximated:

$$J_{ij} = \frac{\partial F_i}{\partial x_j} \approx \frac{F_i(\mathbf{x} + \delta \mathbf{u}_j) - F_i(\mathbf{x})}{\delta}. \quad (4.7)$$

When using `-snes_fd`, PETSC chooses  $\delta$  internally and applies (4.7). In this example the choice amounts to  $\delta = \sqrt{\epsilon}$ , where  $\epsilon$  is machine precision, giving an approximation accurate to  $\sqrt{\epsilon}$  if the solution values  $\mathbf{x}$  are of order one and if the function  $\mathbf{F}$  can be accurately evaluated [89].

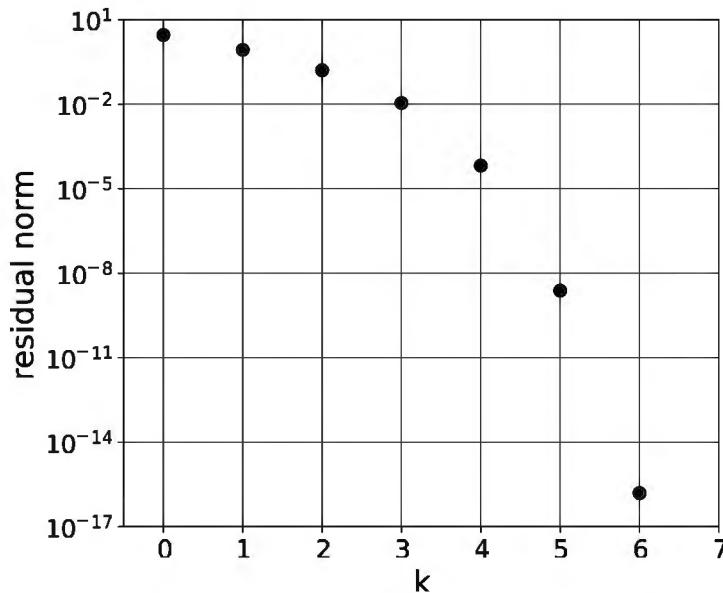
Because there are several variations for the user to consider, it helps to describe the Newton iteration from the point of view of the actions taken by the SNES object. Assuming we are using the default `-snes_type newtonls`, namely a Newton solver with a line search (page 90), SNES does these steps:

- (i) From the current iterate  $\mathbf{x}_k$ ,  $\mathbf{F}(\mathbf{x}_k)$  is evaluated using the call-back function set in `SNESSetFunction()`, e.g., `FormFunction()`,
- (ii) The Jacobian  $J_{\mathbf{F}}(\mathbf{x}_k)$  is evaluated by one of the following methods:
  - a. computed and assembled by a call-back to user-supplied code set using `SNESSetJacobian()`, e.g., `FormJacobian()` in code `ecjac.c` below, or
  - b. computed and assembled by evaluating  $\mathbf{F}(\mathbf{x}_k + \delta \mathbf{u}_j)$  for  $j = 0, \dots, N-1$ , thus calling `FormFunction()`  $N$  times, and then using formula (4.7) up to  $N^2$  times to compute all entries of  $J_{\mathbf{F}}(\mathbf{x}_k)$ , or
  - c. computed and assembled by calling `FormFunction()` substantially fewer than  $N$  times to compute  $\mathbf{F}(\mathbf{x}_k + \delta \mathbf{v})$  for special vectors  $\mathbf{v}$ , by using a graph-coloring algorithm based on the Jacobian sparsity pattern to construct the vectors  $\mathbf{v}$ , and using formula (4.17) below, or
  - d. not assembled, but, in a Krylov iterative method for solving system (4.4a), the action of the Jacobian on vectors (i.e.,  $J_{\mathbf{F}}(\mathbf{x}_k)\mathbf{y}$ ) is computed by finite differences,
- (iii) Linear system (4.4a) is solved for the search direction  $\mathbf{s}$  using the KSP and PC objects chosen by the user,
- (iv) a step length  $\lambda_k$  is determined through line search (page 90),
- (v) the vector update  $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{s}$  is done, and
- (vi) convergence tests are applied, with repeat at (i) if not converged.

Our above run of `expcircle.c` used alternative (ii)b for the Jacobian, namely option `-snes_fd`. Option `-snes_mf`, for method (ii)d, also works. Our next code will allow (ii)a as well. The graph-coloring technique (ii)c, corresponding to option `-snes_fd_color`, requires additional information: either a structured grid or other choice of a graph-coloring object. This technique will be addressed starting on page 83.

Running `expcircle.c` with no options gives an error message. This error is opaque unless you are conscious of the need to form the Jacobian matrix at each Newton iteration. That is, something must supply a Jacobian at step (ii).

The benefit of using Jacobian alternatives (ii)b–(ii)d is that we do not initially need to write any error-prone code based on by-hand differentiation of  $\mathbf{F}$ . Avoiding a Jacobian implementation



**Figure 4.2.** Quadratic convergence in a Newton iteration.

in this way usually shortens the time to create a correct solver. In many cases using an approximate Jacobian in the Newton iteration is effective [89]. On the other hand, a performance problem arises from using (4.7) naively for PDE-type applications of Newton's method. In steps (i) and (ii)b we do  $N+1$  calls to `FormFunction()` per Newton iteration. This is a worrying amount of work if  $N$  is large, as it would be when discretizing a PDE. We will return to this issue later in this chapter, with more detail on alternatives (ii)c and (ii)d in particular.

Evaluating  $\mathbf{F}$  and  $J_{\mathbf{F}}$  can dominate the work in the Newton iteration, especially when these are expensive functions. As a first step to diagnose this, option `-log_view` will show how many times  $\mathbf{F}$  and  $J_{\mathbf{F}}$  are evaluated, as `SNESFunctionEval` and `SNESJacobianEval` counts, respectively; use the search method `-log_view | grep Eval`. The other concern is the work done in solving linear system (4.4a). For assessing the relative cost of function evaluations versus linear solves, compare the `KSPSolve` and `SNESSolve` events in `-log_view` output. If the `KSPSolve` time is the majority of the `SNESSolve` time, as is common, then the linear solves dominate. Otherwise the cost of  $\mathbf{F}$  or  $J_{\mathbf{F}}$  evaluations may be a concern. In any case, a good habit is to profile using `-log_view` to see which kind of work actually dominates.

A further good habit is to use a `--with-debugging=1` PETSC configuration while developing your code, but also to maintain an “optimized” configuration (`--with-debugging=0`) for use in evaluating performance, e.g., when looking at timing results from `-log_view`.

## Convergence of the Newton iteration

Now we return to running `expcircle.c`. Option `-snes_rtol`, with default value `1.0e-8`, specifies by what factor the SNES should reduce the residual norm. The following run asks for much more accuracy, but `-snes_monitor` shows only one more iteration:

```
| $ ./expcircle -snes_fd -snes_monitor -snes_rtol 1.0e-15
```

It may be come as a surprise that asking for a further  $10^7$  reduction in residual norm requires only one more iteration than before, but this is the hoped-for behavior of a Newton iteration.

Figure 4.2 shows the residual norms from the above run in a graph with log scaling on the  $y$  axis. The residual norm drops abruptly, reflecting success in a Newton iteration. The per-step

residual norm decrease becomes huge as the iteration converges because, in exact arithmetic, the numerical error is proportional to the *square* of the numerical error at the previous iteration. If  $\mathbf{x}^*$  denotes the solution of (4.1) to which the iterates  $\mathbf{x}_k$  are converging, i.e., if  $\mathbf{F}(\mathbf{x}^*) = \mathbf{0}$ , then the *numerical error* at iteration  $k$  is

$$\mathbf{e}_k = \mathbf{x}_k - \mathbf{x}^*. \quad (4.8)$$

This is the quantity which we want to send to zero, but  $\mathbf{e}_k$  is just as unknown as  $\mathbf{x}^*$ ; we generally do not have exact access to either. On the other hand, we compute the residuals  $\mathbf{F}(\mathbf{x}_k)$  as part of the Newton iteration. Thus both parts of the following theorem, which assumes exact arithmetic, are important.

**Theorem 4.2.** (See [89].) Suppose that  $\mathbf{F} : \mathbb{R}^N \rightarrow \mathbb{R}^N$  is differentiable,  $J_{\mathbf{F}}$  is Lipschitz near  $\mathbf{x}^*$ , and  $J_{\mathbf{F}}(\mathbf{x}^*)$  is nonsingular. Let  $\mathbf{x}_k$  be the iterates from Newton's method, and suppose  $\|\cdot\|$  is any vector norm.

- (i) If  $\mathbf{x}_0$  is sufficiently close to  $\mathbf{x}^*$  then there is  $C \geq 0$  such that, for all sufficiently large  $k$ ,

$$\|\mathbf{e}_{k+1}\| \leq C\|\mathbf{e}_k\|^2. \quad (4.9)$$

- (ii) If  $\|\cdot\|$  also denotes the induced matrix norm, and if  $\kappa(A) = \|A^{-1}\|\|A\|$  is the corresponding condition number, then

$$\frac{1}{4\kappa(J_{\mathbf{F}}(\mathbf{x}^*))} \frac{\|\mathbf{e}_k\|}{\|\mathbf{e}_0\|} \leq \frac{\|\mathbf{F}(\mathbf{x}_k)\|}{\|\mathbf{F}(\mathbf{x}_0)\|} \leq 4\kappa(J_{\mathbf{F}}(\mathbf{x}^*)) \frac{\|\mathbf{e}_k\|}{\|\mathbf{e}_0\|}. \quad (4.10)$$

By definition, a sequence  $\{\mathbf{x}_k\}$  in  $\mathbb{R}^N$  converges quadratically to  $\mathbf{x}^*$  if the sequence of errors  $\{\mathbf{e}_k\}$  goes to zero and satisfies (4.9) for some  $C \geq 0$ . Heuristically, once  $\|\mathbf{e}_k\|$  gets reasonably small then the number of correct digits in  $\mathbf{x}_k$  doubles with each additional iteration. Theorem 4.2 says that Newton's method converges quadratically under certain assumptions about the initial iterate and the regularity and nonsingularity of the Jacobian.

We seem to see quadratic convergence in Figure 4.2, but it actually shows the residual norm  $\|\mathbf{F}(\mathbf{x}_k)\|_2$  and not the error norm  $\|\mathbf{e}_k\|_2$ . The second part of the theorem reassures us that the residual norm reduction  $\|\mathbf{F}(\mathbf{x}_k)\|/\|\mathbf{F}(\mathbf{x}_0)\|$  is within a factor, determined by the conditioning of the Jacobian at the solution, of the error reduction  $\|\mathbf{e}_k\|/\|\mathbf{e}_0\|$ . If we want to reduce the generally un-knowable error  $\mathbf{e}_k$  by a given amount then it suffices to reduce the residual norm by a comparable amount as long as the Jacobians are well conditioned. For poorly conditioned Jacobians, note that precision is lost in solving (4.1) by any numerical means (Chapter 2).

The residual norm reduction is exactly what `-snes_rtol` controls, that is, the iteration terminates once

$$\frac{\|\mathbf{F}(\mathbf{x}_k)\|_2}{\|\mathbf{F}(\mathbf{x}_0)\|_2} \leq x \quad \text{if } -\text{snes_rtol } x.$$

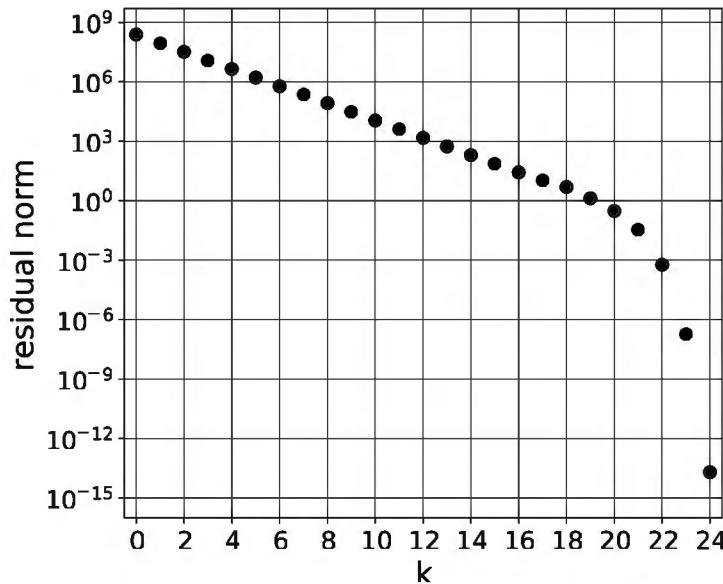
To give a more complete story, however, three SNES tolerances are listed in Table 4.1. The iteration stops as soon as one of these conditions is satisfied, and option `-snes_converged_reason` reports which. For example, the `-snes_rtol` condition stopped this iteration:

```
$ ./expcircle -snes_fd -snes_converged_reason
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 5
...
```

The defaults for the three tolerances in the table are  $x = 10^{-8}, 10^{-50}, 10^{-8}$ , respectively. One can force the SNES to not use a particular stopping criterion by setting the corresponding tolerance to zero.

**Table 4.1.** Three ways SNES can succeed and stop the Newton iteration.

Option	Name	Stopping condition
-snes_rtol X	relative (FNORM_RELATIVE)	$\ F(x_k)\ _2 \leq X \ F(x_0)\ _2$
-snes_atol X	absolute (FNORM_ABS)	$\ F(x_k)\ _2 \leq X$
-snes_stol X	step-length (SNORM_RELATIVE)	$\ s_k\ _2 \leq X \ x_k\ _2$

**Figure 4.3.** Even for well-behaved systems  $F(\mathbf{x}) = \mathbf{0}$ , quadratic convergence can be postponed for many iterations.

We have portrayed the Newton iteration in optimistic terms, but it is not magic and things can go wrong. Note a key hypothesis in the above theorem, namely that  $\mathbf{x}_0$  needs to be sufficiently close to  $\mathbf{x}^*$ . Even on well-behaved nonlinear equations, if  $\mathbf{x}_0$  is far from the solution then the iteration may take many steps before  $\|\mathbf{e}_k\|$  becomes small enough so that quadratic convergence (4.9) “kicks in.” For example, Figure 4.3 shows what happens if we use initial iterate  $\mathbf{x}_0 = [10 \ 10]^\top$  in `expcircle.c`. About 18 iterations of slow progress are needed before the iterate  $\mathbf{x}_k$  enters the region around  $\mathbf{x}^*$  where Theorem 4.2 applies. Furthermore, many real-world problems do not have the smoothness needed to apply Theorem 4.2. In such cases regularization (Chapter 9), continuation [89], grid sequencing (Chapter 7), or other procedures may be needed to make the Newton method effective.

Decrease in residual norm, as displayed in Figures 4.2 and 4.3, is also not guaranteed in general. Indeed, there is nothing intrinsic about problem (4.4) that implies  $\|F(\mathbf{x}_{k+1})\| \leq \|F(\mathbf{x}_k)\|$ . In some cases the Newton iteration actually diverges; Exercise 4.4 gives an example. Line-search methods (page 90), however, enforce residual norm decrease or stop if it cannot be achieved. Such methods are said to “globalize” the convergence [89] by allowing convergence from a larger set of initial states.

## User-supplied Jacobians

We have yet to exploit two important aspects of using SNES objects, namely passing parameters through the call-back mechanism, so that they can be used inside residual- and Jacobian-evaluation functions, and providing a user-written Jacobian function  $J_F(\mathbf{x})$ . Our next code `ecjac.c` does both.

To begin we declare a C struct for the “application context”:

```
typedef struct {
    PetscReal b;
} AppCtx;
```

A struct is not really necessary here, but in future examples there will be multiple parameters. The parameter  $b$ , which appears in formulas (4.5) and (4.6), is the single member of AppCtx.

The method FormFunction() is almost the same as in expcircle.c, but the value of  $b$  now comes from the struct. As shown in Code 4.2, the argument void \*ctx is “cast” in the C language sense [90] to a pointer of type AppCtx\*. Then the parameter is extracted by dereferencing user->b.<sup>17</sup> Note that the signature of FormFunction() matches PETSc type SNESFunction.

```
PetscErrorCode FormFunction(SNES snes, Vec x, Vec F, void *ctx) {
    AppCtx *user = (AppCtx*) ctx;
    const PetscReal b = user->b, *ax;
    PetscReal *aF;

    VecGetArrayRead(x,&ax);
    VecGetArray(F,&aF);
    aF[0] = (1.0 / b) * PetscExpReal(b * ax[0]) - ax[1];
    aF[1] = ax[0] * ax[0] + ax[1] * ax[1] - 1.0;
    VecRestoreArrayRead(x,&ax);
    VecRestoreArray(F,&aF);
    return 0;
}

PetscErrorCode FormJacobian(SNES snes, Vec x, Mat J, Mat P, void *ctx) {
    AppCtx *user = (AppCtx*) ctx;
    const PetscReal b = user->b, *ax;
    PetscReal v[4];
    PetscInt row[2] = {0,1}, col[2] = {0,1};

    VecGetArrayRead(x,&ax);
    v[0] = PetscExpReal(b * ax[0]); v[1] = -1.0;
    v[2] = 2.0 * ax[0]; v[3] = 2.0 * ax[1];
    VecRestoreArrayRead(x,&ax);
    MatSetValues(P,2,row,2,col,v,INSERT_VALUES);
    MatAssemblyBegin(P,MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(P,MAT_FINAL_ASSEMBLY);
    if (J != P) {
        MatAssemblyBegin(J,MAT_FINAL_ASSEMBLY);
        MatAssemblyEnd(J,MAT_FINAL_ASSEMBLY);
    }
    return 0;
}
```

**Code 4.2.** *c/ch4/ecjac.c, part I.* This code solves the same nonlinear system as expcircle.c, but now with an exact Jacobian and parameter-passing.

Function FormJacobian() in Code 4.2 has similar structure to FormFunction(), but it has a new signature matching PETSc type SNESJacobianFunction:

```
PetscErrorCode FormJacobian(SNES snes, Vec x, Mat J, Mat P, void *ctx)
```

Input Vec x and pointer void \*ctx have the same meaning as in FormFunction(), but now there are two output Mats which FormJacobian() should set. The first, called J here,

<sup>17</sup>An equivalent expression is (\*user).b.

corresponds to the Jacobian matrix itself. The second,  $P$ , is the “material” from which PETSc can build a preconditioner. In our case `FormJacobian()` sets entries  $P$  to the correct derivatives of  $F$ , namely (4.6), and we assemble  $P$ . Because we have set the Mats to the same object when calling `SNESSetJacobian()` (below),  $J$  is also now assembled.

In some cases, because of user options,  $J$  is a different `Mat` at the time of the call-back, so it is also assembled. This `Mat` might be a piece of code, such as a user-defined function which defines an operator which approximates the Jacobian. An important case, described on page 86, is when we are using a preconditioned Jacobian-free Newton-Krylov method using option `-snes_mf_operator`.

Regarding the inside of `FormJacobian()`, the roles of real array  $v[4]$  for the entries of the `Mat`, and integer arrays  $row[2]$  and  $col[2]$  as global indices, are the same as they were in Chapter 2. `MatSetValues()`, is also used the same way. Note that for input  $x$  we use `VecGetArrayRead()` and `VecRestoreArrayRead()`.

In `main()` (Code 4.3) we create and configure a  $2 \times 2$  `Mat`  $J$  to hold the Jacobian, and `Create/SetSizes/SetFromOptions/SetUp` mimics what we did for linear systems in Chapter 2. However, this time we pass `Mat`  $J$  to the SNES in two arguments to provide this allocated `Mat` as both the Jacobian and preconditioner-material matrices:

```
SNESSetJacobian(snes,J,J,FormJacobian,&user);
```

We are telling PETSc that we have a Jacobian function and that we want the preconditioner for linear system (4.4a) to be built from it too.

```
int main(int argc,char **argv) {
  SNES  snes;           // nonlinear solver
  Vec   x,r;            // solution, residual vectors
  Mat   J;
  AppCtx user;

  PetscInitialize(&argc,&argv,NULL,help);
  user.b = 2.0;

  VecCreate(PETSC_COMM_WORLD,&x);
  VecSetSizes(x,PETSC_DECIDE,2);
  VecSetFromOptions(x);
  VecDuplicate(x,&r);

  MatCreate(PETSC_COMM_WORLD,&J);
  MatSetSizes(J,PETSC_DECIDE,PETSC_DECIDE,2,2);
  MatSetFromOptions(J);
  MatSetUp(J);

  SNESCreate(PETSC_COMM_WORLD,&snes);
  SNESSetFunction(snes,r,FormFunction,&user);
  SNESSetJacobian(snes,J,J,FormJacobian,&user);
  SNESSetFromOptions(snes);

  VecSet(x,1.0);          // initial iterate
  SNESolve(snes,NULL,x);
  VecView(x,PETSC_VIEWER_STDOUT_WORLD);

  SNESDestroy(&snes);  MatDestroy(&J);  VecDestroy(&x);  VecDestroy(&r);
  return PetscFinalize();
}
```

**Code 4.3.** *c/ch4/ecjac.c, part II. `main()` looks like this when we supply a Jacobian function.*

Now that we have assembled an exact Jacobian, we may verify that its finite difference approximation produces nearly the same result:

```
$ make ecjac
$ ./ecjac -snes_monitor_short
 0 SNES Function norm 2.87411
 1 SNES Function norm 0.859139
 2 SNES Function norm 0.160996
 3 SNES Function norm 0.0110689
 4 SNES Function norm 6.61811e-05
 5 SNES Function norm 2.41926e-09
Vec Object: 1 MPI processes
  type: seq
0.319632
0.947542
$ ./ecjac -snes_monitor_short -snes_fd
 0 SNES Function norm 2.87411
 1 SNES Function norm 0.859139
 2 SNES Function norm 0.160996
 3 SNES Function norm 0.0110689
 4 SNES Function norm 6.61814e-05
 5 SNES Function norm 2.42078e-09
Vec Object: 1 MPI processes
  type: seq
0.319632
0.947542
```

PETSC can provide direct assistance with debugging a user's Jacobian code. Option `-snes_test_jacobian` compares the results of `FormJacobian()` evaluations to finite difference approximations which need only `FormFunction()` evaluations. The user must read the output and decide if the comparison is good:

```
$ ./ecjac -snes_test_jacobian
----- Testing Jacobian -----
Run with ... -snes_test_jacobian ... to show difference
  of hand-coded and finite difference Jacobian entries ...
Testing hand-coded Jacobian, if ... ||J - Jfd||_F/||J||_F is
  O(1.e-8), the hand-coded Jacobian is probably correct.
||J - Jfd||_F/||J||_F = 1.9182e-08, ||J - Jfd||_F = 1.52973e-07
----- Testing Jacobian -----
||J - Jfd||_F/||J||_F = 1.87628e-08, ||J - Jfd||_F = 7.85803e-08
----- Testing Jacobian -----
||J - Jfd||_F/||J||_F = 1.88596e-08, ||J - Jfd||_F = 5.98169e-08
----- Testing Jacobian -----
||J - Jfd||_F/||J||_F = 1.79441e-08, ||J - Jfd||_F = 5.2901e-08
----- Testing Jacobian -----
||J - Jfd||_F/||J||_F = 1.41658e-08, ||J - Jfd||_F = 4.15227e-08
Vec Object: 1 MPI processes
  type: seq
0.319632
0.947542
```

Noting that the relative error in the finite difference Jacobian evaluation is  $O(\sqrt{\epsilon}) = O(10^{-8})$ , we have good evidence that `FormJacobian()` is correct.

## A nonlinear diffusion-reaction equation

Our first nonlinear PDE is a two-point boundary value problem, so it is actually an ODE. The equation

$$-u'' - R(u) = f \quad (4.11)$$

might model the balance of *diffusion* ( $u''$ ), *reaction* ( $R(u)$ ), and *source* ( $f = f(x)$ ) processes for a substance with concentration  $u(x)$ . We will only consider Dirichlet boundary conditions, namely  $u(0) = \alpha$  and  $u(1) = \beta$ .

Equation (4.11) is the steady state of the time-dependent model

$$u_t = u_{xx} + R(u) + f \quad (4.12)$$

for  $u(t, x)$ , an equation which generalizes the one-dimensional, time-evolving heat equation  $u_t = u_{xx}$  (Chapter 5). In a context where  $u$  represents temperature, term  $R(u)$  might model a heat-producing/absorbing chemical reaction and  $f$  an applied heating/cooling source (according to sign). Note that if  $u$  solves (4.12) then positive values of  $R(u) + f$  tend to increase  $u$  relative to solutions of  $u_t = u_{xx}$ .

One may regard (4.11) as a nonlinear elliptic PDE in one dimension. Standard techniques then show that the problem is well posed if  $R(u)$  is a continuous and nonincreasing function because then the nonlinear operator in (4.11) is strictly monotone [91, p. 83] and coercive on the appropriate function space, namely the Sobolev space  $H_0^1(0, 1)$  [51]. Abstract arguments then show unique existence of a solution [91, pp. 93–94].

However, equation (4.11) may not be solvable in cases where  $R$  is an increasing function of  $u$ . If  $R$  is positive and increasing then the ability of the diffusion term to damp out maxima in  $u(x)$  may be exceeded by the production there, so that balance (4.11) becomes impossible. If  $R$  is negative and increasing then the same concern applies at minima of  $u(x)$ . For example, if  $R(u) = \lambda e^u$  then (4.11) is not solvable for sufficiently large  $\lambda$ ; see Exercise 4.5.

As a result of these considerations we plan to solve

$$-u'' + \rho\sqrt{u} = 0. \quad (4.13)$$

This is of form (4.11) with  $R(u) = -\rho\sqrt{u}$  and  $f(x) = 0$ . Because  $R$  is nonincreasing and continuous if  $\rho > 0$ , the corresponding Dirichlet problem is well posed, and also this problem has a convenient exact solution. In fact, both the second-derivative and the square-root operations convert certain 4th-degree polynomials into quadratic polynomials [119, Exercise 5.50]: Substituting  $u(x) = M(x + 1)^4$  into (4.13) finds  $M = (\rho/12)^2$ . We then obtain the boundary conditions from the exact solution:  $\alpha = M$  and  $\beta = 16M$ .

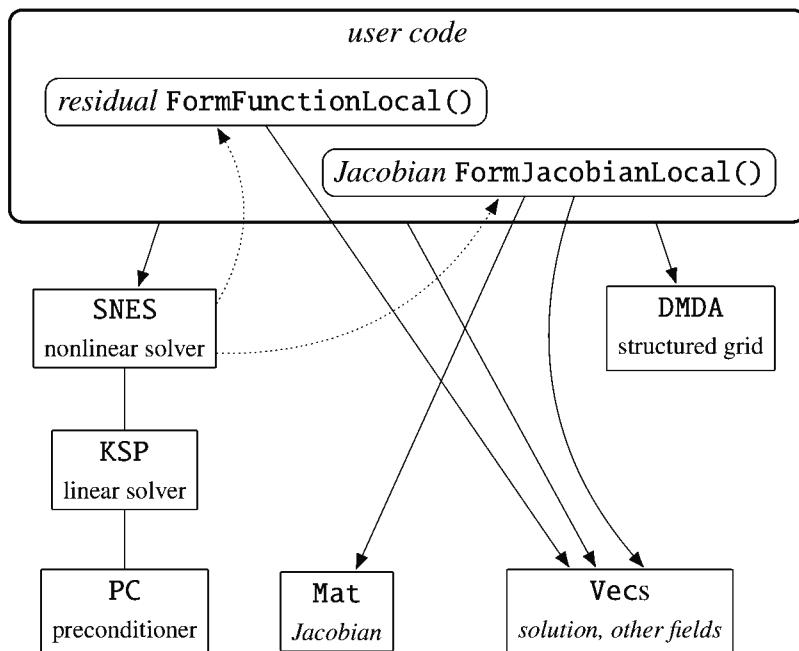
## A numerical method and its convergence

A centered finite difference scheme for differential equation (4.11), on an  $N$ -point grid with  $h = 1/(N - 1) > 0$ ,  $x_j = jh$  for  $j = 0, 1, \dots, N - 1$ , and  $u_j \approx u(x_j)$ , is

$$-\frac{u_{j+1} - 2u_j + u_{j-1}}{h^2} - R(u_j) = f(x_j). \quad (4.14)$$

This scheme has  $O(h^2)$  local truncation error [115].

Our implementation of (4.14), shown in Codes 4.4 and 4.5 below, is called `reaction.c`. This solver uses a SNES object for the Newton iteration and a DMDA object for the grid, and it includes functions which compute  $\mathbf{F}(\mathbf{u})$  and  $J_{\mathbf{F}}(\mathbf{u})$ . Figure 4.4 shows the overall structure. There are six major PETSC object types, but we explicitly create and destroy only three, namely DMDA, SNES,



**Figure 4.4.** An overview of `reaction.c`, a structured-grid solver for a nonlinear PDE. Solid arrows should be read as “user code acts directly on,” while dotted arrows are call-backs. Compare with Figure 3.13.

and `Vec` (Code 4.5). We will also set entries in a `Mat` created by the `DMDA` object, and, because each step of Newton’s method solves a linear system, `KSP` and `PC` objects exist “inside” the `SNES`, but user code does not touch them directly. However, we will need to control these parts of the solver—option `-snes_view` reveals the structure—if we are to get good performance on fine grids.

Functions `FormFunctionLocal()` and `FormJacobianLocal()` in Code 4.4 are call-backs provided to `SNES`. Each takes one argument of type `DMDALocalInfo*` and one or two arguments of type `PetscReal*`. The former provides both the local part of the grid and the global grid size, as shown in Figure 3.7. The `PetscReal *u` arguments are C arrays containing the current Newton iterate, with valid ghost points duplicated from neighboring processes (when run in parallel). In particular, expressions `u[i-1]` and `u[i+1]` are always valid if `i` is an interior-point index. Though we never see the corresponding `Vec` directly, the `SNES`, in fact, has allocated a local `Vec` with memory for the ghost locations, communicated between processes to update those ghost values, and then used `DMDAVecGetArray()` and `DMDAVecRestoreArray()` to get a C array, namely the `PetscReal *u` pointer which is provided to `FormFunctionLocal()` and `FormJacobianLocal()`. (This sequence is also illustrated by the implementation of our function `InitialAndExact()`; see the `reaction.c` code itself.)

```
PetscErrorCode FormFunctionLocal(DMDALocalInfo *info, PetscReal *u,
                                PetscReal *FF, AppCtx *user) {
    PetscInt i;
    PetscReal h = 1.0 / (info->mx-1), x, R;
    for (i=info->xs; i<info->xs+info->xm; i++) {
        if (i == 0) {
            FF[i] = u[i] - user->alpha;
        } else if (i == info->mx-1) {
            FF[i] = u[i] - user->beta;
        } else { // interior location
    }
}
```

```

    if (i == 1) {
        FF[i] = - u[i+1] + 2.0 * u[i] - user->alpha;
    } else if (i == info->mx-2) {
        FF[i] = - user->beta + 2.0 * u[i] - u[i-1];
    } else {
        FF[i] = - u[i+1] + 2.0 * u[i] - u[i-1];
    }
    R = - user->rho * PetscSqrtReal(u[i]);
    x = i * h;
    FF[i] -= h*h * (R + f_source(x));
}
}
return 0;
}

PetscErrorCode FormJacobianLocal(DMDALocalInfo *info, PetscReal *u,
                                 Mat J, Mat P, AppCtx *user) {
    PetscInt i, col[3];
    PetscReal h = 1.0 / (info->mx-1), dRdu, v[3];
    for (i=info->xs; i<info->xs+info->xm; i++) {
        if ((i == 0) | (i == info->mx-1)) {
            v[0] = 1.0;
            MatSetValues(P,1,&i,1,&i,v,INSERT_VALUES);
        } else {
            col[0] = i;
            v[0] = 2.0;
            if (!user->noRinJ) {
                dRdu = - (user->rho / 2.0) / PetscSqrtReal(u[i]);
                v[0] -= h*h * dRdu;
            }
            col[1] = i-1;   v[1] = (i > 1) ? - 1.0 : 0.0;
            col[2] = i+1;   v[2] = (i < info->mx-2) ? - 1.0 : 0.0;
            MatSetValues(P,1,&i,3,col,v,INSERT_VALUES);
        }
    }
    MatAssemblyBegin(P,MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(P,MAT_FINAL_ASSEMBLY);
    if (J != P) {
        MatAssemblyBegin(J,MAT_FINAL_ASSEMBLY);
        MatAssemblyEnd(J,MAT_FINAL_ASSEMBLY);
    }
    return 0;
}

```

**Code 4.4.** *c/ch4/reaction.c, part I. Functions computing the residual  $\mathbf{F}(\mathbf{u})$  and Jacobian  $J_{\mathbf{F}}(\mathbf{u})$ .*

The `main()` function in Code 4.5 has the usual parts which create, configure, and destroy various objects. Note we use structured-grid versions of `SNESSetFunction()` and `SNESSetJacobian()`, and never explicitly allocate a `Mat` to hold the Jacobian. Instead, the `DMDA` object has enough information about the grid and stencil to do the allocation internally (using the `DMCreateMatrix()` function; see Chapter 10).

```

int main(int argc,char **args) {
    DM          da;
    SNES        snes;
    AppCtx     user;
    Vec         u, uexact;
    PetscReal   errnorm, *au, *auex;
    DMDALocalInfo info;

```

```

PetscInitialize(&argc,&args,NULL,help);
user.rho    = 10.0;
user.M      = PetscSqr(user.rho / 12.0);
user.alpha  = user.M;
user.beta   = 16.0 * user.M;
user.noRinJ = PETSC_FALSE;

PetscOptionsBegin(PETSC_COMM_WORLD,"rct_","options for reaction","");
PetscOptionsBool("-noRinJ","do not include R(u) term in Jacobian",
                 "reaction.c",user.noRinJ,&(user.noRinJ),NULL);
PetscOptionsEnd();

DMCreate1d(PETSC_COMM_WORLD,DM_BOUNDARY_NONE,9,1,1,NULL,&da);
DMSetFromOptions(da);
DMSetUp(da);
DMSetApplicationContext(da,&user);

DMCreateGlobalVector(da,&u);
VecDuplicate(u,&uexact);
DMDAVecGetArray(da,u,&au);

DMDAGetLocalInfo(da,&info);
DMDAVecGetArray(da,uexact,&auex);
InitialAndExact(&info,au,auex,&user);
DMDAVecRestoreArray(da,u,&au);
DMDAVecRestoreArray(da,uexact,&auex);

SNESCreate(PETSC_COMM_WORLD,&snes);
SNESSetDM(snes,da);
DMDASNESSetFunctionLocal(da,INSERT_VALUES,
                         (DMDASNESFunction)FormFunctionLocal,&user);
DMDASNESSetJacobianLocal(da,
                          (DMDASNESJacobian)FormJacobianLocal,&user);
SNESSetFromOptions(snes);

SNESolve(snes,NULL,u);

VecAXPY(u,-1.0,uexact); // u <- u + (-1.0) uexact
VecNorm(u,NORM_INFINITY,&errnorm);
PetscPrintf(PETSC_COMM_WORLD,
           "on %d point grid: |u-u_exact|_inf = %g\n",info.mx,errnorm);

VecDestroy(&u); VecDestroy(&uexact);
SNESDestroy(&snes); DMDestroy(&da);
return PetscFinalize();
}

```

**Code 4.5.** *c/ch4/reaction.c, part II.* In `main()` we create a DM<sub>DA</sub>, a SNES, and some Vecs, pass functions to the SNES via the DM<sub>DA</sub>, and then solve the equation.

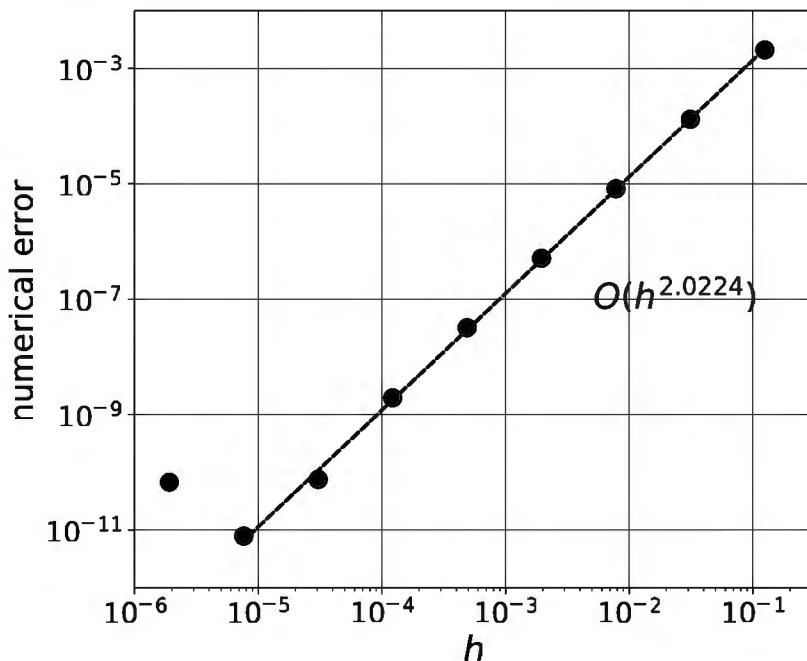
An application context struct is used in our functions:

```

typedef struct {
    PetscReal rho, M, alpha, beta;
    PetscBool noRinJ;
} AppCtx;

```

This includes the physical parameter  $\rho$  and also problem parameters  $\alpha, \beta, M$  for problem (4.13) and its exact solution. The Boolean flag `noRinJ` will be used to optionally remove the reaction term in the calculation of the Jacobian (below).



**Figure 4.5.** Numerical error  $\|u - u_{exact}\|_\infty$  versus grid spacing  $h$ .

Having implemented both a residual function and a Jacobian, we should check the latter by comparing behavior from analytical and finite-differenced Jacobians. We can quickly see that on a modestly refined grid the number of Newton iterations is identical:

```
$ make reaction
$ ./reaction -snes_converged_reason -da_refine 6
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 3
on 513 point grid: |u-u_exact|_inf = 5.13617e-07
$ ./reaction -snes_converged_reason -da_refine 6 -snes_fd
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 3
on 513 point grid: |u-u_exact|_inf = 5.13617e-07
```

and the residual norms from `-snes_monitor` are close (not shown). We can also use `-snes_test_jacobian` on a coarse grid (not shown) or graph the Newton iterates themselves in a solver “movie”:

```
| $ ./reaction -da_refine 6 -snes_monitor_solution draw -draw_pause 1
```

This view (not shown) confirms that the first Newton step moves close to the solution, and then quadratic convergence sets in, with little visible change. Such evidence suggests we have a correctly implemented Jacobian.

The next concern is convergence under grid refinement. By using the exact solution, we want to check that the rate at which the numerical error goes to zero is  $O(h^2)$  like the local truncation error of our scheme. Consider the following loop, with results in Figure 4.5:

```
$ for N in 0 2 4 6 8 10 12 14 16; do
    ./reaction -da_refine $N -snes_rtol 1.0e-10; done
on 9 point grid: |u-u_exact|_inf = 0.00209725
on 33 point grid: |u-u_exact|_inf = 0.000131389
...
```

If we ignore the result on the finest grid then the convergence rate is  $O(h^2)$ . By adding `-snes_converged_reason` we also see consistent evidence of quadratic convergence of the Newton iteration at all levels of refinement (not shown).

Our implementation is correct, and, in fact, the numerical solution shows ideal behavior on this easy problem. However, regarding the anomalous finest-grid result shown in Figure 4.5, it is another numerical “fact of life” that measurable convergence always runs out:

**Fact 9.** Error stagnation will occur at some level of refinement. *For a given floating-point precision, at some point in the refinement path the round-off error will become comparable with the discretization error. Beyond this level, convergence cannot be verified.*

In 2D and 3D examples the refinement levels at which round-off and discretization errors compete are often never reached. Either in terms of run time or memory usage, solving on grids of the necessary resolution is too costly.

## Finite-differenced Jacobians by coloring

Finite-differenced Jacobians save programmer effort, at least at the prototyping stage, and in many cases in production solvers as well. However, runs using `-snes_fd` cause far too many function evaluations when applied to PDEs, so this option cannot be used for 2D and 3D PDE problems at practical levels of refinement. To show the concern in our current problem, the following run requires three Newton iterations, three evaluations of our analytical Jacobian  $J_F$ , four evaluations of  $F$ , and less than 0.1 seconds:

```
$ ./reaction -snes_converged_reason -da_refine 10
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 3
on 8193 point grid: |u-u_exact|_inf = 1.95115e-09
```

(Add `-log_view | grep Eval` to confirm evaluation counts, and likewise for timing.) By contrast the finite difference Jacobian method seems to fail:

```
$ ./reaction -snes_converged_reason -da_refine 10 -snes_fd
Nonlinear solve did not converge due to DIVERGED_FUNCTION_COUNT iterations 1
on 8193 point grid: |u-u_exact|_inf = 0.0549199
```

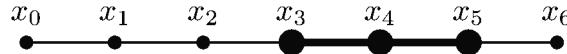
The difficulty is that a single Jacobian evaluation using finite difference formula (4.7) requires  $N + 1 = 8194$  evaluations of  $F$ , one for each column of the Jacobian plus one for the right side of (4.4a), and thus SNES’s default limit of 10000 evaluations of  $F$  is exceeded during the three Newton iterations. If we raise the limit on function evaluations then we do get convergence, but about 30 times slower:

```
$ ./reaction -snes_converged_reason -da_refine 10 -snes_fd \
-snes_max_funcs 100000
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 3
on 8193 point grid: |u-u_exact|_inf = 1.95152e-09
```

This run evaluates  $F$  about  $3N \approx 24000$  times. The trend is unsustainable.

However, an exploitable property of the algebraic systems generated by discretizing PDEs is that they have a small number of unknowns per equation. Finite difference, element, and volume schemes all use a small set of grid values—a small stencil—in approximating the PDE at each location.<sup>18</sup> This observation leads to an effective, graph-theoretic idea for rescuing the finite

<sup>18</sup>Spectral methods [142] have a global stencil and may not benefit from coloring.



**Figure 4.6.** `reaction.c` uses discretization (4.14) at each interior node of the grid. This corresponds to a three-point stencil, as shown.

difference approach. The new idea applies both on structured grids and unstructured meshes; an example of the latter is in Chapter 10.

The new idea is to “color” the unknowns of the problem, and thus the columns of the Jacobian, with a small set of colors so that each scalar equation in the system  $\mathbf{F}(\mathbf{u}) = \mathbf{0}$  relates unknowns with distinct colors. The “colors” are merely integers  $\{0, 1, 2, \dots, c - 1\}$ . Finite difference formula (4.7) can then be redesigned using the coloring, as shown below, to allow computation of entries in many columns of  $J = J_{\mathbf{F}}(\mathbf{u})$  simultaneously. The number of  $\mathbf{F}$  evaluations is reduced to one more than the number of colors needed for a proper vertex coloring [34] of a graph constructed from the sparsity (nonzero) pattern of  $J$ .

Before getting into any details we show a small example. Consider this seven-point grid:

```
| $ ./reaction -da_grid_x 7
```

Figure 4.6 shows the grid and the stencil for scheme (4.14). The scalar equation at  $x_j$  involves three unknowns  $u_{j-1}$ ,  $u_j$ , and  $u_{j+1}$  so the  $j$ th row of  $J$  has three nonzero entries. (Exceptions occur at the boundary nodes. Use option `-ksp_view_mat` to show the Jacobian matrix at each Newton iteration.)

One can assign  $c = 3$  colors  $\{0, 1, 2\}$  to the seven columns of  $J$  as follows:

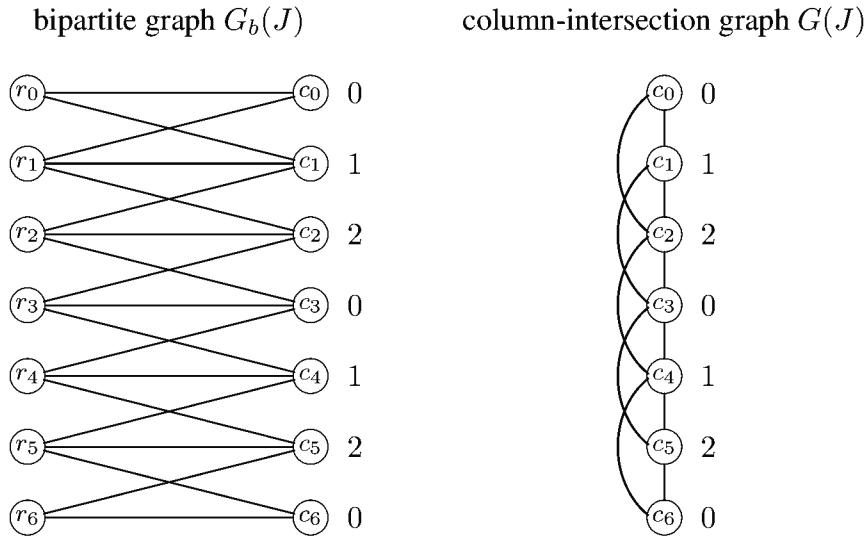
$$J = \begin{bmatrix} J_{00} & J_{01} & & & & & \\ J_{10} & J_{11} & J_{12} & & & & \\ & J_{21} & J_{22} & J_{23} & & & \\ & & J_{32} & J_{33} & J_{34} & & \\ & & & J_{43} & J_{44} & J_{45} & \\ & & & & J_{54} & J_{55} & J_{56} \\ & & & & & J_{65} & J_{66} \end{bmatrix} \rightarrow \begin{array}{cccccc} 0 & 1 & & & & \\ 0 & 1 & 2 & & & \\ 1 & 2 & 0 & & & \\ 2 & 0 & 1 & & & \\ 0 & 1 & 2 & & & \\ 1 & 2 & 0 & & & \\ 2 & 0 & & & & \end{array} \quad (4.15)$$

Each row thus has distinct colors.

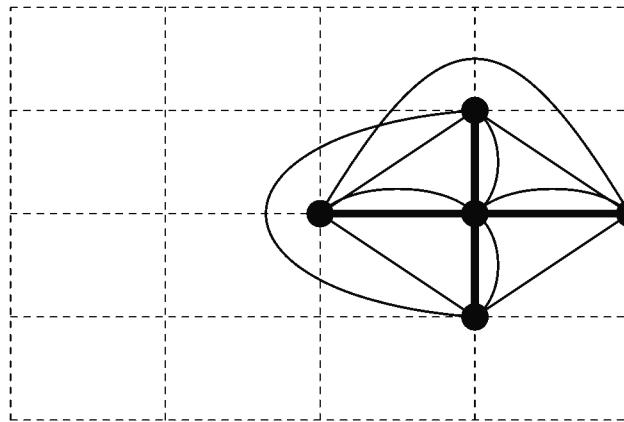
Actually, there are two graphs associated to this coloring problem. On the left in Figure 4.7 is an (undirected) bipartite graph  $G_b(J)$  with two sets of vertices, the rows  $r_0, \dots, r_6$  and the columns  $c_0, \dots, c_6$ , with edges only between a row and a column. There is an edge  $(r_i, c_j)$  if and only if  $J_{ij} \neq 0$ . (Note this bipartite graph construction ignores symmetry.) The goal is then to assign colors to the column vertices so that any two vertices  $c_k$  and  $c_\ell$  connected by a length two path get different colors, a *distance-2* coloring [57] of the columns. The coloring algorithms in PETSc generate such bipartite, distance-2 colorings.

On the other hand, for symmetric matrices like  $J$  here one can construct an (undirected) graph on the columns only, shown at right in Figure 4.7. This *column-intersection graph* [37]  $G(J)$  has an edge  $(c_i, c_j)$  if and only if unknowns  $u_i$  and  $u_j$  both appear in at least one of the equations of the system  $\mathbf{F}(\mathbf{u}) = \mathbf{0}$ , that is, if columns  $c_i$  and  $c_j$  of  $J$  have nonzero entries in a common row. For this graph one seeks a distance-1 coloring so that if  $(c_i, c_j)$  is an edge in  $G(J)$  then  $c_i$  and  $c_j$  are assigned distinct colors. The *chromatic number* [34] of  $G(J)$  is the minimum number of such colors. For the case shown in the Figure 4.7 this number is  $\chi(G(J)) = 3$ , so we cannot do better with our coloring. For symmetric matrices these two coloring problems are equivalent, but the bipartite, distance-2 framework is more flexible [57].

For a structured 2D grid the stencil of our scheme for the Poisson equation (Chapters 3 and 6) involves five unknowns  $u_{i,j+1}$ ,  $u_{i-1,j}$ ,  $u_{i,j}$ ,  $u_{i+1,j}$ ,  $u_{i,j-1}$ . Thus the graph  $G(J)$  includes



**Figure 4.7.** From a symmetric matrix  $J$  one can either build a bipartite graph for the nonzero pattern (left), then find a distance-2 coloring on the column vertices  $\{c_i\}$  only, or build the column-intersection graph (right) and find a distance-1 coloring.



**Figure 4.8.** For the 2D finite difference scheme used in Chapter 3, the graph  $G(J)$  has a  $K_5$  at every node because the stencil (thick lines) involves five unknowns.

a complete graph [34] on five vertices, a so-called  $K_5$ , at each generic interior node, as shown in Figure 4.8, so  $\chi(G(J)) \geq 5$ . PETSC’s default coloring algorithm finds, at least in generic refined cases, that indeed five colors suffice to color  $G(J)$ .

Optimally coloring a graph is, however, a hard problem which PETSC does not attempt to solve. Instead the “incidence-degree ordering” [37], the default, and “smallest-last ordering” [57] algorithms provide  $c$ -colorings for which  $c \leq \alpha N^{1/2} \chi(G(J))$ , for some constant  $\alpha > 0$ , where  $N$  is the number of equations in (4.1). These algorithms also do well in the informal sense that  $c$  is close to  $\chi(G(J))$  for a large selection of test matrices. In any case, they run in a time proportional to the number of nonzeros in  $J$  [37] so they are inexpensive.

Finite-difference formula (4.7) is modified to use a coloring of graph  $G_b(J)$  and/or  $G(J)$ , i.e., a coloring of the columns of  $J$ , in the following way. One first generates vectors  $\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{c-1} \in \mathbb{R}^N$  so that  $\mathbf{v}_k$  is 1 in entry  $j$  if  $k$  is the color of column  $c_j$ , and otherwise is zero.

In the  $c = 3$  case shown in Figure 4.7,

$$\mathbf{v}_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{v}_1 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}. \quad (4.16)$$

We also define a function  $k(j)$  which maps from column index  $j$  to the color  $k$  of  $c_j$ , thus  $(\mathbf{v}_k)_j = \delta_{k,k(j)}$ . Then (4.7) is replaced with

$$J_{ij} = \frac{\partial F_i}{\partial x_j} \approx \frac{F_i(\mathbf{x} + \delta \mathbf{v}_{k(j)}) - F_i(\mathbf{x})}{\delta}. \quad (4.17)$$

The right sides of (4.7) and (4.17) compute exactly the same entries  $J_{ij}$ , but (4.17) requires far fewer evaluations of  $\mathbf{F}$ . In particular, all columns of  $J$  with color  $k$  are computed by (4.17) using only the smallest  $j$  for which  $k(j) = k$ . Thus, given a  $c$ -coloring of  $G_b(J)$  or  $G(J)$  there are exactly  $c$  evaluations  $\mathbf{F}(\mathbf{x} + \delta \mathbf{v}_{k(j)})$ , plus one more for  $\mathbf{F}(\mathbf{x})$  itself, sufficient to fill  $J$ .

The news is good when we actually try coloring on `reaction.c`, with the following run that is just as fast as the one shown on page 83:

```
$ ./reaction -snes_converged_reason -da_refine 10 -snes_fd_color
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 3
on 8193 point grid: |u-u_exact|_inf = 1.95152e-09
```

Counting the number of function evaluations in the various cases gives a more precise comparison than timing. Note there is no need to alter `reaction.c` to count evaluations; just use `-log_view` and `grep` for `Eval`. The result is that for a  $N = 8193$  point grid the number of  $\mathbf{F}$  evaluations was 24586, 13, and 4, respectively, for `-snes_fd`, `-snes_fd_color`, and analytical-Jacobian methods. While 13 is larger than 4, remember that the analytical Jacobian method requires 3 evaluations of user code for  $J_{\mathbf{F}}$ .

We will see in future Chapters that coloring is an effective tool in 2D and 3D structured-grid cases (Chapters 7, 9, and 11) and even with unstructured meshes (Chapter 10).

## Jacobian-free Newton-Krylov (JFNK)

A different approach also avoids user-written Jacobian-evaluation code. The *Jacobian-free Newton-Krylov* (JFNK) method [96] seeks a solution  $\mathbf{s}$  to the Newton step equation (4.4a) from a Krylov subspace

$$\mathcal{K}_m = \text{span}\{\mathbf{r}, J\mathbf{r}, J^2\mathbf{r}, \dots, J^{m-1}\mathbf{r}\} \quad (4.18)$$

without computing or storing the entries of the Jacobian matrix  $J = J_{\mathbf{F}}(\mathbf{x}_k)$ . JFNK instead approximates Jacobian-vector products  $J\mathbf{v}$  by a finite difference formula

$$J\mathbf{v} \approx \frac{\mathbf{F}(\mathbf{x} + \delta \mathbf{v}) - \mathbf{F}(\mathbf{x})}{\delta}. \quad (4.19)$$

Thus JFNK only computes the action of  $J$  on  $\mathbf{v}$ , not the entries  $J_{ij}$ . One usually starts from initial iterate  $\mathbf{s} = 0$ , so  $\mathbf{r} = -\mathbf{F}(\mathbf{x}_k) - J\mathbf{0} = -\mathbf{F}(\mathbf{x}_k)$  is the first basis vector in  $\mathcal{K}_m$ . Then (4.19) computes  $J\mathbf{r}, J^2\mathbf{r} = J(J\mathbf{r}), \dots$ , thus building space  $\mathcal{K}_m$  in  $m$  total evaluations of  $\mathbf{F}$ .

Recall that finite difference formula (4.7) computes a generic  $N \times N$  Jacobian via  $N$  evaluations of  $\mathbf{F}$ . Thus, relative to forming  $J$  using `-snes_fd`, JFNK is worthwhile if the Newton step equation (4.4a) can be solved to desired tolerance in  $m \ll N$  iterations. Other considerations

might also suggest using JFNK, such as if we have no good way to store an  $N \times N$  matrix in memory or if the memory bandwidth for loading matrix entries is too slow, or if a coloring is not available. On the other hand, as we will see on structured grids, the -snes\_fd\_color approach using equation (4.17) is often superior to JFNK. This is because the number of colors is usually smaller than  $m$  and because preconditioners based on matrices (e.g., ILU) can exploit the computed Jacobian entries.

JFNK is a good strategy to the extent that it actually works, so let us try it. Un-preconditioned JFNK is invoked by option -snes\_mf, where mf in stands for “matrix-free.” For example:

```
$ ./reaction -snes_converged_reason -snes_mf
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 3
on 9 point grid: |u-u_exact|_inf = 0.00209725
```

This is fine, but additional results from refined grids will damp our enthusiasm. Noting  $J$  is symmetric, we use the CG method (Chapter 2). On 500 and 1000 point grids the performance is poor in the sense that far too many “inner” Krylov iterations are needed:

```
$ ./reaction -snes_converged_reason -snes_mf -ksp_converged_reason \
-ksp_type cg -da_refine 6
Linear solve converged due to CONVERGED_RTOL iterations 511
Linear solve converged due to CONVERGED_RTOL iterations 504
Linear solve converged due to CONVERGED_RTOL iterations 506
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 3
on 513 point grid: |u-u_exact|_inf = 5.13618e-07
$ ./reaction -snes_converged_reason -snes_mf -ksp_converged_reason \
-ksp_type cg -da_refine 7
Linear solve converged due to CONVERGED_RTOL iterations 1023
Linear solve converged due to CONVERGED_RTOL iterations 1005
Linear solve converged due to CONVERGED_RTOL iterations 996
Linear solve converged due to CONVERGED_RTOL iterations 960
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 4
on 1025 point grid: |u-u_exact|_inf = 1.28418e-07
```

These large iteration counts, apparently proportional to the number of unknowns  $N$ , illustrate the well-known doubling of un-preconditioned CG iterations for FD solutions of the Poisson problem under  $h \rightarrow h/2$  refinement (Chapter 3).

Krylov method convergence generally requires preconditioning, and Newton equation (4.4a) is no exception. That is, we want to build  $\mathcal{K}_m$  for a linear operator  $M^{-1}J$ , not  $J$  itself, in the hope that  $m \ll N$  Krylov iterations give sufficient accuracy. However, most preconditioning approaches need an assembled matrix to generate the action of  $M^{-1}$ . For example, incomplete matrix factorizations and algebraic multigrid (Chapter 10) act on matrix entries. In this sense we see that, though the approach may be “Jacobian-free,” to be effective it should not be entirely matrix-free [96].

The left- and right-sided preconditioned forms (Chapter 2) of the Newton step equation (4.4a) are

$$(M^{-1}J)\mathbf{s} = M^{-1}\mathbf{r}, \quad (4.20)$$

$$(JM^{-1})(M\mathbf{s}) = \mathbf{r}, \quad (4.21)$$

respectively, where  $\mathbf{r} = -\mathbf{F}(\mathbf{u}_k)$ . In (4.20) the action of  $M^{-1}J$  on some vector  $\mathbf{v}$  is a straightforward composition of the finite difference formula (4.19) followed by application of  $M^{-1}$ . In (4.21) the action of  $JM^{-1}$  is computed by first applying  $M^{-1}$  to  $\mathbf{v}$  (i.e., solving  $M\mathbf{y} = \mathbf{v}$ ) and then using (4.19) in the form  $(JM^{-1})\mathbf{v} \approx (\mathbf{F}(\mathbf{x} + \delta\mathbf{y}) - \mathbf{F}(\mathbf{x}))/\delta$ . Recall that  $M$  and  $M^{-1}$  are usually not assembled, but the action of  $M^{-1}$  is computed by an algorithm which extracts entries from an assembled preconditioner-material matrix.

**Table 4.2.** Jacobian options when using SNES, listed according to what evaluation functions are provided by the user:  $\mathbf{F}$  for a residual function,  $J = J_{\mathbf{F}}$  for an exact Jacobian, and  $K \approx J_{\mathbf{F}}$  for an approximate Jacobian. A check mark indicates feasibility, with an underline for recommended usage on PDE problems.

option	Implemented functions		
	$\mathbf{F}$ only	$\mathbf{F}$ and $J$	$\mathbf{F}$ and $K$
none		✓	✓
-snes_fd	✓		
-snes_fd_color	✓		
-snes_mf	✓		
-snes_mf_operator		✓	✓

## Testing Jacobian cases

We are now in position to test various Jacobian choices for our Newton-method solutions, but at risk of being confused by the multiplying options. A pause to review the possibilities, before testing them on our reaction-diffusion equation problem, makes sense.

First, at the minimum, the user must provide a residual function  $\mathbf{F}$ , using `SNESSetFunction()` or `DMDASNESSetFunctionLocal()`, because the SNES needs to know what equations are being solved!<sup>19</sup> Then Table 4.2 summarizes the Jacobian-usage options for SNES-based codes.

If only  $\mathbf{F}$  is provided then the available methods all use finite difference approximations of the Jacobian. In these cases the user does not create a `Mat` for the Jacobian, as this is done internally by the SNES for `-snes_fd` and `-snes_fd_color`, and never done for `-snes_mf`.

If a Jacobian evaluation routine  $J = J_{\mathbf{F}}$  is provided by the user then it is used in the Newton iteration when no option is given. This is the preferred usage, at least once the Jacobian is checked for correctness, but writing an exact Jacobian may be a burden. If instead an approximate Jacobian  $K \approx J_{\mathbf{F}}$  is provided then it is reasonable to use it only for preconditioning the Jacobian-vector product in JFNK, and option `-snes_mf_operator` does this. This approach can achieve both quadratic convergence and small Krylov iteration counts if the chosen preconditioning method, acting on the preconditioner material provided by  $K$ , generates an operator  $M$  which is spectrally equivalent to the exact Jacobian [96]. (This means that  $M^{-1}J$  has eigenvalues clustered around 1. See more on this topic in Chapter 7.) We will illustrate this usage momentarily.

There are two API choices for providing an implemented Jacobian, either  $J$  or  $K$ , to the SNES:

- On a DMDA structured grid, set the call-back to your function, named `FormJacobianLocal()` or similar, using `DMDASNESSetJacobianLocal()`.
- Otherwise, declare a `Mat` variable, say  $\mathbf{J}$ , and then call `MatCreate()`, `MatSetSizes()`, and `MatSetFromOptions()` on it. Then call `MatSetUp()`, or preferably `MatXAIJPreallocate()`,<sup>20</sup> to allocate space for the assembled Jacobian matrix. The call-back function, e.g., `FormJacobian()`, is then provided through `SNESSetJacobian()`.

<sup>19</sup>The only exception is when the residual function  $\mathbf{F}$  is the gradient of an objective function, thus the problem is really optimization [118], and when the user is prepared to accept very poor solver performance. Though an example is given in Chapter 9, this usage is never suitable for solving PDEs at significant resolution.

<sup>20</sup>A nontrivial example of pre-allocation, in an unstructured FE scheme, is in Chapter 10.

**Table 4.3.** A simplified model for the number of call-back evaluations for an  $N$ -dimensional problem solved in  $q$  SNES iterations. Here  $c$  is the number of colors and  $m$  is the Krylov space dimension at KSP convergence.

Option	$F$ evaluations	$J$ or $K$ evaluations
none	$q + 1$	$q$
<code>-snes_fd</code>	$q(N + 1) + 1$	0
<code>-snes_fd_color</code>	$q(c + 1) + 1$	0
<code>-snes_mf</code>	$qm$	0
<code>-snes_mf_operator</code>	$qm$	$q$

In the most common usages, `FormJacobian()` or `FormJacobianLocal()` only needs to set values in the second `Mat` argument (the preconditioner-material argument), but both should also do final assembly on both `Mat` arguments. An example is shown in `FormJacobianLocal()` in Code 4.4.

Another way to compare the options is shown in Table 4.3, which models the number of evaluations in various cases. However, even if the number  $q$  of SNES iterations is independent of other choices in the solver, the number of Krylov iterations per Newton step will depend on many aspects of the problem, especially the preconditioner. Regarding the table we note two fundamental points:

- Improved performance of `-snes_fd_color` relative to `-snes_fd` comes from having a  $c$ -coloring where  $c \ll N$ .
- Improved performance of `-snes_mf_operator` relative to `-snes_mf` comes from using a preconditioner, based on the material generated by the function  $K$ , which significantly reduces the number  $m$  of Krylov iterations.

To give a nontrivial demonstration of `-snes_mf_operator` on our diffusion-reaction problem, we have added a Boolean option `-rct_noRinJ` which simplifies the diagonal of the Jacobian, thereby generating an approximation  $K \approx J_F$ . In Code 4.4 one sees

```
v[0] = 2.0;
if (!user->noRinJ) {
    dRdu = - (user->rho / 2.0) / PetscSqrtReal(u[i]);
    v[0] -= h*h * dRdu;
}
```

If `-rct_noRinJ` is set then we keep only the tridiagonal leading-order part of the Jacobian, but this preserves most spectral characteristics of the linearization of (4.11).

Removing a term from the Jacobian formula slows convergence if we use the approximation as though it were exact. At this resolution the iterations go from 3 to 15:

```
$ ./reaction -da_refine 7 -snes_converged_reason
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 3
on 1025 point grid: |u-u_exact|_inf = 1.28363e-07
$ ./reaction -da_refine 7 -snes_converged_reason -rct_noRinJ
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 15
on 1025 point grid: |u-u_exact|_inf = 1.11714e-07
```

The residual norms from `-snes_monitor` also suggest that convergence is no longer quadratic (not shown); this is expected in theory [89]. However, `-snes_mf_operator` with the approximate Jacobian is now an effective option:

```
$ ./reaction -da_refine 7 -snes_converged_reason -rct_noRinJ \
-snes_mf_operator
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 4
on 1025 point grid: |u-u_exact|_inf = 1.28418e-07
```

One can show that on high-resolution grids this usage is competitive with exact and finite-difference-by-coloring Jacobian choices, though in this case not superior to those methods.

Here is a summary of advice on SNES usage:

- (i) Start by implementing, and double-checking, the residual  $\mathbf{F}$ .
- (ii) Before implementing a Jacobian  $J = J_{\mathbf{F}}$ , try finite difference evaluation (`-snes_fd`) first, using coloring (`-snes_fd_color`) if it applies to your case.
- (iii) JFNK with no preconditioning (`-snes_mf`), which needs no Jacobian, is easy to try on small problems but rarely effective upon refinement because KSP iterations grow rapidly with this un-preconditioned operator.
- (iv) Consider implementing the exact Jacobian  $J$ . If this is too much work, or too error prone, consider a simpler approximate Jacobian  $K \approx J_{\mathbf{F}}$ , used with option `-snes_mf_operator`.

The first item of advice (i) must not be forgotten; it is another numerical fact of life:

Fact 10. Residual-evaluation code must be correct. *All other choices about solving nonlinear equations—Jacobian-evaluation methods, linear solvers, line search, initial iterates, etc.—are irrelevant if your implementation of the nonlinear residual  $\mathbf{F}(\mathbf{x})$  is wrong.*

Regarding item (iv), in PDE cases an approximate Jacobian  $K$  should capture at least the correctly-scaled highest-order derivatives in the true Jacobian. Runs using `-snes_mf_operator` with such  $K$  can then show quadratic convergence, assuming it would occur using the exact Jacobian. On the other hand, one may do “no option” runs which use  $K$  as though it were  $J$ , but then convergence will generally revert to a linear rate.

As further practical advice for the debugging stage, you have correctly and fully implemented the Jacobian if all checked options in Table 4.2 work without error messages for medium resolution grids, *and* if the recommended options give apparent quadratic convergence (i.e., from `-snes_monitor`). As the reader may check, this description applies to `reaction.c`.

An adequate initial iterate  $\mathbf{u}_0$  is critical, but in the current 1D problem the simplest reasonable construction, a straight line connecting the boundary conditions, was adequate. In harder problems, finding an adequate  $\mathbf{u}_0$  may require some insight into the nonlinear problem. Chapter 7 proposes a grid-sequencing method for generating high-quality initial iterates on fine grids. We will see that this technique works well for many nonlinear elliptic problems.

## Line-search methods

The Newton step  $\mathbf{s}$  computed by solving equation (4.4a) may not yield a new iterate  $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}$  that we want. The residual norm  $\|\mathbf{F}(\mathbf{x}_k + \mathbf{s})\|$  may exceed  $\|\mathbf{F}(\mathbf{x}_k)\|$ , so no progress is being made in solving (4.1).

A *globalization* technique will actually reduce the residual norm at each iteration, thus expanding the domain of convergence of the Newton iteration. A *line-search* globalization [40] replaces the original update (4.4b) with

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{s}, \quad (4.22)$$

where  $\lambda_k > 0$ . Here we accept  $\mathbf{s}$  as a search *direction* for solving (4.4a), but we may actually use a step of reduced ( $\lambda_k < 1$ ) length. Clearly, the question is how to choose  $\lambda_k$ .

Observe that for a generic system of nonlinear equations  $\mathbf{F}(\mathbf{x}) = \mathbf{0}$  there is no *a priori* sense of whether a given value is “better” or “worse” as an approximation to an exact solution  $\mathbf{x}^*$ . We may, however, introduce a *merit function* [118] to provide such a sense, for example,

$$\phi(\mathbf{x}) = \frac{1}{2} \|\mathbf{F}(\mathbf{x})\|_2^2. \quad (4.23)$$

Merit function (4.23) is used in all SNES line searches, unless a replacement objective function is provided by the user. (See the example in Chapter 9.)

To determine  $\lambda_k$  one might solve the one-dimensional optimization problem

$$\min_{\lambda > 0} \phi(\mathbf{x}_k + \lambda \mathbf{s}). \quad (4.24)$$

That is, one could make the merit function as small as possible along the search direction. However, solving (4.24) accurately is rarely justified because  $\lambda_k$  only determines the next Newton iterate, not the final answer. Also, after the line search gets us out of difficulties, we expect that the quadratically-convergent Newton iteration with full step  $\lambda_k = 1$  is preferable.

Thus a practical line-search algorithm tests candidate values of  $\lambda$  for whether  $\phi(\mathbf{x}_k + \lambda \mathbf{s})$  is reduced. Requiring *sufficient decrease* [118], namely

$$\phi(\mathbf{x}_k + \lambda \mathbf{s}) \leq \phi(\mathbf{x}_k) + \alpha \lambda \mathbf{s}^\top \nabla \phi(\mathbf{x}_k), \quad (4.25)$$

with parameter  $\alpha \in (0, 1)$ , is a common way to guarantee progress. The default value  $\alpha = 10^{-4}$  (`-snes_linesearch_alpha 0.0001`) corresponds to an easy-to-satisfy norm-reduction criterion, but the Newton iteration fails with an error message if (4.25) is not satisfied after a certain number of reductions of  $\lambda$  (default: `-snes_linesearch_max_it 40`).

The quantity  $\mathbf{s}^\top \nabla \phi(\mathbf{x}_k)$  in (4.25) is the directional derivative of  $\phi(\mathbf{x})$  in the search direction  $\mathbf{s}$ . For merit function (4.23) this is always negative, if equation (4.4a) is solved accurately:

$$\mathbf{s}^\top \nabla \phi(\mathbf{x}_k) = \mathbf{s}^\top J_{\mathbf{F}}(\mathbf{x}_k)^\top \mathbf{F}(\mathbf{x}_k) = (J_{\mathbf{F}}(\mathbf{x}_k)\mathbf{s})^\top \mathbf{F}(\mathbf{x}_k) = -\|\mathbf{F}(\mathbf{x}_k)\|_2^2. \quad (4.26)$$

Thus  $\mathbf{s}$  is at least a descent direction [118] in the most common SNES usage. Notice that the same calculation shows that (4.25) can be tested without evaluating the Jacobian.

Merit function (4.23) has another nice property. Though  $\phi(\mathbf{x})$  may have a local minimum  $\mathbf{x}'$  which is not a solution to (4.1), the Jacobian must be singular at such locations. In fact,

$$\mathbf{0} = \nabla \phi(\mathbf{x}') = J_{\mathbf{F}}(\mathbf{x}')^\top \mathbf{F}(\mathbf{x}'), \quad (4.27)$$

so  $J_{\mathbf{F}}(\mathbf{x}')^\top$  has a nonzero null-space if  $\mathbf{F}(\mathbf{x}') \neq \mathbf{0}$ . Thus a local minima of  $\phi$  which is not a solution of  $\mathbf{F}(\mathbf{x}) = \mathbf{0}$  will not arise in a region where there is a finite bound on the condition number of the Jacobian.

The replacement of an objective function by the squared norm of the residual, namely merit function (4.23), was also an algorithmic transition that occurred when solving linear systems  $A\mathbf{x} = \mathbf{b}$  (Chapter 2). Namely, while the CG algorithm for SPD matrices  $A$  minimizes  $g(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top A\mathbf{x} - \mathbf{b}^\top \mathbf{x}$ , if  $A$  is not positive-definite then we may switch to the MINRES or GMRES algorithms which minimize the merit function  $\phi(\mathbf{x}) = \frac{1}{2}\|\mathbf{b} - A\mathbf{x}\|_2^2$ . That is, if the problem is not a true minimization then we switch to the goal of minimizing the residual norm.

The available line-search options are summarized in Table 4.4. Choose the line-search type by `-snes_linesearch_type X` and observe its progress by `-snes_linesearch_monitor`. For more details on line searches see [40, Chapter 6], [89], and the PETSc source code itself.

In this book we use one type of SNES solver, Newton iteration with line search (`-snes_type newtonls`), almost exclusively. PETSc also implements trust region methods (`-snes_type newtontr`) and nonlinear solvers which are not Newton iterations at all, including nonlinear conjugate gradients (`-snes_type ncg`) and quasi-Newton methods (`-snes_type qn`) [118]; the latter two are demonstrated in Chapter 9. For a general perspective and recent advances in nonlinear solvers see [29].

**Table 4.4.** Line-search types; choose using option `-snes_linesearch_type`.

Name	Summary
basic	No line search: $\lambda_k = 1$ in (4.22).
bt	[default] Polynomial-fit back-tracking [40, section 6.3.2]. A cubic polynomial is built up as a model for $f(\lambda) = \phi(\mathbf{x}_k + \lambda\mathbf{s})$ . Successive values $\lambda$ are required to decrease.
cp	Assumes $\mathbf{F}$ is the gradient of an unknown objective function, i.e., $\mathbf{F} = \nabla g$ . The secant method then finds a critical point of $g$ along the search direction.
l2	Secant-line minimization along the search direction, starting from $\lambda = 1/2$ and $\lambda = 1$ , is repeated a fixed number of times.

## Exercises

- 4.1. One needs to see quadratic convergence to believe it. Observe that for both parts (b) and (c) below, the number of correct digits in  $x_k$  *doubles* at each iteration.
- (a) The sequence  $x_k = 1 - 2^{-k}$  converges linearly to  $x^* = 1$ . Find  $k$  so that  $|e_k| < 10^{-16}$  if  $e_k = x_k - x^*$  is the error.
  - (b) The sequence  $x_k = 1 - 2^{(-2^k)}$  converges quadratically to  $x^* = 1$ . Find  $k$  so that  $|e_k| < 10^{-16}$ . Find the smallest  $C$  so that  $|e_{k+1}| \leq C|e_k|^2$  for all  $k$ .
  - (c) Let  $F(x) = \cos(x-1) - \exp(1-x)$  and  $x_0 = 0.5$ . Using PETSc, or any other tool, compute Newton iterates  $x_k$  for  $k = 1, \dots, 6$  to solve  $F(x) = 0$ . Estimate  $C$  so that  $|e_{k+1}| \leq C|e_k|^2$  for large  $k$ .
- 4.2. Modify `ecjac.c` to set the initial vector to  $\mathbf{x}_0 = [10 \ 10]^\top$ . Rerun it with option `-snes_monitor` and note it does not converge to a solution. Why does the SNES stop? Add one run-time option so that it converges, thereby reproducing Figure 4.3.
- 4.3. Running with `-snes_monitor_solution` will show the Newton iterates, but one might want to see more than the default 6-digit values. One way to do this is to add a *monitor* via `SNESMonitorSet()`. Modify `ecjac.c` by adding a monitor which shows 16-digit values of the solution at each SNES iteration. Thereby observe that the number of correct digits doubles in the Newton iteration.
- 4.4. The original Newton iteration (4.4), without line search, can diverge or enter a limit cycle.
- (a) By straightforward modifications of `expcircle.c`, write a code `atan.c` which solves nonlinear equation (4.1) with  $F(x) = \arctan(x)$  and initial iterate  $x_0 = 2.0$ . Observe that running it as `./atan -snes_fd -snes_linesearch_type basic` causes it to diverge, but the default choice `-snes_linesearch_type bt` gives convergence.
  - (b) Turning to paper calculation, consider the Newton iteration for solving  $\arctan(x) = 0$ . Set up an equation which requires the iteration to enter a sign-flipping limit cycle ( $x_{k+1} = -x_k$ ). Solving this equation yields the positive initial value  $x_0$  so that the Newton iteration makes no progress, and yet remains bounded, for all  $k$ . Sketch the situation.
  - (c) By modifying  $x_0$  in `atan.c` and using line-search type `basic`, confirm that one can make the Newton iteration “stick” in this actually unstable limit cycle for quite a while. Then confirm that line-search types `bt, l2, cp` all get unstuck immediately.

- 4.5. Modify `reaction.c` to a new code `bratu1D.c` which solves the *Liouville-Bratu equation* [23, 106]

$$-u'' - \lambda e^u = 0 \quad (4.28)$$

subject to Dirichlet boundary conditions  $u(0) = u(1) = 0$ . Make  $\lambda$  an adjustable parameter via `PetscOptionsReal()`. Confirm by using fine grids, and option `-snes_converged_reason`, plus other options as needed, that around  $\tilde{\lambda} = 3.513$  there is a transition to nonconvergence of the Newton iteration. The critical parameter value  $\tilde{\lambda}$ , which should be robust across sufficiently fine grids, is intrinsic to the nonlinear PDE problem [42]. Nonconvergence for  $\lambda > \tilde{\lambda}$  is not merely a Newton iteration failure, although that is indeed the symptom at run time.

- 4.6. Because one should not walk on high wires too often without a net, it is worthwhile finding exact solutions even for hard-to-solve nonlinear PDEs. One can usually “manufacture” [129, 153] an exact solution to a generalized form of the problem (if not the original form). For instance, for equation (4.11) with boundary values  $u(0) = u(1) = 0$ , one can choose  $u(x) = \sin(\pi x)$  as the exact solution. Then we determine  $f(x)$  from the differential equation. Add such a manufactured exact solution to `bratu1D.c` (previous exercise). Confirm that we have both quadratic convergence of the Newton iteration on fixed grids and  $O(h^2)$  convergence of the numerical error under grid refinement when the code is verified using the exact solution. (*This verification procedure gives confidence in purely numerical results like those in the previous exercise.*)
- 4.7. As long as the number of MPI processes does not exceed the number of grid points, then `reaction.c` will work correctly, though perhaps not efficiently, in parallel. Confirm this by running

```
| $ mpiexec -n N ./reaction -snes_converged_reason -da_refine M
```

with a few values of `N` and `M`. However, `ecjac.c` does not run correctly even on two MPI processes. Modify it so that it does.

# Chapter 5

# Time-stepping

We now restart our introduction to PETSC and PDEs with a well-known topic: numerical solutions to ordinary differential equation (ODE) initial value problems. Because we have practice with PETSC objects, many aspects of solving ODE systems will be straightforward. Nonetheless, a review of time-stepping methods is appropriate, including the basics of both explicit and implicit methods, after which we will transition to numerical methods for time-dependent PDEs.

Three examples are solved in this chapter:

- a linear system of two ODEs, i.e., an entry-level problem;
- an arbitrarily large system of ODEs arising from a spatial (2D) finite difference discretization—this is the *method of lines*—of the time-dependent heat equation (PDE);
- and a similar approach on a pair of coupled, nonlinear PDEs forming a 2D reaction/diffusion model with two chemical species.

These problems are solved using PETSC TS time-stepping objects [1, 10]. In the PDE examples, both of which use DMDA structured grids (Chapter 3), the user writes code which discretizes in space, thus generating an ODE system, but then the time integration method is determined only at run time. If an implicit time-stepping method is chosen then the full stack of SNES/KSP/PC solver tools is used for the (generally) nonlinear equations at each time step, thus we will be exploiting our experience with PETSC solvers from Chapters 2–4. Though numerical ODE solvers are at a relatively mature stage of development compared to PDE methods, and often treated as black boxes, aspects of the run-time control of their PETSC implementations deserve discussion.

## Systems of ordinary differential equations (ODEs)

Consider an ODE system in form

$$\mathbf{y}' = \mathbf{g}(t, \mathbf{y}), \quad (5.1)$$

where  $\mathbf{y}(t) \in \mathbb{R}^N$  and  $\mathbf{y}' = d\mathbf{y}/dt$ . Suppose an *initial value* is also given:

$$\mathbf{y}(t_0) = \mathbf{y}_0. \quad (5.2)$$

The general purpose of such ODE *initial value problems* (IVPs) is to make predictions forward or backward from the initial time  $t_0$ . That is, there should be a unique solution  $\mathbf{y}(t)$  to (5.1) and (5.2).

Under reasonable assumptions about the behavior of the right-hand-side function in (5.1), the problem is indeed well posed, at least for short times. To be specific, assume that  $\mathbf{g}$  is continuous on a cylinder around the initial point  $(t_0, \mathbf{y}_0)$ ,

$$\mathcal{D} = \{(t, \mathbf{y}) : |t - t_0| \leq \delta, \|\mathbf{y} - \mathbf{y}_0\| \leq \omega\}, \quad (5.3)$$

where  $\delta > 0$  and  $\omega > 0$ , and assume further that  $\mathbf{g}$  is *Lipschitz* in its second argument, namely that there is  $L \geq 0$  so that

$$\|\mathbf{g}(t, \mathbf{y}_1) - \mathbf{g}(t, \mathbf{y}_0)\| \leq L\|\mathbf{y}_1 - \mathbf{y}_0\| \quad (5.4)$$

for all  $(t, \mathbf{y}_i) \in \mathcal{D}$ . Then the problem (5.1), (5.2) has a unique continuously differentiable solution  $\mathbf{y}(t)$  on an interval  $|t - t_0| < \epsilon$ , for some  $0 < \epsilon \leq \delta$ , and this solution depends continuously on the initial value  $\mathbf{y}_0$  [82, section 17.5]. Note that the solution may only exist on a shorter interval than the one used to define  $\mathcal{D}$  in (5.3).

The above brief attention to well-posedness is motivated by the following practical observation: When we run a numerical ODE code it *will* produce numbers! These numbers are never the exact solution of the ODE system, but they are “correct” in a numerical-analysis sense if we can demonstrate convergence to the solution of the continuum problem. That is, we accept wrong (i.e., not exact) numbers in cases where convergent methods are correctly applied to well-posed continuum problems. On the other hand, numbers which approximate no continuum solution at all are meaningless and should not be accepted.

Benign-looking scalar nonlinear ODEs can put us in peril of generating such nonsense. For example, Exercise 5.1 considers a well-known scalar ODE problem in which the solution ceases to exist after a finite interval of time. A code will sail past the end of this interval of time, producing numbers that are infinitely erroneous; they approximate no continuum solution at all. Furthermore, this concern arises even for linear PDE problems, for instance in the sense that one should only integrate forward in time for the heat equation. Nonlinear PDEs are even worse because there may be no solution at all, either forward or backward in time. In summary, when possible we will consider well-posedness prior to starting numerical solutions.

## Numerical methods for ODE initial value problems

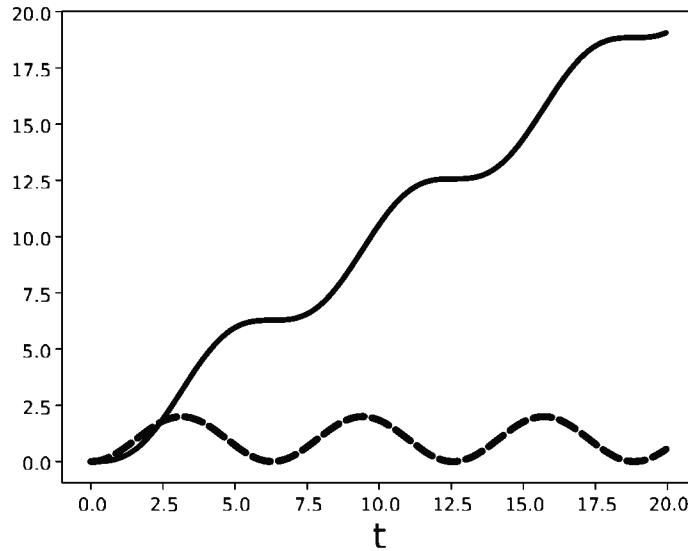
For *linear* ODE systems like the following, unique solutions exist for all time.

**Example 5.1.** Consider the initial value problem

$$\mathbf{y}' = A\mathbf{y} + \mathbf{f}(t), \quad \mathbf{y}(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad (5.5)$$

where  $A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$  and  $\mathbf{f}(t) = \begin{bmatrix} 0 \\ t \end{bmatrix}$ . The exact solution is shown in Figure 5.1:  $\mathbf{y}(t) = \begin{bmatrix} y_0(t) \\ y_1(t) \end{bmatrix} = \begin{bmatrix} t - \sin t \\ 1 - \cos t \end{bmatrix}$ .

This example ODE system can be approximated by almost any time-stepping numerical method. Such methods start from the initial values and use the right-hand-side  $g(t, \mathbf{y})$ , i.e., the slopes defining the ODE, to generate a sequence  $\{\mathbf{Y}_\ell\}$  which approximates the solution at discrete times  $t_\ell$ . The goal, of course, is that  $\mathbf{Y}_\ell \approx \mathbf{y}(t_\ell)$ . In more detail, let  $t_0 < t_1 < t_2 < \dots < t_L$  be a finite sequence of times, with steps  $\Delta t_\ell = t_\ell - t_{\ell-1}$  for  $\ell = 1, 2, \dots, L$ . Let  $\mathbf{Y}_0 = \mathbf{y}_0$  be the (known) initial value (5.2).



**Figure 5.1.** The solution to (5.5):  $y_0(t)$  (solid) and  $y_1(t)$  (dashed).

*Euler's method*, also known as the *forward Euler method*, generates  $\mathbf{Y}_1, \mathbf{Y}_2, \dots, \mathbf{Y}_L$  from

$$\frac{\mathbf{Y}_\ell - \mathbf{Y}_{\ell-1}}{\Delta t_\ell} = \mathbf{g}(t_{\ell-1}, \mathbf{Y}_{\ell-1}) \quad (5.6)$$

or, equivalently, by this update formula:

$$\mathbf{Y}_\ell = \mathbf{Y}_{\ell-1} + \Delta t_\ell \mathbf{g}(t_{\ell-1}, \mathbf{Y}_{\ell-1}). \quad (5.7)$$

The *backward Euler method* is just as easy to state,

$$\frac{\mathbf{Y}_\ell - \mathbf{Y}_{\ell-1}}{\Delta t_\ell} = \mathbf{g}(t_\ell, \mathbf{Y}_\ell), \quad (5.8)$$

but much more work is required to use it in practice. Merely writing the analog of (5.7), namely a formula by which to compute  $\mathbf{Y}_\ell$ , requires doing algebra on the function  $\mathbf{g}$  which may not be possible; backward Euler is thus an *implicit* method. At each time step equation (5.8) generates a (generally) nonlinear algebraic system for the unknown values  $\mathbf{Y}_\ell$ , but this is exactly the kind of problem solved in Chapter 4 using SNES objects to do Newton's method.

Though they evaluate the right-hand-side function  $\mathbf{g}(t, \mathbf{y})$  at different points, the Euler methods (5.6) and (5.8) both use *first-order* finite difference approximations of the derivative of  $\mathbf{y}(t)$ . Recall that for any twice-differentiable function  $f(t)$ , by Taylor's theorem

$$\frac{f(t+h) - f(t)}{h} = f'(t) + O(h^1)$$

as  $h \rightarrow 0$ . The Euler methods thus have  $O(h^1)$  local truncation error  $\tau = O(h^1)$  [7], where  $\tau$  is defined as the residual from applying the scheme to the exact solution. A scheme is *consistent* if the local truncation error goes to zero as  $h \rightarrow 0$ , that is, by the mild condition  $\tau = o(h)$ .

Another meaning of “first-order accuracy” also applies to the Euler methods, namely that the approximations  $\mathbf{Y}_\ell$  converge to the values  $\mathbf{y}(t_\ell)$  as the time steps  $\Delta t_\ell$  go to zero, such that  $\|\mathbf{Y}_\ell - \mathbf{y}(t_\ell)\| \leq Ch^1$  for some constant  $C$ . (In this context  $h$  denotes a uniform bound on the

steps  $\Delta t_\ell$ .) Showing this property of a scheme, that is, its *convergence* at rate  $O(h^1)$ , requires more assumptions than  $O(h^1)$  local truncation error. One also needs to bound the rate at which errors build up over many steps, so the scheme must also have a *stability* property. We discuss *absolute stability* below, but much more on the stability of methods can be found in the references [7, 104].

For Euler's method, an elementary argument [7] suffices to show that the approximations converge at first order. One assumes that the solution  $\mathbf{y}(t)$  exists, that a Lipschitz bound (5.4) applies to  $\mathbf{g}(t, \mathbf{y})$ , and that the solution has bounded second derivative ( $\|\mathbf{y}(t)''\| \leq M$ ); these statements must all apply on some nontrivial interval  $[t_0, t_f]$ . Then

$$\|\mathbf{Y}_\ell - \mathbf{y}(t_\ell)\| \leq \frac{hM}{2L} (e^{L(t_\ell-t_0)} - 1). \quad (5.9)$$

However, this exponentially growing bound is often pessimistic.

## Higher-order and adaptive methods

The *order* of a method, the power  $p > 0$  for which  $\|\mathbf{Y}_\ell - \mathbf{y}(t_\ell)\| = O(h^p)$  as  $h \rightarrow 0$ , tells us what to expect when we shorten the time step. For example, halving the time step in a first-order method reduces the numerical error by half, but for a second-order method by a factor of four.

By definition, *one-step* methods use  $\mathbf{Y}_{\ell-1}$  only, and not previous values, to build the next approximation  $\mathbf{Y}_\ell$ . However, one-step methods generally use multiple evaluations of the right-hand-side  $\mathbf{g}$ , i.e., they have multiple *stages*, and a method of order  $p$  must have at least  $p$  stages. Among one-step methods the *Runge-Kutta* (RK) family [31] is best known, and we will try several RK schemes.

The *explicit trapezoidal* method [7] is a one-step, two-stage, second-order RK method, denoted RK2a. It takes a forward Euler step but then recomputes the step using an average of slopes:

$$\begin{aligned} \hat{\mathbf{Y}} &= \mathbf{Y}_{\ell-1} + h \mathbf{g}(t_{\ell-1}, \mathbf{Y}_{\ell-1}), \\ \mathbf{Y}_\ell &= \mathbf{Y}_{\ell-1} + \frac{h}{2} \mathbf{g}(t_{\ell-1}, \mathbf{Y}_{\ell-1}) + \frac{h}{2} \mathbf{g}(t_\ell, \hat{\mathbf{Y}}). \end{aligned} \quad (5.10)$$

The well-known fourth-order Runge-Kutta method RK4 has four stages. Instead of stating this particular scheme using formulas like (5.10) above, consider tabular notation for any one-step,  $s$ -stage Runge-Kutta method. The formulas

$$\begin{aligned} \hat{\mathbf{Y}}_i &= \mathbf{Y}_{\ell-1} + h \sum_{j=1}^s a_{ij} \mathbf{g}(t_{\ell-1} + c_j h, \hat{\mathbf{Y}}_j), \quad 1 \leq i \leq s, \\ \mathbf{Y}_\ell &= \mathbf{Y}_{\ell-1} + h \sum_{i=1}^s b_i \mathbf{g}(t_{\ell-1} + c_i h, \hat{\mathbf{Y}}_i) \end{aligned} \quad (5.11)$$

correspond to the *tableau* [31]

$c_1$	$a_{11}$	$a_{12}$	$\cdots$	$a_{1s}$
$c_2$	$a_{21}$	$a_{22}$	$\cdots$	$a_{2s}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$c_s$	$a_{s1}$	$a_{s2}$	$\cdots$	$a_{ss}$
	$b_1$	$b_2$	$\cdots$	$b_s$

**Table 5.1.** Tableau for the explicit trapezoidal rule (RK2a; left), the classical fourth-order Runge-Kutta method (RK4; middle), and an embedded four-stage, third-order scheme (RK3bs, the PETSC default; right).

0	0	0
1	$\frac{1}{2}$	$\frac{1}{2}$
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\frac{1}{2}$	0	$\frac{1}{2}$
1	0	1
$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$
$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{6}$

0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0
$\frac{3}{4}$	0	$\frac{3}{4}$	$\frac{1}{2}$	$\frac{1}{2}$
1	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	$\frac{3}{4}$
$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	0	$\frac{1}{2}$
$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{8}$	$\frac{1}{9}$

For example, the  $s = 2$  method RK2a is the left tableau in Table 5.1, and the  $s = 4$  method RK4 is in the middle. Blanks in the tableau correspond to zeros. In all three methods in Table 5.1,  $\hat{\mathbf{Y}}_1 = \mathbf{Y}_{\ell-1}$  so the first rows are trivial. Also, because these schemes are explicit, nonzero values  $a_{ij}$  only appear strictly below the diagonal; we return to this idea below.

Most ODE schemes in PETSC, including the default type for TS time-stepping objects, are *adaptive*. That is, they adjust the time step using information from the already computed values, taking shorter steps when  $\mathbf{g}$  is rapidly changing or irregular, and longer steps when it is smoother and thus more predictable. Adaptive strategies require error estimation and control [7], actions internal to the TS object.

One adaptation strategy is to compare results from an *embedded* pair of methods. These two methods will have the same  $c_j$  and  $a_{ij}$  coefficients in their tableau, but different  $b_j$  and different orders of accuracy. The default TS scheme RK3bs [17] is the embedded pair shown at right in Table 5.1; note the two rows of  $b_j$  coefficients below the line. In RK3bs one method (first row) makes  $O(h^3)$  local truncation errors, while the other (second row) makes  $O(h^2)$  errors. The difference between the results is used to estimate the accuracy of the lower-order scheme [31]; the details are beyond our scope. The work of computing two approximations is nearly the same cost as just one, because they share the same evaluations of  $\mathbf{g}$ .

There are also *multistep* alternatives, for example the implicit, second-order *backwards differentiation formula* (BDF2),

$$\mathbf{Y}_\ell = \frac{4}{3}\mathbf{Y}_{\ell-1} - \frac{1}{3}\mathbf{Y}_{\ell-2} + \frac{2}{3}h\mathbf{g}(t_\ell, \mathbf{Y}_\ell), \quad (5.12)$$

which uses two previous values, and there are usable BDF formulas for orders 3–6 [7]; we compare BDF and other formulas on the heat equation later in this chapter. Other schemes such as the *general linear methods* [31] combine the multistage and multistep ideas.

When comparing the performance of methods one observes for explicit schemes that the computational cost of a time step is essentially proportional to the number of evaluations of the right-hand-side  $\mathbf{g}$ . (The additional arithmetic in a particular scheme, represented by the number of nonzero coefficients in a tableau, for example, is relatively small.) Implicit schemes, however, require solving the algebraic systems that arise at each step. This imposes a large additional cost, including both the costs of solving linear systems and of additional evaluations of  $\mathbf{g}$  or its derivatives, e.g., as needed in a Newton iteration. Implicit schemes thus have more expensive steps, so the question is whether one may use significantly fewer steps by taking longer steps. This question relates fundamentally to scheme stability, not order, and so we consider explicit/implicit comparisons later in this chapter in terms of absolute stability.

A large number of methods are available at the PETSC command line. Choosing the “best” one for a given ODE system may not even be a reasonable goal, so our goal in this chapter is only to suggest reasonable choices for broad classes of problems.

## A first TS example

The linear ODE system in Example 5.1 is a good starting point for our first ODE-solving program. It has fixed dimension and an exact solution from which to evaluate numerical error. Program `c/ch5/ode.c`, shown in Codes 5.1 and 5.2, solves this problem.

Consider how the TS object is initialized:

```
TSCreate(PETSC_COMM_WORLD,&ts);
TSSetProblemType(ts,TS_NONLINEAR);
TSSetRHSFunction(ts,NULL,FormRHSFunction,NULL);
TSSetType(ts,TSRK);
```

By setting the problem type to `TS_NONLINEAR` we are saying that the problem is in form (5.1) with a potentially nonlinear function  $\mathbf{g}(t, \mathbf{y})$ , even though our particular function is linear. Calling `TSSetRHSFunction()` sets a call-back to `FormRHSFunction()` which evaluates  $\mathbf{g}(t, \mathbf{y})$ ; see Code 5.2. Next we set the TS type to be the Runge-Kutta family; this choice can be overridden by run-time option `-ts_type` (more below).

Additional commands configure the time axis of the TS:

```
TSSetTime(ts,t0);
TSSetMaxTime(ts,tf);
TSSetTimeStep(ts,dt);
TSSetExactFinalTime(ts,TS_EXACTFINALTIME_MATCHSTEP);
TSSetFromOptions(ts);
```

The first three of these commands imply that the interval of integration is  $[t_0, t_f]$  and that, if the method were *not* adaptive, the discrete times would be from the list `t0:dt:tf` (MATLAB notation). However, for adaptive methods `dt` is only the initial time step. The `TSSetExactFinalTime()` command makes sure that adaptive TS types respect the final time you just set. Finally, using `TSSetFromOptions()` means these choices can be overridden at the command line.

```
int main(int argc,char **argv) {
  PetscInt steps;
  PetscReal t0 = 0.0, tf = 20.0, dt = 0.1, err;
  Vec      y, yexact;
  TS       ts;

  PetscInitialize(&argc,&argv,(char*)0,help);

  VecCreate(PETSC_COMM_WORLD,&y);
  VecSetSizes(y,PETSC_DECIDE,2);
  VecSetFromOptions(y);
  VecDuplicate(y,&yexact);

  TSCreate(PETSC_COMM_WORLD,&ts);
  TSSetProblemType(ts,TS_NONLINEAR);
  TSSetRHSFunction(ts,NULL,FormRHSFunction,NULL);
  TSSetType(ts,TSRK);

  // set time axis
  TSSetTime(ts,t0);
  TSSetMaxTime(ts,tf);
  TSSetTimeStep(ts,dt);
```

```

TSSetExactFinalTime(ts ,TS_EXACTFINALTIME_MATCHSTEP) ;
TSSetFromOptions(ts ) ;

// set initial values and solve
TSGetTime(ts ,&t0) ;
ExactSolution(t0 ,y) ;
TSSolve(ts ,y) ;

// compute error and report
TSGetStepNumber(ts ,&steps) ;
TSGetTime(ts ,&tf) ;
ExactSolution(tf ,yexact) ;
VecAXPY(y,-1.0,yexact); // y <- y - yexact
VecNorm(y,NORM_INFINITY,&err) ;
PetscPrintf(PETSC_COMM_WORLD,
            "error at tf = %.3f with %d steps: |y-y_exact|_inf = %g\n",
            tf ,steps ,err) ;

VecDestroy(&y) ; VecDestroy(&yexact) ; TSDestroy(&ts) ;
return PetscFinalize() ;
}

```

**Code 5.1.** *c/ch5/ode.c, part I.* *ode.c creates and configures a TS object, then solves the problem and reports the numerical error.*

Because of run-time options, the `TSSetFromOptions()` call can cause the time axis to change. Thus we get the current time using `TSGetTime()`, set the initial values using that, and then call `TSSolve()`. After the solve we also get the time for computing the numerical error; it may not be the original `tf`.

Code 5.2 shows the functions that evaluate the exact solution and the right-hand-side  $\mathbf{g}(t, \mathbf{y})$  for ODE (5.5). For the latter, the input `Vec y` is read into a `const` array with `VecGetArrayRead()` while the output `Vec g` needs to be accessed through `VecGetArray()`.

```

PetscErrorCode ExactSolution(PetscReal t , Vec y) {
    PetscReal *ay;
    VecGetArray(y,&ay);
    ay[0] = t - PetscSinReal(t);
    ay[1] = 1.0 - PetscCosReal(t);
    VecRestoreArray(y,&ay);
    return 0;
}

PetscErrorCode FormRHSFunction(TS ts , PetscReal t , Vec y , Vec g ,
                               void *ptr) {
    const PetscReal *ay;
    PetscReal *ag;
    VecGetArrayRead(y,&ay);
    VecGetArray(g,&ag);
    ag[0] = ay[1]; // = g_1(t,y)
    ag[1] = - ay[0] + t; // = g_2(t,y)
    VecRestoreArrayRead(y,&ay);
    VecRestoreArray(g,&ag);
    return 0;
}

```

**Code 5.2.** *c/ch5/ode.c, part II.* *Evaluating the exact solution  $\mathbf{y}(t)$  and the right-hand-side  $\mathbf{g}(t, \mathbf{y})$ .*

Let us try it:

```
$ cd c/ch5/
$ make ode
$ ./ode -ts_monitor
0 TS dt 0.1 time 0.
1 TS dt 0.170141 time 0.1
2 TS dt 0.169917 time 0.270141
...
87 TS dt 0.161127 time 19.8389
88 TS dt 0.205214 time 20.
error at tf = 20.000 with 88 steps: |y-y_exact|_inf = 0.00928173
```

It is clear that the time-stepping method is adaptive and, given the small numerical error, that it has succeeded. But what solution method was used?

```
$ ./ode -ts_view
TS Object: 1 MPI processes
  type: rk
    RK type 3bs
  ...
maximum time=20.
total number of rejected steps=12
using relative error tolerance of 0.0001, ...
TSAdapt Object: 1 MPI processes
  type: basic
  ...
error at tf = 20.000 with 88 steps: |y-y_exact|_inf = 0.00928173
```

The default subtype for type TSRK is RK3bs (Table 5.1). Note that the adaptive method rejected 12 steps in addition to the 88 which were accepted.

## Controlling TS

As with other PETSC objects, there are many run-time options. To list them do

```
| $ ./ode -help |grep ts_
```

To see numerical values of the solution at each time step, do

```
| $ ./ode -ts_monitor -ts_monitor_solution
```

For a run-time graphical (line-graph) view of the solution:

```
| $ ./ode -ts_monitor_lg_solution -draw_pause 0.1
```

The result is a graphic similar to Figure 5.1.

If desired, option `-ts_monitor` can write the discretized  $t$  axis to a binary file, while option `-ts_monitor_solution` can write the solution values  $\{Y_\ell\}$ . Specifically, the Python script `c/ch5/plotTS.py`<sup>21</sup> takes, as input, such files:

---

<sup>21</sup>It needs copies of, or sym-links to, Python scripts `PetscBinaryIO.py` and `petsc_conf.py` from the `lib/petsc/bin/` directory of `$PETSC_DIR`. Try `make petscPyScripts`.

```
| $ ./ode -ts_monitor binary:t.dat -ts_monitor_solution binary:Y.dat
| $ ./plotTS.py -o figure.png t.dat Y.dat
```

The result is again Figure 5.1.

We are free to adjust the start time, end time, and initial time step at the command line:

```
| $ ./ode -ts_init_time 1.0 -ts_max_time 2.0 -ts_dt 0.001 -ts_monitor
```

Turning off adaptive time-stepping and setting the time step to  $h = 0.1$  yields 200 steps:

```
| $ ./ode -ts_adapt_type none -ts_monitor -ts_dt 0.1
```

We can also change the relative and absolute tolerances, used in adaptive time-stepping, from their default values of  $10^{-4}$ :

```
| $ ./ode -ts_rtol 1.0e-1
| $ ./ode -ts_rtol 1.0e-6
| $ ./ode -ts_rtol 1.0e-6 -ts_atol 1.0e-6
```

These runs take 19, 117, and 392 steps, respectively, and the final numerical errors are 1.4,  $3.6 \times 10^{-3}$ , and  $1.3 \times 10^{-4}$ . Thus one observes that

*Setting -ts\_rtol and/or -ts\_atol does not cause the final numerical error to be bounded by those values.*

In fact the numerical error accumulates with steps, and can grow exponentially, while the tolerances can only control per-step errors.

Various TS methods can be compared on this ODE example—see Exercises 5.2 and 5.3—but we mostly defer comparisons until after introducing the concepts of stiffness and stability. However, note that when an RK method is chosen (`-ts_type rk`) then the particular flavor is assigned with `-ts_rk_type`:

```
| $ ./ode -ts_type rk -help |grep ts_rk
|   -ts_rk_type <3bs> ... 8vr 7vr 6vr 5bs 5dp 5f 4 3bs 3 2a 1fe ...
```

## Implicit methods and stiffness

The RK methods mentioned above, including the forward Euler method, are explicit. They compute  $\mathbf{Y}_\ell$  from previous values without needing to solve any equations. *Implicit* methods, by contrast, require solving, at each time step, the kind of generally nonlinear systems of equations addressed in Chapter 4. That is, given values  $\mathbf{Y}_{\ell-1}$ , for time  $t_\ell$  an implicit scheme generates a nonlinear system

$$\mathbf{F}(\mathbf{Y}_\ell) = \mathbf{0}. \quad (5.13)$$

For example, backward Euler (5.8) corresponds to the function  $\mathbf{F}(\mathbf{X}) = \mathbf{X} - \mathbf{Y}_{\ell-1} - hg(t_\ell, \mathbf{X})$  while BDF2 (5.12) yields  $\mathbf{F}(\mathbf{X}) = \mathbf{X} - \frac{4}{3}\mathbf{Y}_{\ell-1} + \frac{1}{3}\mathbf{Y}_{\ell-2} - \frac{2}{3}hg(t_\ell, \mathbf{X})$ . If we solve equations (5.13) by Newton's method, for the  $N$  real components of  $\mathbf{Y}_\ell$ , then we will need to differentiate  $\mathbf{F}$ , and thus  $g$ , either exactly or by finite differences. An implicit method will generally evaluate  $g$  a variable number of times per time step, depending on the number of Newton steps.

The following so-called *theta* methods are implicit when  $0 < \theta \leq 1$ :

$$\mathbf{Y}_\ell = \mathbf{Y}_{\ell-1} + (1 - \theta)hg(t_{\ell-1}, \mathbf{Y}_{\ell-1}) + \theta hg(t_\ell, \mathbf{Y}_\ell), \quad (5.14)$$

These are, in fact, Runge-Kutta methods with tableau

0		
1	$1 - \theta$	$\theta$
	$1 - \theta$	$\theta$

The  $\theta = 0$  case is the explicit forward Euler method (5.7), while  $\theta = 1$  gives the backward Euler method (5.8); these have first-order local truncation error. The  $\theta = 1/2$  case, the (implicit) trapezoid method, has second-order  $O(h^2)$  local truncation error. This method is called the *Crank-Nicolson* (CN) method in PETSc, and when used on a PDE problem [115].

Both the RK2a rule (5.10) and the CN method are second order. Why would one ever use the latter scheme, an implicit method which requires solving a system of algebraic equations at each time step? The answer is that the CN method is more *stable* than RK2a, and it therefore may allow much longer, and thus significantly fewer, time steps.

Though there are several useful definitions of stability, we focus below on *absolute stability*. Most explicit schemes are conditionally absolutely stable, thus they give good results when used with sufficiently small time steps, but the length of stable time steps depends on the particular ODE. The computational burden of many small steps may be avoidable in some cases, but there is always a trade-off between many inexpensive explicit time steps versus expensively solving equations at each time step in an implicit scheme. Some ODE solutions, with certain accuracy goals, can succeed if computed implicitly while being too costly when done by explicit methods. This is characteristic of parabolic PDEs, at least for fine spatial grids, as explored later in this chapter.

Continuing in this informal vein, the property of certain ODE problems which drives our concern with scheme stability is the “stiffness” of those problems. Though only an informal definition is possible [7], an ODE system is *stiff* if it has short timescales which do not need to be accurately resolved in order to compute the final-time result at the desired accuracy, but which cause explicit schemes to take small time steps.

**Example 5.2.** Consider the linear ODE IVP, in  $N = 3$  dimensions,

$$\mathbf{y}' = B\mathbf{y}, \quad \mathbf{y}(0) = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad (5.15)$$

where

$$B = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0.1 \\ 0 & 0 & -101 \end{bmatrix}. \quad (5.16)$$

The exact solution at a particular time, say  $t_f = 10$ , can be found by exponentiating a matrix (Exercise 5.4). To an absolute accuracy of  $10^{-7}$ ,

$$\mathbf{y}(t_f) = e^{Bt_f} \mathbf{y}(0) = \begin{bmatrix} -1.383623 \\ -0.295886 \\ 0 \end{bmatrix}. \quad (5.17)$$

The eigenvalue  $\lambda_{\min} = -101$  of  $B$  (Exercise 5.4) corresponds to a short timescale and a rapid decay rate. Precisely following this fast timescale is not needed to produce absolutely accurate values for  $\mathbf{y}(t_f)$ .

For the ODE in Example 5.2 we consider the two  $O(h^2)$  methods introduced above, the explicit (RK2a) and implicit (Crank-Nicolson) trapezoid rules. A modification of `ode.c` called

`stiff.c` (Exercise 5.5) solves this problem. Starting with the explicit RK2a method, we take 200 steps from  $t_0 = 0$  to  $t_f = 10$ :

```
$ ./stiff -ts_type rk -ts_rk_type 2a -ts_adapt_type none -ts_dt 0.05
Vec Object: 1 MPI processes
  type: seq
  8.08433e+182
  -8.16517e+184
  8.24763e+187
total steps = 200
```

This is clearly nonsense; the solution has exploded. Using 1000 steps gives an answer which instead has (roughly) three-digit accuracy:

```
$ ./stiff -ts_type rk -ts_rk_type 2a -ts_adapt_type none -ts_dt 0.01
Vec Object: 1 MPI processes
  type: seq
  -1.38367
  -0.295656
  1.03141e-301
total steps = 1000
```

By contrast, in 200 steps of Crank-Nicolson we get two-digit accuracy:

```
$ ./stiff -ts_type cn -ts_adapt_type none -ts_dt 0.05
Vec Object: 1 MPI processes
  type: seq
  -1.383
  -0.298767
  1.6678e-73
total steps = 200
```

The effect caused by the issue in Example 5.2, namely the need for many short time steps for the explicit scheme, and the potential for lower computational cost from an implicit scheme, becomes more pronounced as the dimension of the problem increases and the condition number of the matrix increases. On an example like this, explicit schemes are sensitive to the range of eigenvalue magnitudes. Here  $|\lambda_{\min}|$  is large even though the size of the solution component for that eigenvalue, i.e.,  $e^{\lambda_{\min} t}$ , is tiny. For a stiff problem an explicit scheme exhibits the “symptom” that the number of steps is both large and insensitive to the accuracy goal unless very high accuracy is needed (Exercise 5.5).

Stiffness is supposed to be a property of the problem not the method. However, there is no precise way to decide if a particular ODE system, by itself, is stiff. A problem may be stiff for one accuracy goal, and not for others, or stiff only for long intervals of integration. Thus the word “stiff” is like the word “sparse.” Namely, to use such words precisely one must supply an “enough so that” criterion.

## Absolute stability

The so-called *test equation* is a simple scalar ODE IVP:

$$y' = \lambda y, \quad y(0) = 1, \tag{5.18}$$

with solution  $y(t) = e^{\lambda t}$ . We say that a numerical method with step size  $h$  is *absolutely stable* for a given  $\lambda \in \mathbb{C}$  if it generates iterates  $\{Y_\ell\}$  from equation (5.18) with the property that the

absolute values are nonincreasing,  $|Y_\ell| \leq |Y_{\ell-1}|$ . Noting that the magnitude of the solution  $|y(t)| = e^{(\operatorname{Re} \lambda)t}$  is nonincreasing when  $\operatorname{Re} \lambda \leq 0$ , we clearly want the iterates to have the same property.

What does a numerical solution look like when absolute stability fails? Consider the test problem with  $\lambda = -1$ . With  $h = 0.5$  and  $h = 2.2$ , the (explicit) Euler method gives results shown in Figure 5.2. The exact solution is a decaying exponential, and a sufficiently small step size (e.g.,  $h = 0.5$ ) gives a qualitatively correct result. However, for the larger step size ( $h = 2.2$ ) the method generates a sequence  $\{Y_\ell\}$  which grows in magnitude, clearly wrong. For  $h = 2.2$  the RK2a method *also* yields a growing-magnitude solution (not shown).

On the other hand, iterates should increase in magnitude if the exact solution is actually growing in magnitude. That is, if  $\operatorname{Re} \lambda > 0$  then the numerical scheme should generate exponentially growing iterates when applied to (5.18), so absolute stability is not a desirable scheme property when  $\operatorname{Re} \lambda > 0$ .

The definition of absolute stability is not usually treated as dependent on the separate parameters  $\lambda$  and  $h$ , but on the product  $z = h\lambda$ . In fact, applying any one-step method to the test equation gives

$$Y_\ell = f(z)Y_{\ell-1} \quad (5.19)$$

for some *stability function*  $f(z)$ . For example, applying the one-step methods so far to (5.18) gives these formulas (Exercise 5.6):

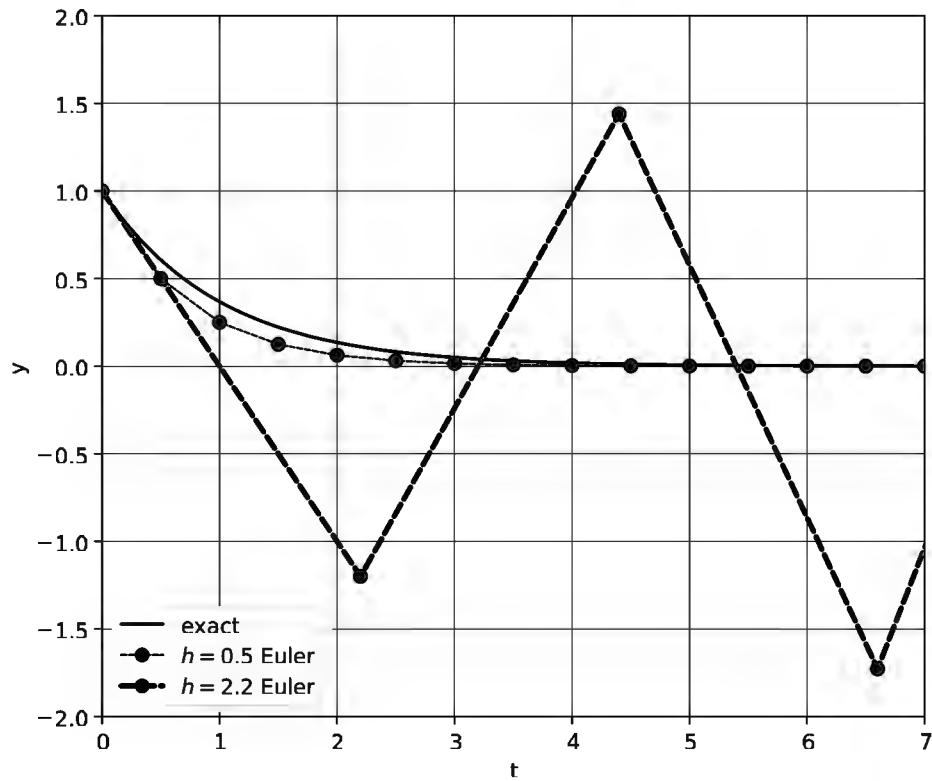
	Euler	$Y_\ell = (1 + z)Y_{\ell-1},$	(5.20)
	RK2a	$Y_\ell = \left(1 + z + \frac{1}{2}z^2\right)Y_{\ell-1},$	
	RK4	$Y_\ell = \left(1 + z + \frac{1}{2}z^2 + \frac{1}{3!}z^3 + \frac{1}{4!}z^4\right)Y_{\ell-1},$	
	Crank-Nicolson	$Y_\ell = \left(\frac{1 + \frac{1}{2}z}{1 - \frac{1}{2}z}\right)Y_{\ell-1},$	
	backward Euler	$Y_\ell = \left(\frac{1}{1 - z}\right)Y_{\ell-1}.$	

The stability function  $f(z)$  of a consistent scheme is a polynomial or rational approximation of the exponential:  $f(z) \approx e^z$  for small  $z$ .

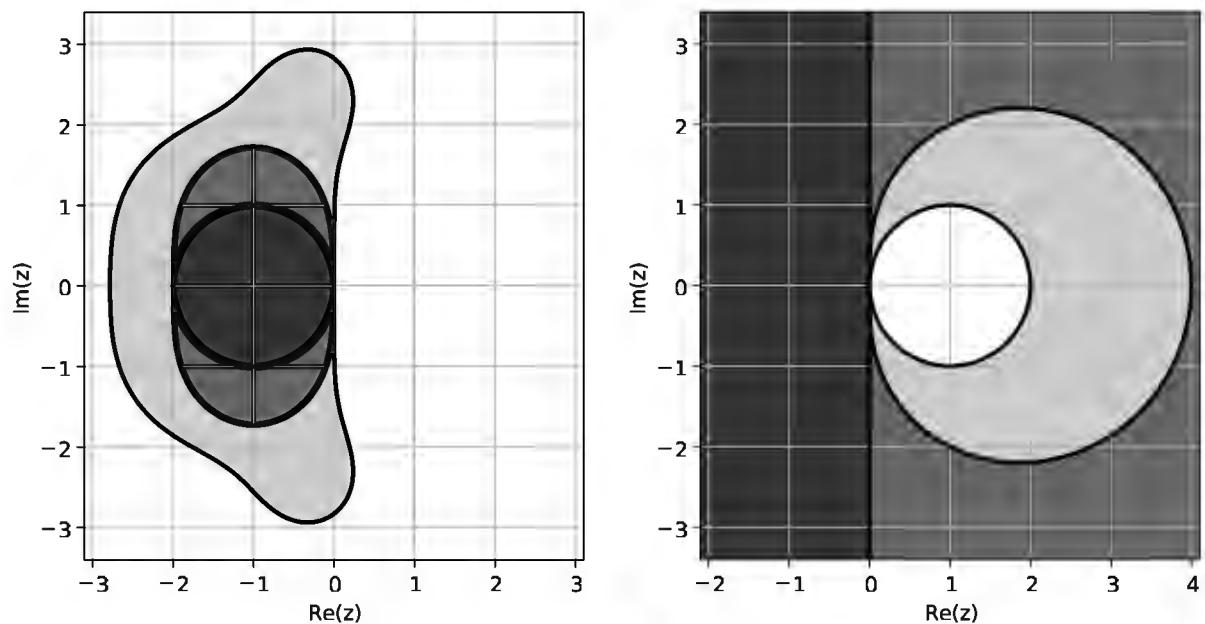
We may revise the definition slightly to say that a numerical method is *absolutely stable* for  $z \in \mathbb{C}$  if  $|Y_\ell| \leq |Y_{\ell-1}|$  holds when  $h\lambda = z$  [7, 125]. Thus absolute stability is equivalent to  $|f(z)| \leq 1$  for a one-step method. For example,  $f(z) = 1 + z$  for the Euler method. For the two runs shown in Figure 5.2, for which  $\lambda = -1$ , we see that  $z = h\lambda$  satisfies  $|1 + z| < 1$  when  $h = 0.5$  and  $|1 + z| > 1$  when  $h = 2.2$ . The *region of absolute stability* for a one-step scheme is the subset  $\{z : |f(z)| \leq 1\} \subset \mathbb{C}$ . For the schemes in (5.20), these regions are shown in Figure 5.3.

The absolute stability region of a multistep scheme is also determined by applying the scheme to the test equation, but the resulting recurrence is at least second order. Absolute stability holds only if all solutions of the recurrence have nongrowing magnitudes [7, 125]. Exercise 5.6 considers the BDF2 case (Figure 5.3), including a precise condition on the roots of the stability function.

By definition, a numerical scheme is *A-stable* if it is absolutely stable for all  $z$  in the left half-plane [7]. All of the implicit schemes in Figure 5.3, and also the  $\theta \geq 1/2$  theta methods (5.14), are A-stable. (However, one should not overgeneralize. The BDF schemes with order greater than 2 are not A-stable, though for orders 3–6 they remain well suited to stiff problems.)



**Figure 5.2.** Numerical solutions to test problem (5.18) with  $\lambda = -1$ . Large time steps  $h$  cause a failure of absolute stability for this explicit method.



**Figure 5.3.** Shaded regions of absolute stability for some ODE schemes. Explicit at left: Euler (darkest), RK2a (next), and RK4 (lightest). Implicit at right: backward Euler (lightest), BDF2 (next), and Crank-Nicolson (darkest).

On the other hand, no explicit methods are A-stable. When an explicit Runge-Kutta scheme is applied to the test equation one gets  $Y_\ell = p(z)Y_{\ell-1}$  for a nonconstant polynomial  $p(z)$ . Because  $|p(z)| \rightarrow \infty$  as  $|z| \rightarrow \infty$ , the absolute-stability region  $|p(z)| \leq 1$  is bounded; it cannot include the left half-plane.

Now consider, instead of the scalar test equation (5.18), a homogeneous linear system with a diagonalizable coefficient matrix:

$$\mathbf{y}' = A\mathbf{y}. \quad (5.21)$$

After diagonalizing  $A = X\Lambda X^{-1}$  one finds that  $e^{At} = Xe^{\Lambda t}X^{-1}$  so the solutions of (5.21) are of the form

$$\mathbf{y}(t) = Xe^{\Lambda t}X^{-1}\mathbf{y}(0). \quad (5.22)$$

Thus the components of  $\mathbf{w}(t) = X^{-1}\mathbf{y}(t)$  are decoupled, and they evolve by the (complex) exponential rates which appear in  $\Lambda$ :  $\mathbf{w}(t) = e^{\Lambda t}\mathbf{w}(0)$ . It follows that growth or decay of the solutions of (5.21) is determined by  $\operatorname{Re} \lambda$  for  $\lambda \in \sigma(A)$ .

These observations lead to our primary conclusion about the concept of absolute stability. Suppose that in (5.21) all eigenvalues have negative real parts,  $\operatorname{Re} \lambda < 0$ . The only way a numerical solution  $\{\mathbf{Y}_\ell\}$  can be close to an exact solution  $\mathbf{y}(t)$  is for the step size  $h$  to be chosen sufficiently small so that, for all  $\lambda \in \sigma(A)$ , the values  $z = h\lambda$  are in the absolute stability region  $R$  of the scheme. This condition should be seen as necessary, at best, but for such  $h$  the numerical solution is in qualitative agreement with the exact solution because both solutions eventually decay. In summary, we have this well-known principle for using numerical ODE solvers:

*When solving a linear system  $\mathbf{y}' = A\mathbf{y}$  by a scheme with absolute stability region  $R$ , choose  $h$  small enough so that  $z = h\lambda \in R$  for all eigenvalues  $\lambda \in \sigma(A)$  such that  $\operatorname{Re} \lambda \leq 0$ .*

In choosing among schemes, any desire for higher-order accuracy is thus constrained by the requirement of stability. For stiff systems, with a large spread of eigenvalue magnitudes, we may accept significant relative error for those eigenvalues which are in the left half-plane because these modes represent small absolute changes to an exact solution which contains faster-growing, or at least slower-decaying, modes. The essential requirement of absolute stability often dictates the choice of implicit methods, with their large stability regions, over higher-order explicit methods which are less expensive in a per-step sense.

An important distinction among implicit methods relates to their ability to follow decaying solutions in the presence of forcing terms. Consider the generalized test problem

$$y' = \lambda(y - \gamma(t)), \quad y(0) = 1, \quad (5.23)$$

for some bounded and continuous function  $\gamma(t)$ . The exact solution is asymptotic to  $\gamma(t)$  as  $\operatorname{Re}(\lambda) \rightarrow -\infty$  (Exercise 5.7). We say that a scheme has *stiff decay* [7] if, when applied to (5.23) for any such  $\gamma(t)$ , it yields approximations  $\{Y_\ell\}$  which approach  $\gamma(t)$  for every fixed  $t > 0$  in the following sense. Choose  $\ell \in \mathbb{Z}$ , define  $h = t/\ell$  so that  $Y_\ell \approx y(t)$ , and require that

$$|Y_\ell - \gamma(t)| \rightarrow 0 \quad \text{as} \quad h \operatorname{Re}(\lambda) = \operatorname{Re} z \rightarrow -\infty. \quad (5.24)$$

Thus a scheme has stiff decay if it can follow any nonhomogeneity  $\gamma(t)$  in the limit of a sufficiently negative exponential rate  $\lambda$ . Backward Euler (5.8) has stiff decay, as do the order 1–6 BDF schemes implemented in PETSc [7]. However, the A-stable Crank-Nicolson rule does not.

Note that BDF2 has three nice properties, namely second order, A-stability, and stiff decay, so it will be the default TS type for our time-dependent heat equation solver later in this chapter.

By way of a conclusion to this section, the need for stability is another numerical fact of life [141]:

*Fact 11. Stability is obligatory in a numerical scheme. You may seek greater accuracy, but exponential growth of the approximation to a bounded or decaying solution is never acceptable.*

## Jacobians for implicit methods

TS objects contain a SNES solver object inside because Newton's method may be needed to solve the nonlinear equations which arise at each time step of an implicit method. Two aspects of using Newton's method are then relevant, namely finding an initial iterate and providing or approximating the appropriate Jacobian. The first concern is (generally) not challenging because the last value of the solution can be used. That is, by default the implicit TS types choose  $\mathbf{Y}_\ell^{(0)} = \mathbf{Y}_{\ell-1}$  if  $\mathbf{Y}_\ell^{(k)}$  denotes the  $k$ th Newton iterate when solving (5.13).

On the other hand, we need the Jacobian derivative of  $\mathbf{F}$  in (5.13). As a representative example, the  $\theta$  method (5.14) corresponds to the residual function

$$\mathbf{F}(\mathbf{X}) = \mathbf{X} - \mathbf{Y}_{\ell-1} - (1 - \theta)hg(t_{\ell-1}, \mathbf{Y}_{\ell-1}) - \theta hg(t_\ell, \mathbf{X}), \quad (5.25)$$

where  $\mathbf{Y}_{\ell-1}$  is known from the previous time step. The Jacobian is then

$$J_{\mathbf{F}}(\mathbf{X}) = \frac{\partial \mathbf{F}}{\partial \mathbf{X}} = I - \theta h \frac{\partial \mathbf{g}}{\partial \mathbf{y}}(t_\ell, \mathbf{X}).$$

There is no need, however, to write Jacobian user code which is specific to the  $\theta$  method or any other method. Each implicit TS type knows how to convert user-supplied code for the Jacobian derivative of  $\mathbf{g}(t, \mathbf{y})$ , the right-hand side of the original ODE system (5.1), into the Jacobian of  $\mathbf{F}$  for Newton's method on (5.13). That is, instead of the Jacobian of  $\mathbf{F}$  the user supplies code for evaluating

$$\frac{\partial \mathbf{g}}{\partial \mathbf{y}} = \left[ \frac{\partial g_i}{\partial y_j} \right]. \quad (5.26)$$

As an illustration we have program `odejac.c`, built by small modifications of `ode.c` and excerpted below. Note we call `TSSetRHSJacobian()` in Code 5.3 to set a call-back to `FormRHSJacobian()`, which is shown in Code 5.4, and to set the TS type to Crank-Nicolson (CN). An allocated `Mat` called `J`, for holding a value of  $\partial \mathbf{g} / \partial \mathbf{y}$ , is assigned to both the Jacobian and preconditioner-material arguments of `TSSetRHSJacobian()`; compare usage of `SNESSetJacobian()` in Chapter 4.

```
MatCreate(PETSC_COMM_WORLD,&J);
MatSetSizes(J,PETSC_DECIDE,PETSC_DECIDE,2,2);
MatSetFromOptions(J);
MatSetUp(J);
TSSetRHSJacobian(ts,J,J,FormRHSJacobian,NULL);
TSSetType(ts,TSCN);
```

**Code 5.3.** `c/ch5/odejac.c`, part I. A `Mat` is created to hold  $\partial \mathbf{g} / \partial \mathbf{y}$ . The TS type is set to the implicit Crank-Nicolson method.

```

PetscErrorCode FormRHSJacobian(TS ts, PetscReal t, Vec y, Mat J, Mat P,
                                void *ptr) {
    PetscInt row[2] = {0, 1}, col[2] = {0, 1};
    PetscReal v[4] = { 0.0, 1.0,
                       -1.0, 0.0};
    MatSetValues(P, 2, row, 2, col, v, INSERT_VALUES);
    MatAssemblyBegin(P, MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(P, MAT_FINAL_ASSEMBLY);
    if (J != P) {
        MatAssemblyBegin(J, MAT_FINAL_ASSEMBLY);
        MatAssemblyEnd(J, MAT_FINAL_ASSEMBLY);
    }
    return 0;
}

```

**Code 5.4.** *c/ch5/odejac.c, part II.* *FormRHSJacobian()* computes the matrix  $\partial \mathbf{g} / \partial \mathbf{y}$  for Example 5.1.

So, how does nonadaptive, implicit CN compare to the default explicit, adaptive RK3bs method?

```

$ make odejac
$ ./odejac
error at tf = 20.000 with 200 steps: |y-y_exact|_inf = 0.0151358
$ ./odejac -ts_type rk
error at tf = 20.000 with 88 steps: |y-y_exact|_inf = 0.00928173

```

The former run evaluates the Jacobian at every time step,

```

$ ./odejac -log_view | grep TSJacobianEval
TSJacobianEval      200 1.0 ...

```

while the latter run, which is equivalent to the default case of *ode.c*, does not evaluate the Jacobian at all. Noting that the current example is small and nonstiff, in contrast to the next example we see no benefit here to the CN method.

In any case, the following three runs yield the same displayed results:

```

$ ./ode -ts_type cn -snes_fd
$ ./ode -ts_type cn -snes_mf
$ ./odejac

```

All of these call the SNES inside the TS to solve the system of equations (5.13) arising from setting up the CN (trapezoid rule) equations at each time step, but the latter two runs use a finite-differenced Jacobian.

Table 5.2 shows the TS types considered so far [10], including their (local truncation error) order. The backward Euler and CN methods are actually implemented as special cases of the  $\theta$  method:

```

$ ./odejac -ts_type theta -help |grep ts_theta
-ts_theta_theta <0.5 : 0.5>: Location of stage (0<Theta<=1) ...
-ts_theta_endpoint: <FALSE : FALSE> Use the endpoint instead ...
-ts_theta_initial_guess_extrapolate: <FALSE : FALSE> Extrapolate ...

```

**Table 5.2.** A selection of ODE integration methods; see allowed `-ts_type` choices for more. Schemes below the line are implicit.

Method	<code>-ts_type</code>	Order	Adaptive?
forward Euler (5.6)	euler	1	no
Runge-Kutta (5.11)	rk		
<code>-ts_rk_type 2a</code>		2	yes
<code>-ts_rk_type 3</code>		3	no
<code>-ts_rk_type 3bs [default]</code>		3	yes
<code>-ts_rk_type 4</code>		4	no
<code>-ts_rk_type 5bs</code>		5	yes
$\theta$ method (5.14)	theta		
<code>-ts_theta_theta</code> $\theta \in (0, 1]$		1 or 2	optional
backward Euler (5.8): $\theta = 1$	beuler	1	optional
Crank-Nicolson: $\theta = 1/2$	cn	2	optional
backward differentiation (5.12)	bdf		
<code>-ts_bdf_order X</code>		$X = 1, \dots, 6$	yes

Thus the following are equivalent:

```
| $ ./odejac -ts_type beuler
| $ ./odejac -ts_type theta -ts_theta_theta 1.0
```

as are

```
| $ ./odejac -ts_type cn
| $ ./odejac -ts_type theta -ts_theta_theta 0.5 -ts_theta_endpoint
```

These methods are optionally adaptive. For example,

```
| $ ./odejac -ts_type cn -ts_adapt_type basic
```

is an adaptive, though not embedded, form of Crank-Nicolson.

For the rest of this chapter we will use methods suitable for stiff and nonlinear ODE systems (5.1). That is, we will solve problems of the form  $\mathbf{y}' = \mathbf{g}(t, \mathbf{y})$  by implicit and semi-implicit methods which use the SNES inside the TS object.

## A time-dependent heat equation problem

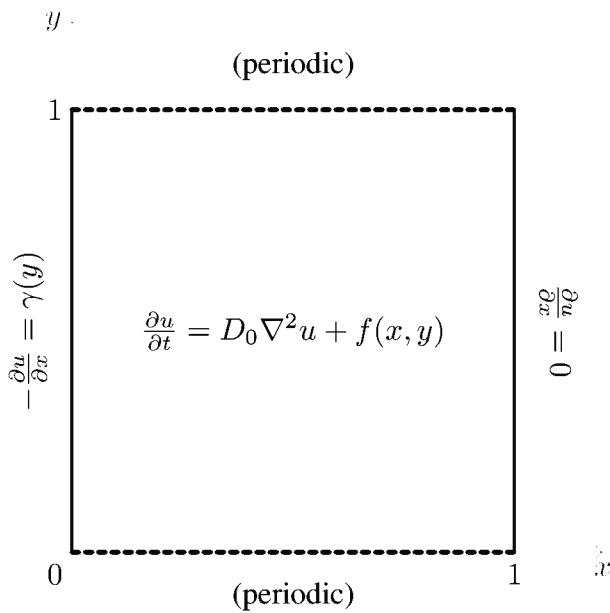
We now have the tools to solve many time-dependent PDEs. Such problems generate arbitrarily large systems of ODEs once the spatial derivatives are discretized. For example, consider this time-dependent heat equation in two space dimensions:

$$\frac{\partial u}{\partial t} = D_0 \nabla^2 u + f(x, y). \quad (5.27)$$

The solution is the temperature  $u(t, x, y)$  for  $t \in [0, T]$  and  $(x, y)$  in the unit square  $\mathcal{S} = (0, 1) \times (0, 1)$ ; see Figure 5.4. For simplicity we set the *diffusivity*  $D_0 > 0$  to be constant, while the *heat source*  $f(x, y)$  is time independent, and the initial temperature is zero:  $u(0, x, y) = 0$ .

To complete the specification of a concrete, well-posed [51] problem we choose a rather arbitrary source function which takes both positive and negative values,

$$f(x, y) = 3e^{-25(x-0.6)^2} \sin(2\pi y).$$



**Figure 5.4.** A time-dependent initial/boundary value problem on a square.

On the left ( $x = 0$ ) and right ( $x = 1$ ) sides we pick nonhomogeneous and homogeneous Neumann conditions, respectively, with

$$-\frac{\partial u}{\partial x} = \gamma(y) = \sin(6\pi y) \quad (5.28)$$

along the left side. For the sake of variety, periodic boundary conditions apply along the top and bottom boundaries.

We do not know the exact solution to the above problem, but a scalar quantity is conserved, namely the total thermal energy [119]. Recall that this is the integral of  $\rho c u$  over the domain if the material has density  $\rho$  and heat capacity  $c$ . (If the material also has constant thermal conductivity  $k$  then  $D_0 = k/(\rho c)$ .) The main idea here is that the integral of temperature  $u$  is conserved if the material properties are constant.

For our particular problem it is easy to verify (Exercise 5.8) that the sources integrate to zero,

$$\int_S f(x, y) dx dy = 0, \quad \int_0^1 \gamma(y) dy = 0, \quad (5.29)$$

and thus integrating equation (5.27) yields a constant total heat energy,

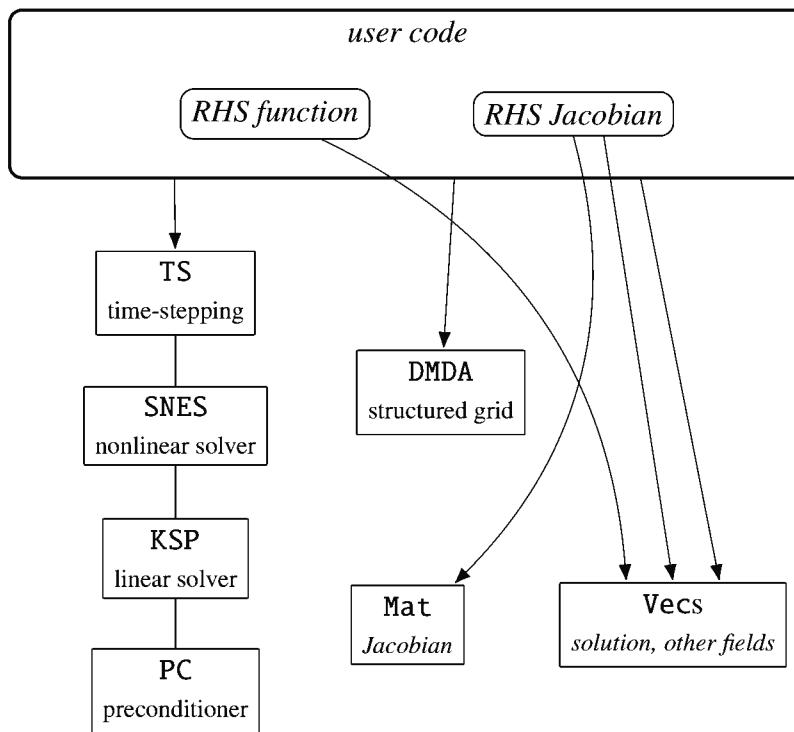
$$\frac{d}{dt} \left( \int_S u(t, x, y) dx dy \right) = D_0 \int_0^1 \gamma(y) dy + \int_S f(x, y) dx dy = 0. \quad (5.30)$$

Thus we will want our discretized equations to conserve thermal energy too.

## Method of lines

A time-dependent PDE like the heat equation becomes a system of ODEs after semidiscretization in space. This technique is called the *method of lines* [84, 104, 115], a name which imagines the continuous-time ODE domain as lines  $t \mapsto (t, x_i, y_j)$  in space-time.

For problem (5.27) we will discretize spatial derivatives just as in Chapter 3, using a grid of  $N = m_x \times m_y$  points  $(x_i, y_j)$ , with grid spacings  $h_x$  and  $h_y$ , and replacing the spatial derivatives



**Figure 5.5.** Component stack used for time-dependent, structured-grid PDE examples based on the method of lines. “RHS” stands for right-hand side.

in (5.27) by centered finite differences:

$$u'_{i,j} = D_0 \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} + D_0 \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_y^2} + f_{i,j}. \quad (5.31)$$

Here  $f_{i,j} = f(x_i, y_j)$  and  $u_{i,j}(t) \approx u(t, x_i, y_j)$ . Note that partial derivatives in  $t$  have become ordinary derivatives. Equations (5.31) form an ODE system  $\mathbf{y}' = \mathbf{g}(t, \mathbf{y})$ , which we will solve numerically. The method of lines strategy requires that our code define the right-hand-side  $\mathbf{g}$ , but there is no need to choose a time-stepping method just yet.

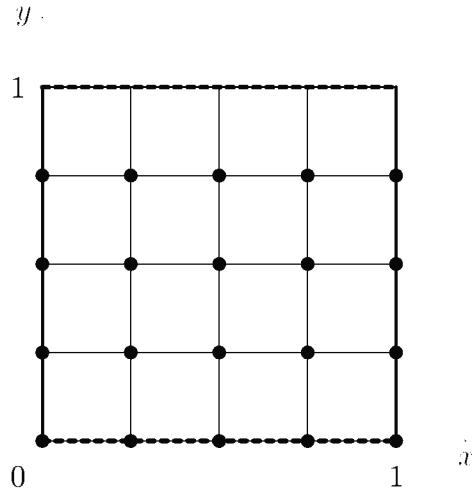
Program `c/ch5/heat.c`, extracted in Codes 5.5–5.9, implements (5.31). It has the structure shown in Figure 5.5, and it is easy to write based on our experience using DMDA in Chapters 3 and 4. Code 5.5, an extract of `main()`, sets up the DMDA object for the grid shown in Figure 5.6. By periodicity, grid points along the lower boundary  $y = 0$  are also (conceptually) located on the upper boundary  $y = 1$ . The default grid of  $m_x \times m_y = 5 \times 4$  points therefore has square cells; the spacings are  $h_x = 1/(m_x - 1)$  and  $h_y = 1/m_y$ , respectively, as computed by a helper function `Spacings()` (not shown).

```

DMDACreate2d(PETSC_COMM_WORLD,
    DM_BOUNDARY_NONE, DM_BOUNDARY_PERIODIC, DMDA_STENCIL_STAR,
    5,4,PETSC_DECIDE,PETSC_DECIDE, // default to hx=hx=0.25 grid
    1,1, // degrees of freedom, stencil width
    NULL,NULL,&da);
DMSetFromOptions(da);
DMSetUp(da);
DMCreateGlobalVector(da,&u);

```

**Code 5.5.** `c/ch5/heat.c`, part I. Set up the DMDA grid object and a `Vec`.



**Figure 5.6.** Because of periodicity in  $y$ , the default  $5 \times 4$  point grid in `heat.c` has square cells.

Code 5.6, another extract of `main()`, shows how we create and configure the TS. Through an association to the DMDA grid object, namely `TSSetDM()`, the TS knows the dimension  $N$  of the system of ODEs. We also set DMDA-based call-back functions; see Codes 5.7 and 5.8 below. Finally, because a stiff-decay method is appropriate for the stiff heat equation, we set the default TS type to BDF; the default order is 2.

```

TSCreate(PETSC_COMM_WORLD,&ts) ;
TSSetProblemType(ts ,TS_NONLINEAR) ;
TSSetDM(ts ,da) ;
TSSetApplicationContext(ts ,&user) ;
DMDATSSetRHSFunctionLocal(da ,INSERT_VALUES,
    (DMDATSRHSFunctionLocal)FormRHSFunctionLocal,&user) ;
DMDATSSetRHSJacobianLocal(da ,
    (DMDATSRHSJacobianLocal)FormRHSJacobianLocal,&user) ;
if (monitorenergy) {
    TSMonitorSet(ts ,EnergyMonitor,&user,NULL) ;
}
TSSetType(ts ,TSBDF) ;
TSSetTime(ts ,0.0) ;
TSSetMaxTime(ts ,0.1) ;
TSSetTimeStep(ts ,0.001) ;
TSSetExactFinalTime(ts ,TS_EXACTFINALTIME_MATCHSTEP) ;
TSSetFromOptions(ts ) ;

```

**Code 5.6.** `c/ch5/heat.c`, part II. Create the TS and associate the DMDA, RHS evaluation, Jacobian evaluation, and energy monitor to it.

`FormRHSFunctionLocal()` in Code 5.7 evaluates  $\mathbf{g}(t, \mathbf{y})$  in equation (5.31) and `FormRHSJacobianLocal()` in Code 5.8 evaluates its derivatives  $\partial\mathbf{g}/\partial\mathbf{y}$ . These functions use the same finite difference formula and 5-point stencil shown in Figure 3.5. Note that `heat.c` also defines functions `f_source(x,y)` and `gamma_neumann(y)` (not shown) which compute  $f$  and  $\gamma$  in (5.27).

```

PetscErrorCode FormRHSFunctionLocal(DMDALocalInfo *info,
                                    PetscReal t, PetscReal **au,
                                    PetscReal **aG, HeatCtx *user) {
  PetscInt i, j, mx = info->mx;
  PetscReal hx, hy, x, y, ul, ur, uxx, uyy;

  Spacings(info, &hx, &hy);
  for (j = info->ys; j < info->ys + info->ym; j++) {
    y = hy * j;
    for (i = info->xs; i < info->xs + info->xm; i++) {
      x = hx * i;
      // apply Neumann b.c.s
      ul = (i == 0) ? au[j][i+1] + 2.0 * hx * gamma_neumann(y)
                    : au[j][i-1];
      ur = (i == mx-1) ? au[j][i-1] : au[j][i+1];
      uxx = (ul - 2.0 * au[j][i] + ur) / (hx*hx);
      // DMDA is periodic in y
      uyy = (au[j-1][i] - 2.0 * au[j][i] + au[j+1][i]) / (hy*hy);
      aG[j][i] = user->D0 * (uxx + uyy) + f_source(x, y);
    }
  }
  return 0;
}

```

**Code 5.7.** *c/ch5/heat.c, part III. Evaluate RHS  $\mathbf{g}(t, \mathbf{y})$  for ODE system (5.31).*

A Neumann condition scheme is needed at the left and right boundaries. In detail, at each point  $(x_0, y_j) = (0, y_j)$  we have (5.28), which becomes

$$-\frac{u_{+1,j} - u_{-1,j}}{2h_x} = \gamma(y_j),$$

using a  $O(h_x^2)$  centered finite difference approximation and a notional value “ $u_{-1,j}$ ” [115]. Solving this for  $u_{-1,j}$ , namely  $u_{-1,j} = u_{+1,j} + 2h_x\gamma(y_j)$ , defines a version of the ODE RHS (5.31) at points  $(0, y_j)$  on the left boundary. The same technique also applies at the right boundary.

```

PetscErrorCode FormRHSJacobianLocal(DMDALocalInfo *info,
                                      PetscReal t, PetscReal **au,
                                      Mat J, Mat P, HeatCtx *user) {
  PetscInt i, j, ncols;
  const PetscReal D = user->D0;
  PetscReal hx, hy, hx2, hy2, v[5];
  MatStencil col[5], row;

  Spacings(info, &hx, &hy);
  hx2 = hx * hx; hy2 = hy * hy;
  for (j = info->ys; j < info->ys+info->ym; j++) {
    row.j = j; col[0].j = j;
    for (i = info->xs; i < info->xs+info->xm; i++) {
      // set up a standard 5-point stencil for the row
      row.i = i;
      col[0].i = i;
      v[0] = -2.0 * D * (1.0 / hx2 + 1.0 / hy2);
      col[1].j = j-1; col[1].i = i; v[1] = D / hy2;
      col[2].j = j+1; col[2].i = i; v[2] = D / hy2;
      col[3].j = j; col[3].i = i-1; v[3] = D / hx2;
      col[4].j = j; col[4].i = i+1; v[4] = D / hx2;
      ncols = 5;
      // if at the boundary, edit the row back to 4 nonzeros
    }
  }
}

```

```

    if (i == 0) {
        ncols = 4;
        col[3].j = j; col[3].i = i+1; v[3] = 2.0 * D / hx2;
    } else if (i == info->mx-1) {
        ncols = 4;
        col[3].j = j; col[3].i = i-1; v[3] = 2.0 * D / hx2;
    }
    MatSetValuesStencil(P,1,&row,ncols,col,v,INSERT_VALUES);
}
}

MatAssemblyBegin(P,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(P,MAT_FINAL_ASSEMBLY);
if (J != P) {
    MatAssemblyBegin(J,MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(J,MAT_FINAL_ASSEMBLY);
}
return 0;
}

```

**Code 5.8.** *c/ch5/heat.c, part IV. Evaluate the Jacobian of (5.31).*

Finally, a new feature of this example is a *monitor* function (call-back) which reports the total thermal energy as well as a ratio related to stability. We use `TSMonitorSet()` to set a call-back to `EnergyMonitor()`, shown in Code 5.9. Option `-ht_monitor` activates the new code, which supplements the usual `-ts_monitor` output. In this function the trapezoid rule is used to compute the local processor's contribution to the integral  $\int u dx dy$ , in variable `lenergy`, and then `MPI_Allreduce()` sums the integral over the entire grid.

The monitor also prints the value of  $\nu = D_0 \Delta t / (h_x h_y)$  where  $\Delta t$  is the current time step. Because  $D_0 / (h_x h_y) \sim |\lambda_{\min}|$  is comparable to the magnitude of the most negative eigenvalue of the discrete ODE system,  $\nu$  measures the stiffness of the ODE problem. Theorems showing that  $\nu \lesssim 1$  is either necessary or sufficient for conditional stability of fully discrete explicit methods for the heat equation can be found in [115]; additional discussion of this connection is in [104]. We will see that adaptive explicit methods generate  $\nu$  values less than one while implicit methods may take time steps  $\Delta t$  such that  $\nu \gg 1$ .

```

PetscErrorCode EnergyMonitor(TS ts, PetscInt step, PetscReal time, Vec u,
                           void *ctx) {
    HeatCtx      *user = (HeatCtx*) ctx;
    PetscReal     lenergy = 0.0, energy, dt, hx, hy, **au;
    PetscInt      i, j;
    MPI_Comm      com;
    DM            da;
    DMDALocalInfo info;

    TSGetDM(ts,&da);
    DMDAGetLocalInfo(da,&info);
    DMDAVecGetArrayRead(da,u,&au);
    for (j = info.ys; j < info.ys + info.ym; j++) {
        for (i = info.xs; i < info.xs + info.xm; i++) {
            if ((i == 0) || (i == info.mx-1))
                lenergy += 0.5 * au[j][i];
            else
                lenergy += au[j][i];
        }
    }
    DMDAVecRestoreArrayRead(da,u,&au);
    Spacings(&info,&hx,&hy);
}

```

```

lenergy *= hx * hy;
PetscObjectGetComm(( PetscObject)(da),&com);
MPI_Allreduce(&lenergy,&energy,1,MPIU_REAL,MPIU_SUM,com);
TSGetTimeStep(ts,&dt);
PetscPrintf(PETSC_COMM_WORLD," energy = %9.2e      nu = %8.4f\n",
            energy,user->D0*dt/(hx*hy));
return 0;
}

```

**Code 5.9.** *c/ch5/heat.c*, part V. Print the total thermal energy and the stability ratio  $\nu = D_0 \Delta t / (h_x h_y)$ .

## Visualization and performance

A helpful first run exposes the TS/SNES/KSP/PC stack:

```

$ make heat
$ ./heat -ts_view

```

This reveals that we are using the implicit BDF2 type for the TS (`-ts_type bdf` `-ts_bdf_order 2`), adaptive time-stepping (`-ts_adapt_type basic`), Newton's method with line search (`-snes_type newtonls`), a GMRES Krylov solver (`-ksp_type gmres`), and ILU as the preconditioner (`-pc_type ilu`).

One can understand the time-stepping by using both monitors:

```

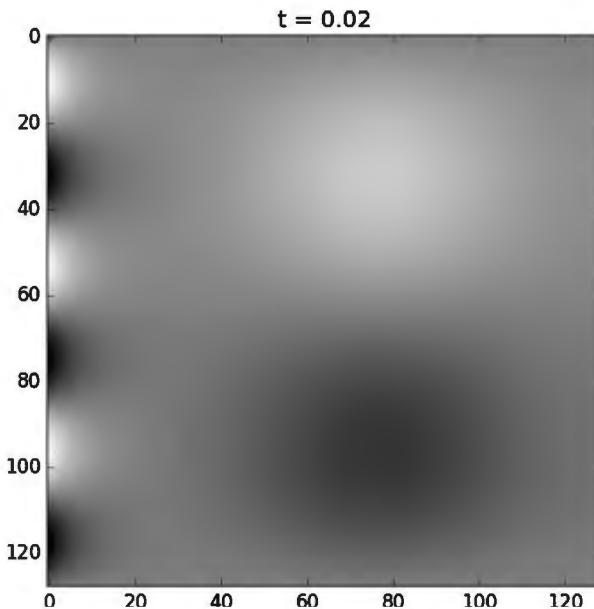
$ ./heat -ts_monitor -ht_monitor -snes_converged_reason
solving on 5 x 4 grid for t0=0. to tf=0.1 ...
  energy =  0.00e+00    nu =   0.0160
0 TS dt 0.001 time 0.
  Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
  Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
  energy = -2.51e-12    nu =   0.0196
1 TS dt 0.00122484 time 0.001
  Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
  energy = -3.60e-12    nu =   0.0303
...
17 TS dt 0.00877976 time 0.0912202
  Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
  energy = -5.03e-12    nu =   0.2589
18 TS dt 0.0161833 time 0.1

```

Note the default initial time step is  $dt = 0.001$  and the default final time is  $tf = 0.1$ . (Change these with `-ts_dt` and `-ts_max_time`, respectively.) In the above run adaptive time-stepping lengthens the step so that the final time is reached in only 18 steps. Because our heat equation problem has no time-varying inputs, the solution is converging to steady state.

Solving the implicit time-step equations apparently requires two Newton iterations, but, because these equations are actually linear for the heat equation, one can reduce this to one Newton iteration by using `-ksp_rtol 1.0e-10` or `-snes_type ksponly`. However, the first time step is exceptional. Because BDF2 is a multistep method there must be a separate strategy to get started. For that purpose it solves two systems of equations, the first of which is a preliminary solution with BDF1, that is, backward Euler.

Because the initial thermal energy is zero we would want the monitor to show that it is zero at every time step, but of course there is rounding error. Noting the coarseness of the grid, so that



**Figure 5.7.** Temperature as grayscale, the final frame from a `heat.c` run.

truncation errors are many orders of magnitude larger than the energy errors seen here, we see that energy is conserved.

Refining the grid and extending the run duration shows that the adaptive BDF2 method continues to lengthen the step, resulting in large  $\text{nu} = D_0\Delta t/(h_x h_y)$  values. For example, at the end of the following run `nu` exceeds 24,000:

```
| $ ./heat -da_refine 4 -ts_monitor -ht_monitor -ts_max_time 10
```

On the other hand, with spatial refinement the initial time step should also be shortened:

```
| $ ./heat -da_refine 4 -ts_monitor -ts_dt 1.0e-4
```

The problem can, of course, also be solved in parallel:

```
| $ mpiexec -n 4 ./heat -da_refine 5 -ts_dt 1.0e-5 -ts_monitor
```

A graphical (X windows) view of the evolving solution, i.e., a movie, comes from

```
| $ ./heat -da_refine 5 -ts_max_time 0.02 -ts_dt 1.0e-5 -ts_monitor \
    -ts_monitor_solution draw
```

Figure 5.7 shows the last frame. The spatial variation in temperature at the left side comes from the nonhomogeneous boundary condition  $\gamma(y)$ , while in the interior emerging variation is caused by  $f(x, y)$ .

A Python script `plotTS.py` in the same directory (`c/ch5/`) can be used to put either a trajectory like Figure 5.1 or a sequence of frames like Figure 5.7 into image files. One generates PETSC binary files from a run with a given resolution  $m_x \times m_y$  as follows; in this context we also ask for fixed time steps:

```
| $ ./heat -da_refine 5 -ts_adapt_type none -ts_monitor binary:t.dat \
    -ts_monitor_solution binary:u.dat
| $ ./plotTS.py -mx 129 -my 128 t.dat u.dat -oroot bar
```

Files `bar000.png` ... `bar100.png` are generated. `MOVIES.md` describes how these frames can be used to generate a movie in `.m4v` format.

To summarize our heat equation problem so far, the method of lines converts the heat equation (5.27) into a large, stiff, and linear system of ODEs in form (5.1). An implicit time-stepping method using SNES solves equations (5.13) at each time step.

Now we may compare the performance of schemes. The following base run takes  $M = 29$  adaptive steps to reach the final time  $t_f = 0.1$  using the default  $O(\Delta t^2)$  BDF2 method on a modestly refined grid of  $N = m_x \times m_y \approx 10^4$  points:

```
| $ ./heat -da_refine 5 -ts_monitor -ht_monitor
```

Note that the initial time step (`-ts_dt 0.001`) grows to much larger values as the solution approaches steady state. The wall clock time is a couple of seconds using a `-with-debugging=0` PETSC configuration.

In the list below we add options to the base run. The results are compared by relative wall clock time  $R$ , number of steps  $M$ , and final  $\nu = \nu_f$  value. Thus  $R < 1$  for a method faster than BDF2 and  $R > 1$  for a slower method. Note that  $\nu_f \gg 1$  indicates time steps which are long relative to the limitations imposed by stiffness.

**(a)** (base run)

$R = 1, M = 28, \nu_f = 265$

This implicit, A-stable, stiff-decay, and  $O(\Delta t^2)$  local truncation error method, namely `-ts_type bdf -ts_bdf_order 2`, is the default for `heat.c`.

**(b)** `-ts_type rk`

$R = 7.7, M = 5200, \nu_f = 0.31$

The explicit, adaptive, and  $O(\Delta t^3)$  RK3bs method (`-ts_rk_type 3bs`) detects the stiffness as a difference between the embedded schemes and thus takes many short time steps and too much run time. It gets worse on further-refined grids, with  $R = 18$  on the next (`-da_refine 6`) grid.

**(c)** `-ts_type rk -ts_adapt_type none`

$R = ?, M = 100, \nu_f = 16.4$

The same method as **(b)**, but without adaptivity. This unstable numerical solution overflows at around 50 steps.

**(d)** `-ts_type beuler`

$R = 2.5, M = 100, \nu_f = 16.4$

This implicit, A-stable, stiff-decay, and  $O(\Delta t^1)$  method generates a qualitatively good solution (`-ts_monitor_solution draw`) without adaptivity, but (presumably) with larger numerical error than BDF2. Slower run time comes from not lengthening the step as the solution approaches equilibrium.

**(e)** `-ts_type beuler -ts_adapt_type basic`

$R = 1.2, M = 32, \nu_f = 232$

Adding adaptivity speeds things up, but the numerical error is still larger than for BDF2.

**(f)** `-ts_type cn`

$R = 2.2, M = 100, \nu_f = 16.4$

Crank-Nicolson ( $\theta = 1/2$ ) is an implicit, A-stable, and  $O(\Delta t^2)$  method, but without stiff decay. Here it is nonadaptive. Visualization (`-ts_monitor_solution draw`) shows a subtle problem: solution features oscillate unphysically near the left boundary.

(g) `-ts_type cn -ts_dt 0.01`  $R = 0.49, M = 10, \nu_f = 164$

With longer fixed time steps, visualizations shows large, though bounded, oscillations everywhere. This implicit method is unsafe for the heat equation without adaptivity.

(h) `-ts_type cn -ts_adapt_type basic`  $R = 1.1, M = 36, \nu_f = 230$

With adaptivity, a perfectly reasonable method.

(i) `-ts_type theta -ts_theta_theta 0.7 -ts_theta_endpoint`

$R = 2.3, M = 100, \nu_f = 16.4$

By moving the  $\theta$  value toward the more stable  $\theta = 1$  (`beuler`) end, we tame the oscillation of nonadaptive Crank-Nicolson. But now the order is  $O(\Delta t^1)$  like `beuler`.

(j) `-ts_bdf_order 3`  $R = 0.90, M = 28, \nu_f = 324$

BDF3: This implicit, stiff decay, and  $O(\Delta t^3)$  method is excellent, even though it slightly fails A-stability [7].

(k) `-ts_bdf_order 4`  $R = 0.99, M = 32, \nu_f = 302$

BDF4: This  $O(\Delta t^4)$  method has the same properties as BDF3; it is also excellent though, again, it fails A-stability [7].

For a heat or diffusion equation problem like this we conclude that BDF2–4 and adaptive Crank-Nicolson are all good choices. For these methods the trade-off between speed and accuracy can be controlled by options `-ts_rtol` and `-ts_atol` (not shown). We also conclude that only A-stable and stiff decay methods (backward Euler and BDF2), or methods whose violation of absolute stability in the left half-plane is small (BDF3–4), are safe to use without adaptivity. Adaptive explicit methods are sometimes usable for diffusion problems but rarely a good choice on fine grids.

## Coupled reaction-diffusion equations

The next example also uses a method of lines discretization, a 2D DMDA, and centered finite differences to produce a stiff ODE system. This example is more interesting than the heat equation in three ways:

- It is a system of two coupled, scalar PDEs.
- They are nonlinear.
- The stiff part is separated to allow implicit/explicit (IMEX) methods.

We solve the following reaction-diffusion problem [123]:

$$\begin{aligned} \frac{\partial u}{\partial t} &= D_u \nabla^2 u - uv^2 + \phi(1-u), \\ \frac{\partial v}{\partial t} &= D_v \nabla^2 v + uv^2 - (\phi + \kappa)v. \end{aligned} \tag{5.32}$$

Here  $u(t, x, y)$  and  $v(t, x, y)$  model chemical concentrations, both of which diffuse in space, with constant coefficients  $D_u > 0$  and  $D_v > 0$ . A reaction between the two chemicals converts  $u$  into  $v$  at a rate  $uv^2$ . Chemical  $u$  is supplied externally at rate  $\phi > 0$ , which also determines a decay rate for both species, and an additional chemical reaction eliminates  $v$  by converting it into an inert product at rate  $\kappa > 0$ . We consider an initial/boundary value problem for (5.32) in a square domain  $\mathcal{S} = [0, L) \times [0, L)$  with periodic boundary conditions, i.e., on a torus with no boundary.

One needs a numerical approximation to reveal the “surprising variety of irregular spatiotemporal patterns” [123] produced by this model. Turing himself suggested that it “might be possible to treat a few particular cases [of such a reaction-diffusion model] in detail with the aid of a digital computer” [145], and we are doing exactly that.

While the 1993 simulations by Pearson [123] used the explicit Euler scheme on an early parallel supercomputer, we may take implicit time steps which are potentially thousands of times longer than such stability restricted forward Euler steps. On the other hand, though this problem illustrates the value of implicit time-stepping in the presence of stiff diffusive terms, long time steps will lose accuracy even if done stably. Thus this example illustrates the typical trade-offs in a (nonlinear and autonomous) problem where the evolving solution remains far from steady state; compare this with the previous heat equation model which approaches steady state.

As with the heat equation, the Laplacian  $\nabla^2$  in (5.32) is stiff. For such situations we may write the (spatially discretized) system in the form

$$\mathbf{F}(t, \mathbf{y}, \mathbf{y}') = \mathbf{G}(t, \mathbf{y}). \quad (5.33)$$

We will put the stiff terms in  $\mathbf{F}$  and the nonstiff in  $\mathbf{G}$  so that a TS can apply different schemes to each side. Actually, considering a method of lines discretization of (5.32), either form (5.1) or (5.33) can be made to work (Exercise 5.10), but form (5.33) increases our solver options. The latter form is also suitable for *differential-algebraic equations* (DAEs) [7], wherein  $\partial\mathbf{F}/\partial\mathbf{y}'$  is singular, but such problems are outside of our scope.

Extracts from code `pattern.c` appear below. We first define two structs named `Field` and `PatternCtx`. The first is simply the pointwise value of the solution, a pair  $(u, v)$ :

```
typedef struct {
    double u, v;
} Field;
```

The second contains parameter values:

```
typedef struct {
    double L,           // domain side length
           Du,          // diffusion coefficient: u equation
           Dv,          //                      v equation
           phi,          // "dimensionless feed rate" (F in Pearson 1993)
           kappa;        // "dimensionless rate constant" (k in Pearson 1993)
} PatternCtx;
```

Each parameter can be adjusted using option `-ptn_parameter`.

Next we show how the major types are created in `main()`, starting with a periodic, 2D DMMDA with two degrees of freedom at each point, i.e.  $u, v$ , and a “box” stencil for the 9-point FD formula<sup>22</sup> applied to the Laplacian terms:

---

<sup>22</sup>The default grid is  $3 \times 3$ , in part so that `-snes_fd_color` can be effective. On a periodic grid with a stencil width of one, the grid dimensions must be divisible by 3 for coloring to work.

```
DMDACreate2d(PETSC_COMM_WORLD,
              DM_BOUNDARY_PERIODIC, DM_BOUNDARY_PERIODIC,
              DMDA_STENCIL_BOX, // for 9-point stencil
              3,3,PETSC_DECIDE,PETSC_DECIDE,
              2, 1,           // degrees of freedom, stencil width
              NULL,NULL,&da);
```

Code 5.10 shows the setup of the TS object and the time axis, which differs from `heat.c` because we set *four* call-backs for parts of form (5.33):

- `DMDATSSetIFunctionLocal()` for  $\mathbf{F}(t, \mathbf{y}, \mathbf{y}')$ ,
- `DMDATSSetIJacobianLocal()` for derivatives of  $\mathbf{F}$ ,
- `DMDATSSetRHSFunctionLocal()` for  $\mathbf{G}(t, \mathbf{y})$ , and
- `DMDATSSetRHSJacobianLocal()` for derivatives of  $\mathbf{G}$ .

Note there are options to not set the Jacobians. (Compare Exercise 5.10.)

```
TSCreate(PETSC_COMM_WORLD,&ts) ;
TSSetProblemType(ts ,TS_NONLINEAR) ;
TSSetDM(ts ,da) ;
TSSetApplicationContext(ts ,&user) ;
DMDATSSetRHSFunctionLocal(da,INSERT_VALUES,
                           (DMDATSRHSFunctionLocal)FormRHSFunctionLocal,&user) ;
if (!no_rhsjacobian) {
    DMDATSSetRHSJacobianLocal(da,
                               (DMDATSRHSJacobianLocal)FormRHSJacobianLocal,&user) ;
}
DMDATSSetIFunctionLocal(da,INSERT_VALUES,
                        (DMDATSIFunctionLocal)FormIFunctionLocal,&user) ;
if (!no_ijacobian) {
    DMDATSSetIJacobianLocal(da,
                            (DMDATSIJacobianLocal)FormIJacobianLocal,&user) ;
}
TSSetType(ts ,TSARKIMEX) ;
TSSetTime(ts ,0.0) ;
TSSetMaxTime(ts ,200.0) ;
TSSetTimeStep(ts ,5.0) ;
TSSetExactFinalTime(ts ,TS_EXACTFINALTIME_MATCHSTEP) ;
TSSetFromOptions(ts ) ;
```

**Code 5.10.** *c/ch5/pattern.c, part I. Set up the TS and its callbacks.*

We have chosen ARKIMEX (*adaptive Runge-Kutta implicit/explicit*) [8], which treats  $\mathbf{F}$  in (5.33) implicitly and  $\mathbf{G}$  explicitly, as the default TS type. However, nonsplit, fully implicit methods also work well on this example without any modifications to user code; see Exercise 5.9 which compares CN and BDF methods.

Regarding the spatial discretization of (5.32), we assume a vector  $\mathbf{u}$  of discrete values  $u_{i,j}(t) \approx u(t, x_i, y_j)$ , and a similar vector  $\mathbf{v}$ . Together these form the state vector of ODE system (5.33), namely  $\mathbf{y} = [\begin{smallmatrix} \mathbf{u} \\ \mathbf{v} \end{smallmatrix}]$ , but the actual order of the values in memory is

$$\mathbf{y} = \begin{bmatrix} u_{0,0} \\ v_{0,0} \\ u_{1,0} \\ v_{1,0} \\ \vdots \end{bmatrix}. \quad (5.34)$$

That is, the components  $u, v$  are *interleaved*, with the particular  $i, j$  grid order coming from the DMDA (Chapter 3). On the other hand, awareness of the storage order is not needed when writing code because `DMDAVecGetArray()` and `MatSetValuesStencil()` let us to refer to values of  $u$  and  $v$  by component name (for `Vecs`) or component index ( $c=0, 1$  for `Mat` entries), respectively. For example, in Codes 5.11–5.13 below, `Field **aY` is a C array holding the gridded approximation to  $\mathbf{y}$ , with components referenced by names `aY[j][i].u` and `aY[j][i].v`.

```
PetscErrorCode FormIFunctionLocal(DMDALocalInfo *info, PetscReal t,
                                    Field **aY, Field **aYdot, Field **aF,
                                    PatternCtx *user) {
    PetscInt i, j;
    const PetscReal h = user->L / (PetscReal)(info->mx),
                    Cu = user->Du / (6.0 * h * h),
                    Cv = user->Dv / (6.0 * h * h);
    PetscReal u, v, lapu, lapv;

    for (j = info->ys; j < info->ys + info->ym; j++) {
        for (i = info->xs; i < info->xs + info->xm; i++) {
            u = aY[j][i].u;
            v = aY[j][i].v;
            lapu = aY[j+1][i-1].u + 4.0*aY[j+1][i].u + aY[j+1][i+1].u
                  + 4.0*aY[j][i-1].u - 20.0*u + 4.0*aY[j][i+1].u
                  + aY[j-1][i-1].u + 4.0*aY[j-1][i].u + aY[j-1][i+1].u;
            lapv = aY[j+1][i-1].v + 4.0*aY[j+1][i].v + aY[j+1][i+1].v
                  + 4.0*aY[j][i-1].v - 20.0*v + 4.0*aY[j][i+1].v
                  + aY[j-1][i-1].v + 4.0*aY[j-1][i].v + aY[j-1][i+1].v;
            aF[j][i].u = aYdot[j][i].u - Cu * lapu;
            aF[j][i].v = aYdot[j][i].v - Cv * lapv;
        }
    }
    return 0;
}
```

**Code 5.11.** *c/ch5/pattern.c, part II. Evaluate  $\mathbf{F}$  in (5.33).*

Code 5.11 shows `FormIFunctionLocal()` which computes

$$\mathbf{F}(t, \mathbf{y}, \mathbf{y}') = \mathbf{y}' - \begin{bmatrix} D_u A \mathbf{u} \\ D_v A \mathbf{v} \end{bmatrix}. \quad (5.35)$$

Here  $A$  is a centered, 9-point finite difference approximation of the scalar Laplacian  $\nabla^2$  using square-cell spacing  $h = h_x = h_y$ . Note that the 5-point, star-stencil scheme used in `heat.c` corresponds to this submatrix at each grid point:

$$A^{[5]} = \frac{1}{h^2} \begin{bmatrix} 1 & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix}, \quad (5.36)$$

By contrast, the 9-point, box-stencil scheme used in `pattern.c` has submatrix

$$A^{[9]} = \frac{1}{6h^2} \begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix}. \quad (5.37)$$

This scheme is recommended for reaction-diffusion models because it is more isotropic [84, section III.6]. Both (5.36) and (5.37) are  $O(h^2)$  approximations of the Laplacian, but, as demonstrated in Exercise 5.11, the former leads to much larger grid artifacts.

```

PetscErrorCode FormRHSFunctionLocal(DMDALocalInfo *info,
                                    PetscReal t, Field **aY, Field **aG, PatternCtx *user) {
  PetscInt i, j;
  PetscReal uv2;

  for (j = info->ys; j < info->ys + info->ym; j++) {
    for (i = info->xs; i < info->xs + info->xm; i++) {
      uv2 = aY[j][i].u * aY[j][i].v * aY[j][i].v;
      aG[j][i].u = -uv2 + user->phi * (1.0 - aY[j][i].u);
      aG[j][i].v = +uv2 - (user->phi + user->kappa) * aY[j][i].v;
    }
  }
  return 0;
}

```

**Code 5.12.** *c/ch5/pattern.c, part III. Evaluate  $\mathbf{G}$  in (5.33).*

`FormRHSFunctionLocal()` in Code 5.12 computes the nonlinear terms, but it includes no differential operators and thus is nonstiff:

$$\mathbf{G}(t, \mathbf{y}) = \begin{bmatrix} -\mathbf{u}\mathbf{v}^2 + \phi(1 - \mathbf{u}) \\ +\mathbf{u}\mathbf{v}^2 - (\phi + \kappa)\mathbf{v} \end{bmatrix}. \quad (5.38)$$

We want the stiff part  $\mathbf{F}$  of form (5.33) to be treated implicitly for stability. The corresponding Jacobian is implemented in `FormIJacobianLocal()` (Code 5.13). This call-back function computes a shifted combination of derivatives,

$$J = (\text{shift}) \frac{\partial \mathbf{F}}{\partial \mathbf{y}'} + \frac{\partial \mathbf{F}}{\partial \mathbf{y}}, \quad (5.39)$$

because all TS schemes use only such linear combinations, and not the derivatives separately. In our case  $\partial \mathbf{F} / \partial \mathbf{y}'$  is the identity, and thus  $J$  equals  $\partial \mathbf{F} / \partial \mathbf{y}$  plus a constant along the diagonal. The other part of the Jacobian,  $\partial \mathbf{G} / \partial \mathbf{y}$  implemented in `FormRHSJacobianLocal()`, is less interesting and its evaluation is not shown.

```

PetscErrorCode FormIJacobianLocal(DMDALocalInfo *info,
                                    PetscReal t, Field **aY, Field **aYdot,
                                    PetscReal shift, Mat J, Mat P,
                                    PatternCtx *user) {
  PetscInt i, j, s, c;
  const PetscReal h = user->L / (PetscReal)(info->mx),
                 Cu = user->Du / (6.0 * h * h),
                 Cv = user->Dv / (6.0 * h * h);
  PetscReal val[9], CC;
  MatStencil col[9], row;

  for (j = info->ys; j < info->ys + info->ym; j++) {
    row.j = j;
    for (i = info->xs; i < info->xs + info->xm; i++) {
      row.i = i;
      for (c = 0; c < 2; c++) { // u,v equations are c=0,1
        row.c = c;
        CC = (c == 0) ? Cu : Cv;
        for (s = 0; s < 9; s++)
          col[s].c = c;
        col[0].i = i; col[0].j = j;
        val[0] = shift + 20.0 * CC;
        col[1].i = i-1; col[1].j = j; val[1] = -4.0 * CC;
        col[2].i = i+1; col[2].j = j; val[2] = -4.0 * CC;
      }
    }
  }
}

```

```

        col[3].i = i;    col[3].j = j-1;  val[3] = - 4.0 * CC;
        col[4].i = i;    col[4].j = j+1;  val[4] = - 4.0 * CC;
        col[5].i = i-1; col[5].j = j-1;  val[5] = - CC;
        col[6].i = i-1; col[6].j = j+1;  val[6] = - CC;
        col[7].i = i+1; col[7].j = j-1;  val[7] = - CC;
        col[8].i = i+1; col[8].j = j+1;  val[8] = - CC;
        MatSetValuesStencil(P,1,&row,9,col,val,INSERT_VALUES);
    }
}
}

MatAssemblyBegin(P,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(P,MAT_FINAL_ASSEMBLY);
if (J != P) {
    MatAssemblyBegin(J,MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(J,MAT_FINAL_ASSEMBLY);
}
return 0;
}

```

**Code 5.13.** *c/ch5/pattern.c*, part IV. Derivatives (5.39) of  $\mathbf{F}$  in (5.33).

## Generating patterns

It is time to try it out. The following run includes visualization of solution components  $(u, v)$  at each time step:

```
$ make pattern
$ ./pattern -da_refine 5 -ts_monitor_solution draw
```

We can also monitor the TS and SNES behavior of the default method:

```
$ ./pattern -da_refine 4 -ts_monitor -snes_converged_reason
```

The output (not shown) reveals the adaptive time steps, and three SNES solves per time step, of the default ARKIMEX type, namely `-ts_arkimex_type 3`, a third-order method with one explicit stage and three implicit stages [8]. The SNES solves each show two Newton iterations, which is surprising because we have separated the (stiff) *linear* diffusion term into  $\mathbf{F}$ . However, tightening the KSP tolerance to `-ksp_rtol 1e-10` yields one Newton step per implicit stage as expected. (One may instead avoid the issue by using `-snes_type ksponly`.)

A fully implicit method like CN or BDF can also use the separated form (5.33); see Exercise 5.9.

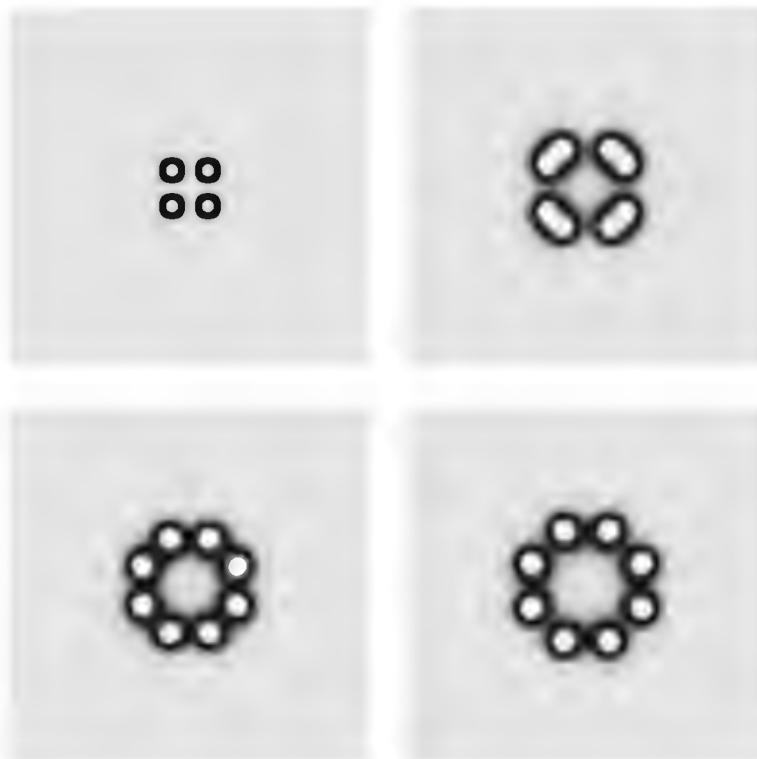
Next we reproduce Figure 4 of [123] which used a  $256 \times 256$  grid and the default parameter values  $D_u = 8 \times 10^{-5}$ ,  $D_v = 4 \times 10^{-5}$ ,  $\phi = 0.024$ ,  $\kappa = 0.06$ . The following run uses 65 total time steps of fixed length  $\Delta t = 10.0$ , with the result shown in Figure 5.8:

```
$ mpiexec -n 4 ./pattern -da_grid_x 256 -da_grid_y 256 -snes_type ksponly \
    -ts_dt 10 -ts_max_time 650 -ts_adapt_type none -ts_monitor
```

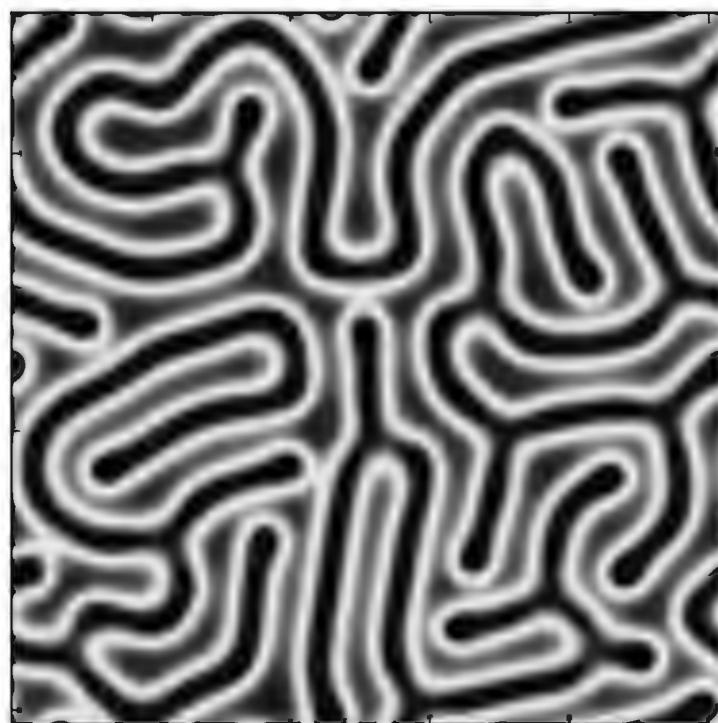
Pearson [123] used 65,000 forward-Euler steps of  $\Delta t = 0.01$ . This run would complete slightly faster using adaptivity, but here we are duplicating the frames shown in the figure in [123]. Exercise 5.9 compares the performance of a number of TS types on this problem.

The reader might want to explore the  $\phi, \kappa$  parameter space of pattern-formation equations (5.32). By adding noise to the default four-spot initial condition one sees a variety of patterns after sufficient duration. For example, Figure 5.9 is the last frame from the following run:

```
$ mpiexec -n 4 ./pattern -da_refine 6 -ptn_phi 0.05 -ptn_kappa 0.063 \
    -snes_type ksponly -ts_max_time 15000 -ptn_noisy_init 0.15 \
    -ts_monitor -ts_monitor_solution draw
```



**Figure 5.8.** Component  $u$  as grayscale at  $t = 0$  and  $t = 350$  (top row) and  $t = 510$  and  $t = 650$  (bottom row).



**Figure 5.9.** Pattern generated from parameter values  $\phi = 0.05$ ,  $\kappa = 0.063$ .

---

## Exercises

- 5.1. Show that  $y(t) = \tan(t)$  is the (unique) solution to the scalar ODE initial value problem  $y' = 1 + y^2$ ,  $y(0) = 0$ . Modify `ode.c` to solve this problem, and run from  $t_0 = 0$  to  $t_f = 2$ . What numerical evidence shows that your approximation of  $y(2)$  is totally meaningless?
- 5.2. Which TS types work with `ode.c`, that is, do not give run-time errors? Do `-help | grep ts_type` to find possibilities, and try them. Which ones work using the additional option `-snes_fd`? Which ones work with `odejac.c`? Explain as much as you can.
- 5.3. Code `ode.c` is serial only. With `mpiexec -n 2`, what happens and why?
- 5.4. Show by hand that the eigenvalues of matrix  $B$  in (5.16) are  $\lambda = i, -i, -101$ . Use a full eigen-decomposition  $B = X\Lambda X^{-1}$  to confirm (5.17), noting that

$$e^{Bt} = X e^{\Lambda t} X^{-1} = X \begin{bmatrix} e^{\lambda_0 t} & & \\ & \ddots & \\ & & e^{\lambda_{N-1} t} \end{bmatrix} X^{-1}.$$

- 5.5. Modify `odejac.c` to a similar code `stiff.c` which solves (5.15)–(5.16). Use `VecView()` to print the computed solution at  $t_f = 10$ , and use `TSGetStepNumber()` to print the number of steps. Confirm the results shown in the text for `-ts_type rk -ts_rk_type 2a` and for `-ts_type cn`. Now use the adaptive, default RK scheme RK3bs with an accuracy goal set by `-ts_rtol`, `-ts_atol` as in the following Bash loop:

```
for POW in 2 3 4 5 6 7 8 9 10; do
    ./stiff -ts_type rk -ts_rtol 1.0e-$POW -ts_atol 1.0e-$POW
done
```

One may conclude that for this example about 400 steps are necessary to get any accuracy at all (e.g., a couple of digits) using method RK3bs.

- 5.6. (a) One can compute the stability function  $f(z)$  for a one-step scheme by applying it to the scalar test equation (5.18), using  $z = h\lambda$ , and simplifying. Confirm all formulas (5.20).
- (b) Finding the stability region  $\{|f(z)| \leq 1\}$  is easy with a contour plotter tool. For the RK2a scheme, for example, confirm that  $|f(z)|^2 = 1$  is as shown in Figure 5.3.
- (c) Find the stability region of the BDF2 scheme (5.12) by first applying the scheme to the test equation and deriving a second-order recurrence for  $Y_\ell$ ; the coefficients will be  $z$  dependent. Solutions of this recurrence are linear combinations of solutions of the form  $Y_\ell = r^\ell$ . Require these solutions to be bounded, i.e.,  $|r| \leq 1$  for both roots, and thereby generate the region shown in Figure 5.3. (Generally one needs  $|r| \leq 1$  for all roots, and also that all roots with  $|r| = 1$  are simple [7].)

- 5.7. Show that the stiff-decay test equation (5.23) has solution

$$y(t) = e^{\lambda t} - \lambda \int_0^t e^{\lambda(t-s)} \gamma(s) ds.$$

Then show that the family of functions  $D_\lambda(x) = -\lambda e^{\lambda x}$  form a Dirac delta function [51] in the sense that, for any continuous function  $\varphi(x)$ ,  $\int_0^\infty D_\lambda(x) \varphi(x) dx = \varphi(0)$  as  $\text{Re } \lambda \rightarrow -\infty$ . Conclude that the solution to (5.23) is asymptotic to  $\gamma(t)$  as  $\text{Re } \lambda \rightarrow -\infty$ .

- 5.8. Show (5.29). Then fill in the details to show (5.30). Note that the periodic boundary condition amounts to treating the top/bottom locations as interior points of a cylindrical surface.

- 5.9. PETSC TS provides an impressive variety of methods which are implicit (or semi-implicit), adaptive, at least second order, and quadratically convergent at each step. Consider adaptive versions of the run which produced Figure 5.8. Which of the following is fastest among the following IMEX, BDF, and Crank-Nicolson types?

```
-ts_type arkimex -ts_arkimex_type a2|12|ars122|2c|2d|2e|3|4|5
-ts_type bdf -ts_bdf_order 2|3|4|5|6
-ts_type cn -ts_adapt_type basic
```

The ARKIMEX runs can use `-snes_type ksponly` but the BDF and CN runs should not because then the SNES sees a nonlinear residual function. Jacobians can be tested by comparing runs with and without `-snes_fd_color`, and note the Jacobians of parts F and G can be unset with `-ptn_no_ijacobian` and `-ptn_no_rhsjacobian`, respectively. The BDF and CN types do nonlinear solves and may therefore benefit from adding `-ts_max_snes_failures -1` to shorten the step after a SNES failure.

- 5.10. Modify `pattern.c` by converting the problem to form (5.1). Compare performance. Demonstrate that `-ts_type arkimex` is no longer effective.
- 5.11. Consider this pair of reaction-diffusion equations from Barkley [13]:

$$\begin{aligned}\frac{\partial u}{\partial t} &= \nabla^2 u + \frac{1}{\epsilon} u(1-u) \left( u - \frac{1}{\alpha}(v + \beta) \right), \\ \frac{\partial v}{\partial t} &= \delta \nabla^2 v + u - v.\end{aligned}\tag{5.40}$$

Set the spatial domain to  $[0, 80] \times [0, 80]$  and impose homogeneous Neumann boundary conditions on the entire boundary. The suggested parameter values in [84, section III.6] are  $\alpha = 0.25$ ,  $\beta = 0.001$ ,  $\delta = 0$ ,  $\epsilon = 0.002$  and the initial conditions are

$$u(0, x, y) = \begin{cases} 0, & x < 40, \\ 1, & x \geq 40, \end{cases} \quad v(0, x, y) = \begin{cases} 0, & y < 40, \\ \frac{1}{2}\alpha, & y \geq 40. \end{cases}$$

Hundsdorfer and Verwer [84] show that there is a significant difference between using the 5-point stencil (5.36) and the 9-point stencil (5.37). The natural spiraling patterns generated by this problem become strongly aligned to the cartesian grid with the 5-point scheme. Reproduce this result by modifying `pattern.c` into `barkley.c`, which uses the new equations (5.40), and allow the user to choose between the 5-point and 9-point stencils at run time.

- 5.12. In running `pattern.c` one may split the preconditioning of the implicit time-step equations over the blocks generated by  $u$  and  $v$ :

```
-pc_type fieldsplit -pc_fieldsplit_type additive \
    -fieldsplit_u_pc_type ilu -fieldsplit_v_pc_type ilu
```

This PC type, discussed further in Chapters 7 and 14, replaces the default preconditioning `-pc_type ilu`. Is there any benefit? Why or why not?

## Chapter 6

# Preconditioners for PDEs

Our approach to solving nonlinear PDEs is to apply *preconditioned* Krylov methods to the discrete Newton step equations. In fact, from Chapter 2 onward we have emphasized the need for preconditioning to make a Krylov iteration effective as a linear solver. A good preconditioner should be a fast, though approximate, solution method on its own. Preconditioning is usually more critical to constructing a fast solver than is the Krylov iteration, which may shrink in importance to being an accelerator of the preconditioner [134].

The current chapter explores new preconditioner (PC) types which are designed for PDE problems. Specifically, when using a DMDA structured grid we will choose, at run time, *domain decomposition* (DD) [43, 134] or *geometric multigrid* (GMG) [21, 26, 144] methods. These preconditioners, originally conceived as stand-alone solution methods for linear elliptic PDEs, deserve substantial introductions.

This chapter only addresses scalar and linear elliptic PDEs on structured grids, but all of the major ideas generalize. An unstructured-mesh DD or GMG preconditioning method must replace the DMDA type with either DMPlex as its mesh-topology/geometry data structure (Chapter 13), or some other unstructured-mesh “infrastructure” (Chapter 10 constructs a naive version). In any case, for DD and GMG preconditioners there must be a connection between the topology/geometry of the mesh and the PC object.

On the other hand, even without using underlying grid or mesh information, the *algebraic multigrid* (AMG) method [22, 53, 144] can be applied as a preconditioner. An introduction to AMG is deferred to Chapter 10, but we will use multigrid preconditioners, either GMG or AMG or both, in all remaining chapters.

The previous four chapters have introduced seven major PETSc types:

Ch. 2: Vec, Mat, KSP, PC	for iterative linear algebra,
Ch. 3: DMDA	for structured grids,
Ch. 4: SNES	for Newton’s method,
Ch. 5: TS	for time-stepping.

Of course our PDE solutions always use Vec, Mat, KSP, and PC, though sometimes these components are out of sight. Our codes use DMDA when the grid is structured and TS for initial value problems. However, from now on we will solve all PDEs using SNES, including linear problems, because this makes the call-backs to user code, for residual and Jacobian evaluations, more uniform and flexible. Using SNES means that our linear systems “ $A\mathbf{u} = \mathbf{b}$ ” actually refer to the Newton step  $J(\mathbf{u}_k)\mathbf{s} = -F(\mathbf{u}_k)$ . These systems resemble the error equation (2.10), but with a residual as the right-hand side and a solution which converges to zero in the iteration. A single Newton iteration suffices for a linear PDE if the step equations are solved accurately.

For most PDE problems the reader should seek advice from the literature regarding preconditioner choices. Review articles [16, 152] and textbooks [64, 66] are recommended starting points.

This chapter is a not-very-rigorous introduction to subgrid-based preconditioning techniques. The leading goal is to understand and explore the PETSc API and run-time options relating to DD and GMG preconditioners, but we start by clarifying what *is* a “preconditioner.”

## Preconditioners in PETSc

If  $M$  is an invertible matrix then the left- and right-preconditioned forms of the linear system  $A\mathbf{u} = \mathbf{b}$  are the new systems, equations (2.19) and (2.20), respectively:

$$(M^{-1}A)\mathbf{u} = M^{-1}\mathbf{b} \quad \text{and} \quad (AM^{-1})(M\mathbf{u}) = \mathbf{b}. \quad (6.1)$$

Note  $M^{-1}$  is often dense—it is never assembled—and the action of  $M$  itself is not needed. Given a vector  $\mathbf{r}$ , preconditioning with  $M$  requires a code which solves systems  $M\mathbf{y} = \mathbf{r}$  for  $\mathbf{y}$ , and this is only effective if applying  $M^{-1}$  is much faster than applying  $A^{-1}$ .

From PETSc’s point of view, a *preconditioner* is a function that takes in various information about the problem and generates the action of  $M^{-1}$ ,

$$M^{-1} = \mathcal{P}(A_{\text{pre}}). \quad (6.2)$$

A PC object is a preconditioner in this sense. The notional input  $A_{\text{pre}}$  is the *preconditioning material*, consisting of various data including possibly  $A$  itself (as a linear operator), the matrix entries of  $A$  (if available), and/or other information. Note  $A_{\text{pre}}$  may be a matrix which only approximates  $A$ . For a PDE problem the material  $A_{\text{pre}}$  might include the topology and geometry of the underlying grid. (How is the grid partitioned into subdomains? What are interpolation operators from subgrids?) In any case, a PC object *may* use the action of  $A$  on vectors while a KSP method *only* uses the action of  $A$  (or  $A^\top$ ) on vectors.

A common case is that  $A_{\text{pre}}$  is sparse matrix `Mat` object holding the entries of  $A$ . When a preconditioner uses only this data to construct the action of  $M^{-1}$  on vectors, we call it a *black-box preconditioner*. Examples include the `jacobi` preconditioner (`-pc_type jacobi`), which uses the diagonal entries of  $A$ , and Gauss-Seidel and successive over-relaxation methods using the lower/upper triangles of  $A$  (`sor`); these are all considered momentarily. Other black-box preconditioners include direct factorizations of  $A$  (`lu`, `cholesky`, `svd`), incomplete factorizations (`ilu`, `icc`), and classical or smoothed-aggregation algebraic multigrid (`gamg`; Chapter 10).

However, a preconditioner which is specifically designed for PDE problems will often not be such a black box. This chapter focuses on DD (`bjacobi`, `asm`) and GMG (`mg`) methods which use as their material both the action of  $A$  and additional information about the topology and the geometry of the grid or mesh. In this chapter the grid is structured and this information comes from the `DMDA` object. Chapter 13 shows how a `DMPLex` object can play the same role for an unstructured mesh.

The goal of a preconditioner is that the spectrum of  $M^{-1}A$  or  $AM^{-1}$ —these matrices are similar and have the same spectrum (Exercise 2.4)—is well behaved for some large class of problems. For example, because the mainstream Krylov methods converge rapidly if polynomials exist which have small magnitude on the spectrum of  $A$  (Chapter 2 and [44, 66, 152]), the goal might be that the spectrum consists of a few small clusters in the complex plane.

We have previously identified some PC types as weak preconditioners for the 2D Poisson problem, especially Jacobi and incomplete Cholesky decomposition (ICC; see Chapter 3). “Weak” is here in the sense that the resulting solvers scale poorly as the problem size increases under grid refinement. For example, in Chapter 3 the combination `-ksp_type cg`

`-pc_type icc` seemed promising at first, because  $\kappa(M^{-1}A) \ll \kappa(A)$ , but as the problem size  $N = O(h^{-2})$  increased the number of iterations increased as  $O(h^{-1})$ . In fact the condition number  $\kappa(M^{-1}A)$ , like  $\kappa(A)$  itself, grows as  $O(h^{-2})$ . The best preconditioning for CG would instead generate  $M^{-1}$  such that  $\kappa(M^{-1}A)$  is bounded independently of  $h$  so that the number of preconditioned CG iterations is independent of  $N$ .

It is sometimes said that a good preconditioner should exploit information about a particular problem class. It is at least true that an effective preconditioner must provide *something else* beyond the ideas already used in Krylov iterations (Chapter 2). The new idea might come from a direct linear algebra algorithm, such as a matrix splitting or an incomplete factorization, or it might be a subgrid-based divide-and-conquer strategy as in this chapter.

**Fact 12.** Effective preconditioners aren't like Krylov iterations. *If you are already using a norm-minimizing Krylov method then you need to add a fundamentally different idea to build a fast solver. In PETSc such ideas are lumped into the preconditioner paradigm. LU decomposition, domain decomposition, multigrid, and fieldsplit are examples.*

## Classical iterations

The classical Jacobi and Gauss-Seidel (GS) iterations, defined below but possibly familiar to the reader, are preconditioned simple iterations based on matrix splittings. There are several reasons to consider these tools. They are inexpensive in a per-iteration sense, and they have smoothing effects when used on linear systems arising from elliptic PDEs. Chebyshev iteration (Chapter 2) also makes a good smoother, particularly suitable in parallel [3]. The Jacobi iteration can be used blockwise as the parallel part of a composed preconditioner. However, as we consider these roles for the classical iterations, i.e., as components of multigrid and DD methods, keep in mind that they have little power as stand-alone linear solvers.

Recall that simple iteration (2.23) is Richardson iteration (2.12) applied to the left-preconditioned system  $M^{-1}Au = M^{-1}\mathbf{b}$ . PETSc allows an additional scaling  $\alpha$  (default  $\alpha = 1$ ) in simple iteration:

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \alpha M^{-1}(\mathbf{b} - A\mathbf{u}_k). \quad (6.3)$$

This equation corresponds to the option combination

```
-ksp_type richardson -ksp_richardson_scale alpha -pc_type PC
```

where PC generates the action of  $M^{-1}$ . Note that right-preconditioned Richardson iteration is not defined.

Direct solution methods like LU decomposition, i.e., the “extreme preconditioners” with  $M^{-1} = A^{-1}$  in Chapter 2, can be applied as simple iteration (6.3). For example, in exact arithmetic `-ksp_type richardson -pc_type lu` would compute the exact solution in one iteration from any  $\mathbf{u}_0$  because  $\mathbf{u} = \mathbf{u}_0 + A^{-1}(\mathbf{b} - A\mathbf{u}_0) = A^{-1}\mathbf{b}$ . (The solver `-ksp_type preonly -pc_type lu` also computes  $\mathbf{u} = A^{-1}\mathbf{b}$ , but without a convergence check or an initial iterate.)

Suppose  $D$  is the matrix created from the diagonal of  $A$ , and that  $L, U$  are the strictly lower and strictly upper triangular parts of  $A$ . This defines a matrix *splitting* [66]:

$$A = D + L + U, \quad (6.4)$$

where  $d_{ij} = 0$  if  $i \neq j$ ,  $\ell_{ij} = 0$  if  $i \geq j$ , and  $u_{ij} = 0$  if  $i \leq j$ .

The *classical Jacobi iteration* is (6.3) with  $\alpha = 1$  and  $M = D$ ,

$$\mathbf{u}_{k+1} = \mathbf{u}_k + D^{-1}(\mathbf{b} - A\mathbf{u}_k). \quad (6.5)$$

Generally this requires all diagonal entries of  $A$  to be nonzero ( $a_{ii} \neq 0$ ), but PETSC will substitute  $a_{ii} = 1$  otherwise.<sup>23</sup> Iteration (6.5) converges if  $\rho(I - D^{-1}A) < 1$ , that is, if  $D^{-1}A$  is close to the identity matrix in a spectral sense (Exercises 2.3 and 6.2). The option combination is

```
-ksp_type richardson -pc_type jacobi
```

Certain references [26, 144] add a weight to the Jacobi iteration, but this is the same as using  $\alpha$  in Richardson iteration (Exercise 6.1). Regarding the entrywise form of the iteration, if  $v[i]$  denotes the  $i$ th entry of  $\mathbf{v}$  then (6.5) is

$$u_{k+1}[i] = \frac{1}{a_{ii}} \left( b[i] - \sum_{j \neq i} a_{ij} u_k[j] \right). \quad (6.6)$$

Entrywise form (6.6), which writes the updated value as an average of old values, suggests why the Jacobi iteration is smoothing when  $A$  is the discretization of an elliptic operator [144, section 4.7.1].

**Example 6.1.** The centered FD approximation of  $-u'' = f$  on a uniform grid with spacing  $h$  is

$$-u_{i-1} + 2u_i - u_{i+1} = h^2 f(x_i).$$

Note  $a_{ii} = 2$  while  $a_{i,i-1} = a_{i,i+1} = -1$ . Thus (6.6) is

$$u_{k+1}[i] = \frac{h^2}{2} f(x_i) + \frac{1}{2} (u_k[i-1] + u_k[i+1]).$$

The updated value  $u_{k+1}[i]$  adds the average of the neighbors  $u_k[i-1], u_k[i+1]$  from the previous iteration, which smooths out bumps in  $\mathbf{u}_k$ .

In the Jacobi iteration all values  $u_{k+1}[i]$  must be stored in a temporary work vector before use at the next iteration. This is inefficient; the updated entries *could* be used as soon as they are computed. Doing so by sweeping through indices in increasing order gives the classical *Gauss-Seidel (GS) iteration*,

$$u_{k+1}[i] = \frac{1}{a_{ii}} \left( b[i] - \sum_{j < i} a_{ij} u_{k+1}[j] - \sum_{j > i} a_{ij} u_k[j] \right), \quad (6.7)$$

again assuming  $a_{ii} \neq 0$ . Equivalently, GS uses  $\alpha = 1$  and  $M = D + L$  in simple iteration (6.3):

$$\mathbf{u}_{k+1} = \mathbf{u}_k + (D + L)^{-1} (\mathbf{b} - A\mathbf{u}_k). \quad (6.8)$$

The equivalence of (6.7) and (6.8) follows from inverting the lower-triangular matrix  $D + L$  by forward substitution (Exercise 6.3).

PETSC regards Gauss-Seidel as a case of *successive over-relaxation* (SOR) [157]. This iteration adds a weight (relaxation factor)  $\omega$  to the splitting (6.4), thus

$$A = \left( \frac{1}{\omega} D + L \right) + \left( (1 - \frac{1}{\omega}) D + U \right).$$

In other words, SOR is

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \left( \frac{1}{\omega} D + L \right)^{-1} (\mathbf{b} - A\mathbf{u}_k) \quad (6.9)$$

---

<sup>23</sup>This can be a cause of confusion, and it makes a potentially dangerous assumption about the scaling of your equations. It is wise, instead, either to avoid using Jacobi preconditioning on the part of a matrix with zero diagonal—see Chapter 14 for a nontrivial case—or to explicitly modify the zero diagonal entries yourself.

as a simple iteration, with an entrywise form similar to (6.7). The weight can be set with `-pc_sor_omega`  $\omega$ ; the default is  $\omega = 1$ .

The option combination for classical GS is

```
-ksp_type richardson -pc_type sor -pc_sor_forward
```

Option `-pc_sor_backward` reverses the order, i.e., it uses  $M = \frac{1}{\omega}D + U$  in simple iteration.

The default PETSc version of SOR is actually *symmetric*, i.e., SSOR [64]. The classical algorithm is simple iteration with

$$M = \frac{\omega}{2-\omega} \left( \frac{1}{\omega}D + L \right) D^{-1} \left( \frac{1}{\omega}D + U \right). \quad (6.10)$$

The action of  $M^{-1}$  is equivalent to a forward SOR pass followed by a backward pass. SSOR is appropriate if  $A$  itself is symmetric, in which case  $U = L^\top$  and  $M$  is symmetric. The options are

```
-ksp_type richardson -pc_type sor -pc_sor_omega OMEGA
```

using the default `-pc_sor_symmetric` ordering.

There is an oddity in the PETSc implementation of classical SOR, likely to afflict anyone playing with the classical iterations (Exercise 6.4). Namely, to improve its speed as a parallel smoother, the convergence test in `-ksp_type richardson -pc_type sor` is skipped because it would require a global reduction. Thus this particular solver runs to the maximum number of iterations allowed. It does so *unless* `-ksp_monitor` is used, in which case the norm is computed at each iteration anyway, and the convergence test *is* applied.

Any Krylov iteration, not just Richardson, can be used with Jacobi and SOR PC objects. The `jacobi` preconditioner merely extracts the entries of  $M = D$  from  $A$  and applies  $M^{-1} = D^{-1}$  to vectors, the GS preconditioner uses  $M = D + L$  the same way, and so on. On the other hand, using `-ksp_type cg -pc_type sor` as a linear PDE solver, for example, is generally not very fast. In fact, the Jacobi and SOR PCs are most useful as components of more-complicated (composed) PCs, regardless of KSP choice.

The Jacobi preconditioner `-pc_type jacobi` is always available in parallel and its meaning is unaltered by the number of processes. However, the combination

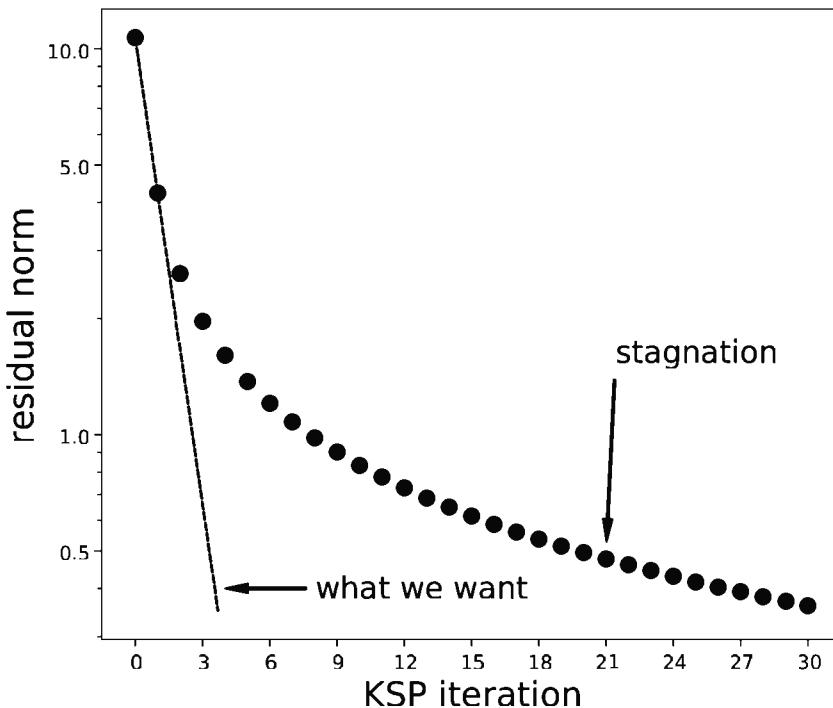
```
-pc_type bjacobi -sub_pc_type PC
```

applies PC blockwise, with no communication between the blocks during the preconditioner application. Here  $M^{-1}$  is a block diagonal matrix with the “sub” PC providing the blocks. We will generalize `bjacobi`, the default parallel PC in PETSc, to overlapping DD methods later in this chapter.

In parallel, when applying SOR or GS preconditioning, if an entry has been computed on a different processor then interprocess communication would be required. In fact, PETSc does *not* implement classical SOR, GS, or SSOR in parallel. However, `-pc_type bjacobi` can be used to provide “processor-block Gauss-Seidel” [3]. For example, processor-block SSOR is

```
-ksp_type richardson -pc_type bjacobi -sub_pc_type sor
```

(Add `-sub_pc_sor_forward` to get processor-block GS.) When run on  $P > 1$  processes this is a parallel, no-communication version of SOR, an example of a solver which depends on the number of processors. Block SOR may be a reasonable approximation of full SOR if the number of unknowns per process is large. At the other extreme, with one unknown per process, block SOR becomes the Jacobi iteration.



**Figure 6.1.** The classical iterations, such as Gauss-Seidel (GS) on the Poisson equation as shown here, tend to stagnate.

We have not yet addressed the rate of convergence of the classical iterations for a given linear system  $Ax = b$  [64, 66]. This question is somewhat tangential because we do not use these iterations directly as linear solvers, but primarily as smoothers (next). However, a well-known observation is that GS is a factor of two faster than Jacobi. For example, using a tridiagonal, diagonally dominant problem from Chapter 2, compare

```
$ ./tri -ksp_type richardson -pc_type jacobi -ksp_monitor
$ ./tri -ksp_type richardson -pc_type sor -pc_sor_forward -ksp_monitor
```

The first run gives 18 iterations while the second gives 10. Exercise 6.12 shows the phenomenon in a less trivial case.

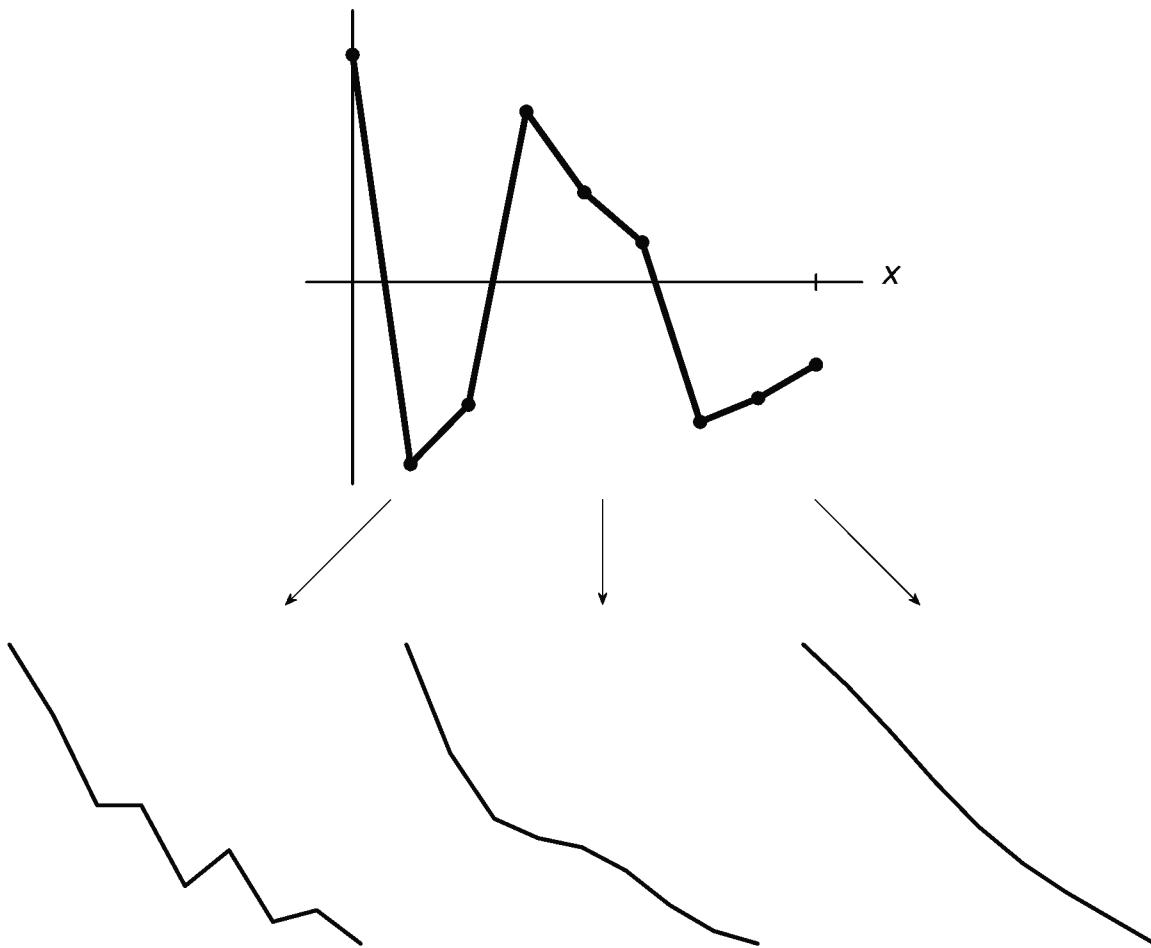
The well-known bad news about applying the classical iterations to PDE problems is that they tend to stall or stagnate after making initial progress [49, 64, 66, 152]. For example, in a 2D Poisson problem with a random initial iterate, and using the code developed later in this chapter, the residual norms from classical GS iteration from the following run are shown in Figure 6.1:

```
$ ./fish -da_refine 4 -ksp_type richardson -pc_type sor -pc_sor_forward \
-fsh_initial_type random -ksp_monitor
```

The first iteration reduces the residual norm by a factor of more than two. We would be happy if this continued, but soon the ratio of consecutive residual norms is close to one. Achi Brandt, the guru of multigrid methods, makes the following bold observation about this:

Fact 13. Stalling numerical processes must be wrong [21]. *Whenever the computer grinds very hard for small or slow effect, there must be a better way to achieve the same goal.*

Part of righting this “wrong” comes in recognizing that the classical iterations do have a desirable effect. Namely, their first few iterations smooth the error, though they are slow to eliminate it.



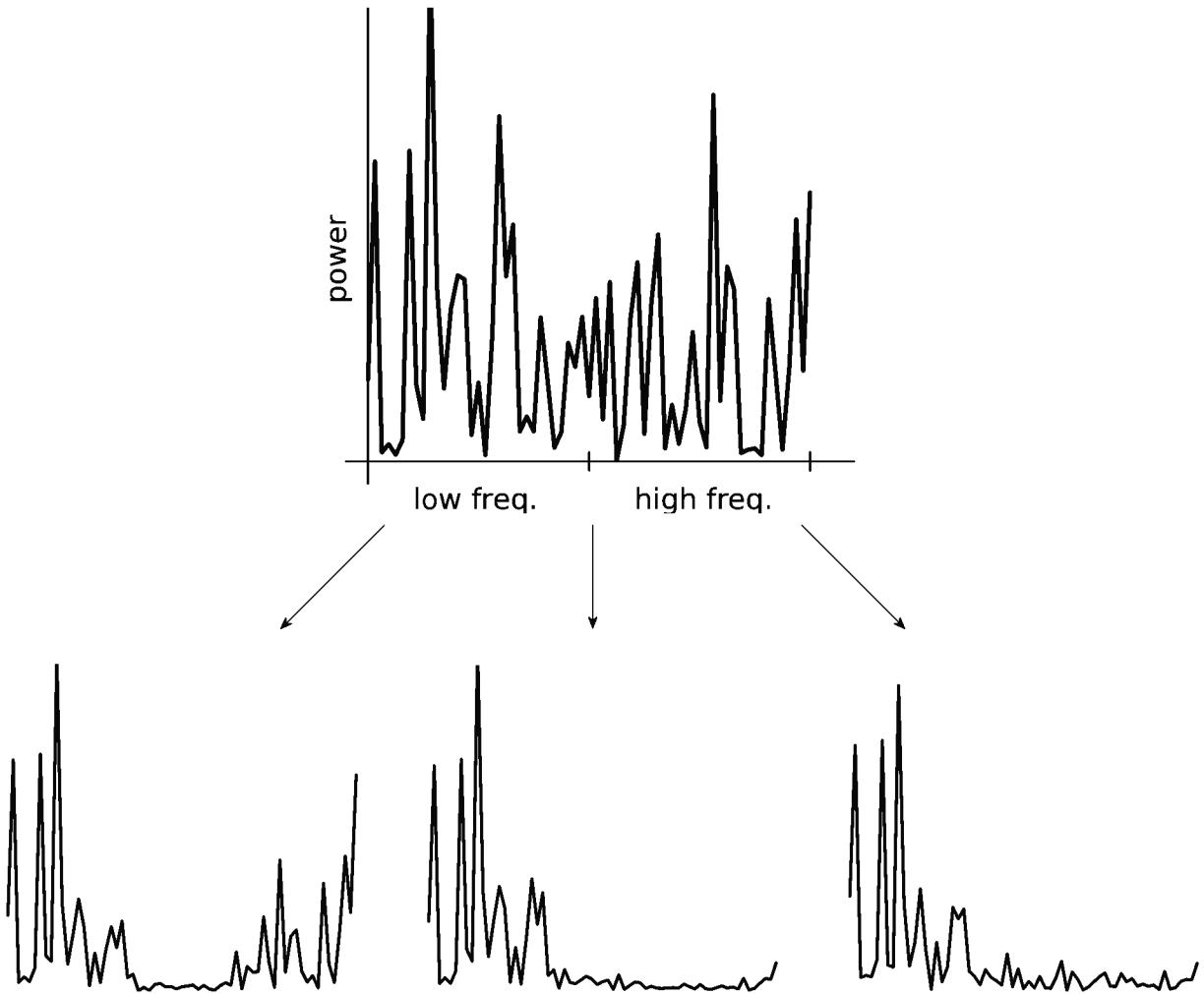
**Figure 6.2.** Smoothing of a random 1D function (top) on a 9-point grid using 4 iterations of Jacobi (left),  $\alpha = 2/3$  weighted Jacobi (middle), and GS (right) iterations, respectively, on a discrete Laplacian matrix.

## Smoothers

Suppose matrix  $A$  is the discretization of an elliptic PDE operator. As shown in Example 6.1, the classical Jacobi iteration tends to average out variation in the previous iterate. Figure 6.2 compares the effect of a few steps of the Jacobi,  $\alpha = 2/3$  weighted Jacobi, and GS iterations using a random initial iterate on a 9-point grid. We see that unweighted Jacobi is the least-effective smoother and GS the most-effective.

To compare different smoothers in a more quantitative manner we recall that the *power spectrum* of a vector is the squared-magnitude of its frequency-domain representation. That is, for a vector  $\mathbf{u}$  representing  $u(x)$ , its power spectrum is  $|\hat{\mathbf{u}}(s)|^2$ , where  $\hat{\mathbf{u}}$  is its discrete Fourier transform [25], so  $\hat{\mathbf{u}}(s)$  is the coefficient of the mode with frequency  $s$ . This definition can be generalised to any dimension.

Figure 6.3 shows the relative smoothing effect of the same classical iterations as in Figure 6.2, but by comparing the power spectra after one iteration. This figure uses a finer grid, but again the initial iterate is random. Defining a *high frequency* as greater than half of the maximum representable frequency for that grid, the better smoothers, namely  $\alpha = 2/3$  weighted Jacobi and GS, strongly damp the high frequencies. Unweighted Jacobi does not damp the highest frequencies at all, though it does in the middle of the spectrum; a little thought about sawtooth functions will suggest why.



**Figure 6.3.** The power spectrum of a random initial vector (top), and from a single iteration, of Jacobi (left),  $\alpha = 2/3$  weighted Jacobi (middle), and GS (right), on a 129-point grid.

For discrete Laplacian matrices on structured grids one may precisely quantify the smoothing effect in any number of dimensions. Define the *smoothing factor* of an iteration as the worst-case ratio of magnitudes  $|\hat{u}_1(s)|/|\hat{u}_0(s)|$  over all high frequencies  $s$ . For weighted Jacobi the  $\alpha$  value which minimizes the smoothing factor is  $\alpha = 2/3$  in 1D [26] and  $\alpha = 4/5$  in 2D [144]. Such theory can be extended to constant-coefficient elliptic operators with nice boundary conditions. For variable-coefficient and nonlinear PDE problems one can at least observe that smoothing is a local property associated to the small-wavelength modes, and discuss smoothing based on “freezing” the coefficients and using Fourier analysis on small patches [21].

However, instead of pursuing *a priori* estimation of smoothing, we will do run-time experimentation using smoothers as components of the multigrid preconditioners introduced soon (Exercises 6.26 and 6.27). Empirical comparisons of the composed multigrid methods will indirectly compare smoother quality, although other multigrid components (restriction and interpolation methods, cycle choices, and coarse grid solvers—all of these are introduced below) will interact with the smoother to determine multigrid performance.

One need not use the Richardson iteration in a smoother, but at least it avoids parallel communication. Using a KSP such as CG or MINRES, with a Jacobi or GS preconditioner gives different, and sometimes superior, smoothers.

The Chebyshev iteration (Chapter 2), which applies a polynomial in the preconditioned matrix  $M^{-1}A$ , is also suitable for smoothing:

```
-ksp_type chebyshev -pc_type x
```

Each Chebyshev iteration increases the dimension of the Krylov space and degree of the polynomial. The polynomial coefficients are chosen for effectiveness at damping error components, but can be chosen either to speed up convergence (damp all components) or to improve performance as a smoother (damp high-frequency components). In any case, the polynomial coefficients are based on computed, iterative estimates of the spectrum of  $M^{-1}A$ , a nontrivial aspect of the Chebyshev iteration which we will not pursue [64].

As a smoother, especially when combined with an SSOR PC acting only on the degrees of freedom owned by the processor, often described as “backward-forward processor-block GS,” the Chebyshev iteration is recommended. This combination, `-ksp_type chebyshev -pc_type sor`, is the PETSc default smoother inside multigrid preconditioners. Especially in parallel, a Chebyshev-iteration-based smoother for multigrid is generally superior to either classical GS—not implemented in parallel in PETSc—or classical processor-block GS [3]. (In this context “classical” means “using `-ksp_type richardson`.”) This parallel default avoids interprocessor communication as would be required in pure GS or in a Krylov iteration like CG with global reductions (Chapter 2).

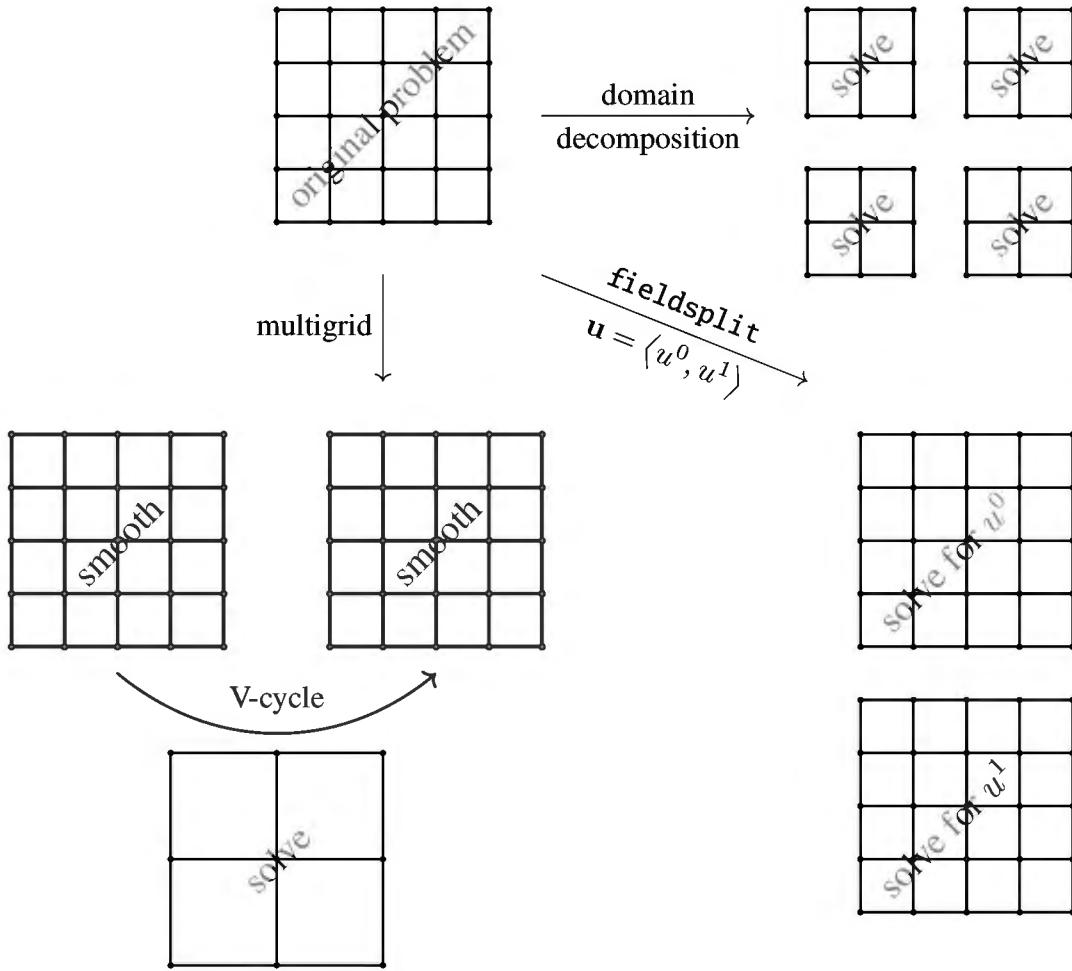
For the rest of the book, all one really needs to know about the classical iterations in their role as smoothers can be compressed into this short list:

- The Jacobi, GS, and SOR methods are implemented as PC objects.
- The classical Jacobi, GS, and SOR iterations use `-ksp_type richardson`, but one may combine these PCs with any Krylov iteration.
- Because they tend to stagnate, the classical iterations are rarely used as stand-alone linear solvers.
- The classical iterations are often used as smoothers in multigrid, but the Chebyshev iteration with backward-forward processor-block GS (i.e., local SSOR) is the default because it has good smoothing performance while avoiding interprocess communication.

## Restricting to subgrids

The most powerful preconditioners for elliptic PDE problems use subgrids to decompose the solution space into vector subspaces, and then they apply a solver on each subspace. Figure 6.4 sketches this divide-and-conquer view of three classes of preconditioners in PETSc in the structured-grid case. (A similar figure could be drawn for unstructured meshes.) In each case the subspaces are defined by restricting information about the problem to subsets of the grid indices:

- In nonoverlapping (*block Jacobi*) and overlapping (*Schwarz*) *domain decomposition* (DD) methods, the subspaces are defined by subgrids covering subdomains of the original solution domain.
- In *geometric multigrid* (GMG) methods the subspaces are defined by coarse subgrids which cover the same domain as the original grid (coextensive grids [21]). This generates a decomposition in frequency.
- In *fieldsplit* methods ([27]; Chapters 7 and 14) the subspaces correspond to choices of components of a vector-valued solution.



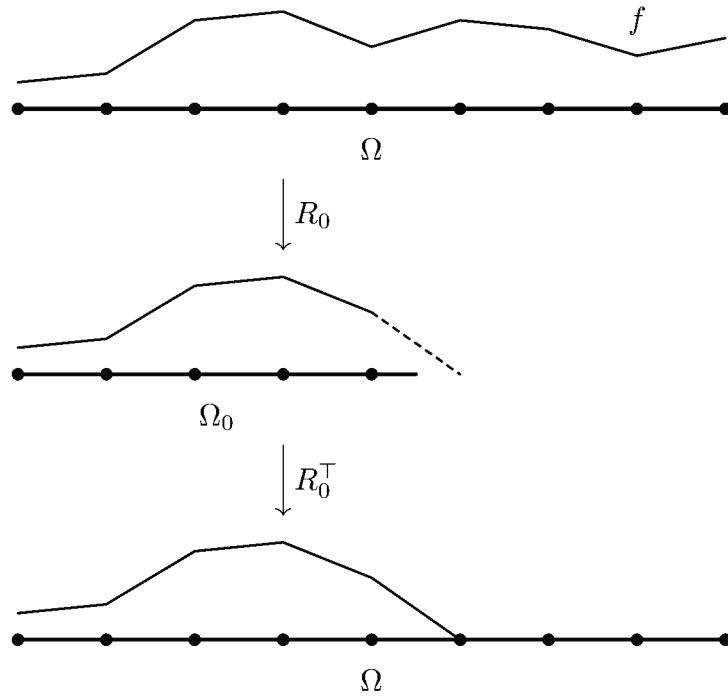
**Figure 6.4.** Powerful preconditioners for discretized PDEs use subgrid-based divide-and-conquer strategies: domain decomposition, multigrid, and componentwise decomposition of vector-valued problems (*fieldsplit*).

The original problem could be decomposed once or many times, as is done in multigrid, but the figure only shows a single-level domain-decomposition and a two-level decomposition as representing multigrid.

Suppose we have an original grid of  $N$  points. Each subgrid is both a subset of the  $N$  indices, i.e., the points of the subgrid, and a space of real-valued functions on those points. In fact, a subgrid corresponds to a rectangular matrix as follows. Suppose we have identified  $p$  subgrids and that the  $i$ th subgrid has  $n_i$  points. The  $n_i \times N$  matrix  $R_i$  has a single nonzero entry in each row, namely a 1 in the column corresponding to the global index of that point; this is the *injection matrix* for that subgrid. Applying  $R_i$  to  $\mathbf{v} \in \mathbb{R}^N$  leaves subgrid values unaltered but it eliminates all other values. Though matrix representation is helpful in presenting ideas, the matrix  $R_i$  is never formed. Instead an injection  $R_i$  is stored as an *index set*, a PETSc IS type [10, 27], an ordered list of the  $n_i$  global indices of the subgrid points.

The transpose  $R_i^\top$  is a kind of *prolongation matrix*. If  $R_i$  is an injection then  $R_i^\top$  extends a vector on the subgrid by zero to all of the original grid (Figure 6.5). Also, when  $R_i$  is an injection then the  $N \times N$  matrix  $Q_i = R_i^\top R_i$  is an orthogonal projection; it zeros the entries which are not in the subgrid.

For a given subgrid of  $n_i$  points, a *restriction matrix* generalizes an injection. We define it as any  $n_i \times N$  matrix with nonnegative entries and full row rank. Each row thus represents an average, or scaled average, from values on the original grid points to a vector supported on the



**Figure 6.5.** A subgrid injection matrix  $R_0$ , acting on a piecewise-linear function  $f : \bar{\Omega} \rightarrow \mathbb{R}$ , yields a function on a subdomain  $\Omega_0$ . Its transpose  $R_0^\top$ , a prolongation, extends functions by zero.

subgrid. Injections will suffice as the restriction matrices in DD methods, but for multigrid we will use averaging restrictions when transferring functions from fine grids to coarse subgrids.

Now suppose  $A\mathbf{u} = \mathbf{b}$  is the discretization of a linear PDE problem on the original grid, where  $A$  is the *system matrix*. For an injection or restriction  $R_i$  define

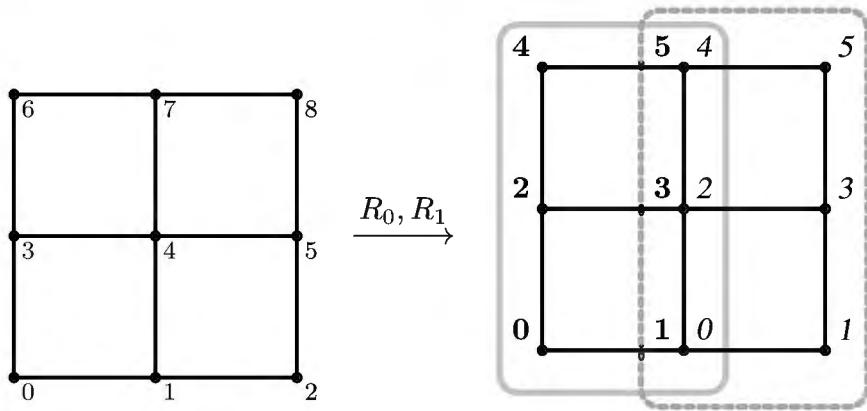
$$A_i = R_i A R_i^\top \quad (6.11)$$

as the *subgrid (system) matrix* for the  $i$ th subgrid. If  $R_i$  is an injection then the action of  $A_i$  is to zero out the columns (unknowns) other than on the  $i$ th subgrid, then apply  $A$ , and then zero out the rows (equations) other than on the  $i$ th subgrid.

**Example 6.2.** Suppose we FD discretize (Chapter 3) the Poisson equation on a square, with zero Dirichlet conditions on the sides and periodic conditions on top and bottom, using a  $3 \times 3$  FD grid. The linear system  $A\mathbf{u} = \mathbf{b}$  has a  $9 \times 9$  matrix

$$A = \begin{bmatrix} 1 & & & & & & & & \\ & 4 & & -1 & & -1 & & & \\ & & 1 & & & & & & \\ & & & 1 & & & & & \\ -1 & & & 4 & & 1 & & -1 & \\ & & & & 1 & & & & \\ -1 & & & -1 & & 4 & & & \\ & & & & & & & & \\ & & & & & & & & 1 \end{bmatrix} \quad (6.12)$$

and the unknowns have global indices  $\{0, \dots, 8\}$  (Figure 6.6, left). Suppose we decompose the grid into two overlapping subgrids as shown in Figure 6.6, right. The two  $6 \times 9$  injections  $R_i$ ,



**Figure 6.6.** A grid is decomposed into two overlapping subgrids via restriction operators  $R_0$  (solid loop; bold indices) and  $R_1$  (dashed loop; italic indices).

for  $i = 0, 1$ , are

$$R_i = \begin{bmatrix} 1 & & & & & \\ & 1 & 0 & & & \\ & & 1 & & & \\ & & & 1 & 0 & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix}, \quad \begin{bmatrix} 0 & 1 & & & & \\ & 1 & & & & \\ & & 0 & 1 & & \\ & & & 1 & 0 & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix},$$

with index set representations  $R_0 = (0, 1, 3, 4, 6, 7)$  and  $R_1 = (1, 2, 4, 5, 7, 8)$ , respectively. The subgrid problems have  $6 \times 6$  system matrices

$$A_i = R_i A R_i^\top = \begin{bmatrix} 1 & & & & & \\ & 4 & -1 & -1 & & \\ & & 1 & & & \\ & & & 4 & -1 & \\ & & & & 1 & \\ & & & & & 4 \end{bmatrix}, \quad \begin{bmatrix} 4 & -1 & -1 & & & \\ & 1 & & & & \\ & & 4 & -1 & & \\ & & & 1 & & \\ & & & & 4 & \\ & & & & & 1 \end{bmatrix}.$$

Neither the  $R_i$  nor the  $A_i$  matrices need to be formed, even as sparse matrices, as their action can be computed using  $A$  and index sets.

The above example of a subgrid decomposition goes with an underlying subdomain decomposition. The next example illustrates how domain decomposition, and the matrices  $R_i$  and  $R_i^\top$ , relate to piecewise-linear functions on intervals in  $\mathbb{R}^1$ .

**Example 6.3.** Suppose  $\Omega = (0, 8) \subset \mathbb{R}^1$  and choose overlapping subdomains (subintervals)  $\Omega_0 = (0, 4.5)$  and  $\Omega_1 = (3.5, 8)$ . Put a 9-point grid on  $\Omega$ :  $x_i = i$  for  $i = 0, 1, \dots, 8$ , so  $\{0, 1, \dots, 8\} \subset \overline{\Omega}$ . Choose subgrids  $\{0, \dots, 4\} \subset \overline{\Omega_0}$  and  $\{4, \dots, 8\} \subset \overline{\Omega_1}$ . The grid overlap (intersection) consists of a single point:  $\{4\} \subset (3, 5) = \Omega_0 \cap \Omega_1$ . Figure 6.5 shows the effect of injection  $R_0$  and prolongation  $R_0^\top$  on a piecewise-linear function  $f(x)$  defined on  $\overline{\Omega}$ . We may regard  $R_0 f$  as a function on  $\Omega_0$ , though the boundary value is indeterminant. The vector  $Q_0 f = R_0^\top R_0 f$  is a well-defined function on  $\overline{\Omega}$  which is zero at grid points not in  $\overline{\Omega_0}$ .

Note that, as long as the subgrids and their overlaps are precisely defined, some imprecision in the subgrid  $\leftrightarrow$  subdomain correspondence is accepted with respect to whether subdomains include the grid points on their topological boundaries.

In a parallel computation a grid can be decomposed into (generally) overlapping subgrids with each assigned to an MPI process. The actions of the restriction and prolongation matrices  $R_i, R_i^\top$  then involve communication between neighboring processors. Such communication updates the “ghost” grid-overlap points (Figure 3.4), and, in this context, PETSc allows two representations of a vector  $\mathbf{v}$  on the original grid. A *global* representation of  $\mathbf{v}$  assigns each value  $v[j]$  uniquely to a single process. A *local* representation is a collection of subgrid representations  $R_i \mathbf{v}$  which generally contain redundant information on the overlaps. In our examples in Chapters 3 and 4 we called `DMCreateGlobalVector()` to allocate global Vecs to hold the initial iterate, solution, or the right-hand side. When evaluating a residual we worked with the corresponding local Vec; on each process it is a C array including the ghosts. The necessary ghost communication, an update of the local Vec from the global Vec, uses a `DMDAGlobalToLocalBegin|End()` pair. Once  $\mathbf{v}$  has an up-to-date local representation, the process of solving on one subgrid, namely the action  $(A_i)^{-1} R_i \mathbf{v}$ , requires no communication. However, the result of this local solution process may need to be communicated back to other processes, and be combined into global information, as explained in the next section.

## Subgrid corrections and their compositions

From now on  $R_i$  denotes a restriction, that is, a full-rank  $n_i \times N$  matrix with nonnegative rows; injections are examples. The  $i$ th subgrid, now denoted  $\Omega_i$ , is defined as the range of  $R_i$ .

To use a solver on  $\Omega_i$  in a preconditioner, we will need the following sequence of actions:

- restrict the equation to the subgrid using  $R_i$  (*restrict the residual*),
- solve the subgrid problem using the matrix  $A_i = R_i A R_i^\top$ , and
- extend the solution back using  $R_i^\top$  (*generate a correction*).

In terms of a residual  $\mathbf{r} = \mathbf{b} - A\mathbf{v}$ , for a given global vector  $\mathbf{v}$ , this sequence is

$$\mathbf{r} \rightarrow R_i^\top (A_i)^{-1} R_i \mathbf{r}.$$

This action is a “correction” in the same sense that each step of simple iteration (6.3) should move closer to  $\mathbf{u} = A^{-1}\mathbf{b}$ , thereby correcting the error.

For the  $i$ th subgrid  $\Omega_i$  we define an  $N \times N$  *subgrid correction matrix* [134],

$$B_i = R_i^\top (A_i)^{-1} R_i, \quad (6.13)$$

about which we make several observations:

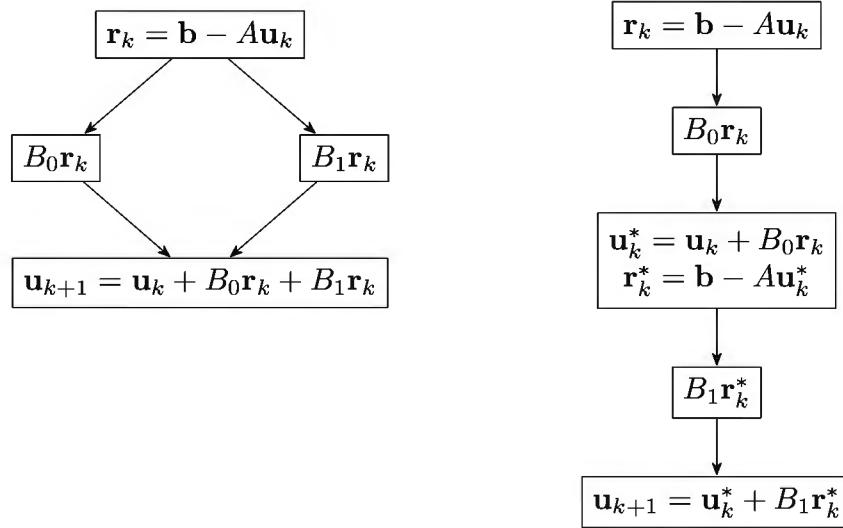
- The calculation

$$B_i = R_i^\top (R_i A R_i^\top)^{-1} R_i \stackrel{\text{wrong!}}{=} R_i^\top (R_i^\top)^{-1} A^{-1} (R_i)^{-1} R_i = A^{-1}$$

is wrong for any proper subgrid— $R_i$  is not even square—but it captures the idea that  $B_i$  wants to be  $A^{-1}$ . (Matrix  $B_i$  is in fact limited to subgrid information so it cannot actually solve the whole problem.)

- If  $R_i$  is a subgrid injection, and if the unknowns are ordered so that points in the  $i$ th subgrid  $\Omega_i$  come first, then  $B_i$  has block form

$$B_i = \begin{bmatrix} A_i^{-1} & 0 \\ 0 & 0 \end{bmatrix}. \quad (6.14)$$



**Figure 6.7.** One simple iteration using additive (left) and multiplicative (right) composition of two subgrid corrections  $B_i$ .

- If  $A$  is symmetric then  $A_i$  and  $B_i$  are symmetric.
- If  $A$  is symmetric positive definite (SPD) then the matrices  $B_i A$  are orthogonal projections in the inner product  $\langle \mathbf{u}, \mathbf{v} \rangle_A = \mathbf{u}^\top A \mathbf{v}$  (Exercise 6.7).

The  $B_i$  are defined in (6.13) using the exact inverses  $(A_i)^{-1}$ . However, even on each subgrid we will only solve the linear system approximately, and often iteratively. By themselves correction matrices  $B_i$  are not preconditioners of the original systems  $A\mathbf{u} = \mathbf{b}$  because generally they are not invertible, and in fact  $\text{rank}(B_i)$  is bounded by the number of subgrid points  $n_i$ . Efficient DD and multigrid methods are based on the idea that approximations of the corrections  $B_i$  are effective as *components* of preconditioners for the original linear system.

Thus we *compose* (combine) the subgrid corrections  $B_i$  to create a preconditioner  $M^{-1}$ . In the case of  $p = 2$  subgrids, corrections  $B_0, B_1$  can be composed *additively*,

$$M^{-1} = B_0 + B_1, \quad (6.15)$$

or *multiplicatively*,

$$M^{-1} = B_0 + B_1 - B_1 A B_0. \quad (6.16)$$

Figure 6.7 illustrates additive and multiplicative compositions in simple iteration (6.3). We first compute the residual  $\mathbf{r}_k = \mathbf{b} - A\mathbf{u}_k$ . Additive composition applies each  $B_i$  to  $\mathbf{r}$  separately (e.g., solves on  $\Omega_i$  by applying  $A_i^{-1}$ ), and then adds the results to update the solution. Multiplicative composition computes one subspace correction  $B_0\mathbf{r}$  first, then updates the solution and residual, then computes the other correction  $B_1\mathbf{r}$ , and then updates the solution. An easy computation (Exercise 6.5) shows that, for  $p = 2$  subgrids and simple iteration, this multiplicative sequence yields (6.16).

Suppose that two subgrids  $\Omega_0, \Omega_1$  overlap, and that we order the grid points so that those in  $\Omega_0 \setminus \Omega_1$  come first, then those in the overlap  $\Omega_0 \cap \Omega_1$ , and then those in  $\Omega_1 \setminus \Omega_0$ . The additive composition (6.15) can then be visualized as an overlapping sum of subgrid inverses,

$$B_0 + B_1 = \begin{bmatrix} A_0^{-1} & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & A_1^{-1} \end{bmatrix} = \begin{bmatrix} \text{[gray square]} & \text{[white square]} \\ \text{[white square]} & \text{[gray square]} \end{bmatrix}, \quad (6.17)$$

showing how overlapping additive subgrid corrections generalize the block Jacobi method (Chapter 2).

Compositions (6.15) and (6.16) extend straightforwardly to  $p$  subgrid corrections  $B_0, \dots, B_{p-1}$ . The multiplicative composition needs an ordering of the subgrids, recalling the difference between the classical Jacobi and GS iterations. In fact, additive/multiplicative composition generalizes the Jacobi/GS methods from individual entries to subgrids, respectively. Note that even if subgrids are not overlapping, i.e., in block Jacobi, additive and multiplicative compositions are usually different because the system matrix  $A$  generally couples the subgrids when computing the intermediate residuals.

If  $A$  is SPD then the  $B_i$  are also SPD, so the additive composition may be used in symmetric preconditioning (Chapter 2). Likewise, multiplicative composition can be modified to preserve symmetry (Exercise 6.9).

Multiplicative composition computes residuals, i.e., it applies  $A$  to compute residuals, many times as it computes the action of  $M^{-1}$ . By contrast, additive composition only computes a residual once per preconditioner application. This contrast suggests that the multiplicative approach is likely to be a better preconditioner because it uses global (overlapped) information in  $A$  more frequently. For DD preconditioning (below), for example, multiplicative composition requires half as many iterations as additive, again recalling the Jacobi-versus-GS difference (Chapter 2).

On the other hand, the potential for parallelism when using additive composition  $M^{-1} = \sum_i B_i$  should be clear [134]. That is, in the additive case one does not wait for an update from one subgrid before computing the correction for another. In fact, for parallel DD preconditioning PETSc *only* implements the additive composition. However, multiplicative composition remains conceptually and practically important in multigrid and `fieldsplit` preconditioning.

As a preview of things to come, suppose now that the subgrid is a coarse grid covering the original domain. If  $R_C$  is a restriction matrix for this coarse grid then  $A_C = R_C A R_C^\top$  would be a *coarse-grid matrix*. (We will call it a *Galerkin* coarse-grid matrix.) The matrix  $B_C = R_C^\top (A_C)^{-1} R_C$ , which acts on the original grid, is the *coarse-grid correction*. However, we will see that a subgrid *injection*  $R$ , while appropriate to DD, gives a completely unhelpful prolongation  $R^\top$  for multigrid purposes because it introduces high frequencies. Thus we will consider new restriction and prolongation matrices for multigrid schemes. Nonetheless the concepts and notation of subgrid matrices and corrections, and their compositions, apply to both DD and multigrid methods.

## A better Poisson code

The code in Chapter 3 already solves the problem, but here is another structured-grid FD Poisson equation solver. This one works for 1, 2, or 3 dimensions, allows anisotropic coefficients, uses SNES for easy extension to nonlinear elliptic problems, and, most importantly, can exploit DD and GMG preconditioners. It is thus a much better basis for further developments. In fact, parts of this code are reused in solvers for the nonlinear minimal-surface equation (Chapter 7) and for an obstacle problem (Chapter 12).

The code is `ch6/fish.c`.<sup>24</sup> In 3D it solves linear elliptic PDEs of the form

$$-c_x u_{xx} - c_y u_{yy} - c_z u_{zz} = f(x, y, z), \quad (6.18)$$

where  $c_x, c_y, c_z$  are positive constants, subject to arbitrary nonhomogeneous Dirichlet boundary conditions  $u = g$  on  $\partial\Omega$ , where the domain is  $\Omega = (0, L_x) \times (0, L_y) \times (0, L_z)$ . In 1D and 2D it solves the obvious restricted problems.

---

<sup>24</sup>Poisson in French is fish in English.

Code 6.1 shows `main()`. As expected from previous chapters, we create a DMDA for the chosen dimension, defaulting to a  $3 \times 3 \times 3$  grid in 3D. The user can set options using prefix `-fsh_`, including the dimension `-fsh_dim`. We set up a SNES object, supply it with both residual and Jacobian call-backs, and set the default SNES type to KSPONLY type because the problem is linear. The KSP inside the SNES is set to CG because the matrix is SPD. Then we choose an initial iterate (not shown) and call `SNESSolve()` to solve the problem.

```

// create DMDA in chosen dimension
switch (dim) {
    case 1:
        DMCreate1d(PETSC_COMM_WORLD,
                    DM_BOUNDARY_NONE, 3, 1, 1, NULL, &da);
        break;
    case 2:
        DMCreate2d(PETSC_COMM_WORLD,
                    DM_BOUNDARY_NONE, DM_BOUNDARY_NONE, DM_DA_STENCIL_STAR,
                    3, 3, PETSC_DECIDE, PETSC_DECIDE, 1, 1, NULL, NULL, &da);
        break;
    case 3:
        DMCreate3d(PETSC_COMM_WORLD,
                    DM_BOUNDARY_NONE, DM_BOUNDARY_NONE, DM_BOUNDARY_NONE,
                    DM_DA_STENCIL_STAR,
                    3, 3, 3, PETSC_DECIDE, PETSC_DECIDE, PETSC_DECIDE,
                    1, 1, NULL, NULL, NULL, &da);
        break;
    default:
        SETERRQ(PETSC_COMM_SELF, 1, "invalid dim for DMDA creation\n");
}
DMSetApplicationContext(da, &user);
DMSetFromOptions(da);
DMSetUp(da); // call BEFORE SetUniformCoordinates
DMDASetUniformCoordinates(da, 0.0, user.Lx, 0.0, user.Ly, 0.0, user.Lz);

// set SNES call-backs
SNESCreate(PETSC_COMM_WORLD, &snes);
SNESSetDM(snes, da);
DMDASNESSetFunctionLocal(da, INSERT_VALUES,
                         (DMDASNESFunction)(residual_ptr[dim-1]), &user);
DMDASNESSetJacobianLocal(da,
                          (DMDASNESJacobian)(jacobian_ptr[dim-1]), &user);

// default to KSPONLY+CG because problem is linear and SPD
SNESSetType(snes, SNESKSPONLY);
SNESGetKSP(snes, &ksp);
KSPSetType(ksp, KSPCG);
SNESSetFromOptions(snes);

// set initial iterate and then solve
DMGetGlobalVector(da, &u_initial);
InitialState(da, initial, gonboundary, u_initial, &user);
SNESSolve(snes, NULL, u_initial);

```

**Code 6.1.** *c/ch6/fish.c, part I. `main()`: Create DMDA and SNES, and then solve.*

Regarding a possible performance concern, note that `DMDASetUniformCoords()` does use memory for a coordinate `Vec`; in 3D this consumes as much memory as three scalar `Vecs`. While this could be avoided for any structured grid, by generating coordinates on the fly, the strategy

here makes our code performance comparable to unstructured-mesh codes in which coordinate storage is obligatory (Chapter 10).

Next is a subtle change relative to previous examples. As shown in Code 6.2, the grid coming out of `SNESolve()` could be different from the one going in. For that reason we get a new DMDA (`SNESGetDM()`) and Vec (`SNESGetSolution()`) for the solution. Note that the exact solution is also computed on the new grid when evaluating the numerical error.

```
// -snes_grid_sequence could change grid resolution
DMRestoreGlobalVector(da,&u_initial);
DMDestroy(&da);

// evaluate error and report
SNESGetSolution(snes,&u); // SNES owns u; do not destroy it
SNESGetDM(snes,&da_after); // SNES owns da_after; do not destroy it
DMDAGetLocalInfo(da_after,&info);
DMCreateGlobalVector(da_after,&u_exact);
getuexact = getuexact_ptr[dim-1];
(*getuexact)(&info ,u_exact,&user);
VecAXPY(u,-1.0,u_exact); // u <- u + (-1.0) uexact
VecDestroy(&u_exact); // no longer needed
VecNorm(u,NORM_INFINITY,&errinf);
VecNorm(u,NORM_2,&err2h);
```

**Code 6.2.** *c/ch6/fish.c, part II. main(): After `SNESolve()`, get the new grid and solution.*

The “meat” of `fish.c` is, of course, in the residual and Jacobian call-backs. For code reuse they are put in a separate file `poissonfunctions.c`, so we `#include` (not shown) the header `poissonfunctions.h` into `fish.c`. The makefile links the compiled object file (not shown). Arrays of function pointers (Code 6.3) are then used to manage the dimension-dependent call-backs and exact solutions.

```
// arrays of pointers to functions
static DMDASNESFunction residual_ptr[3]
= {(DMDASNESFunction)&Poisson1DFunctionLocal,
(DMDASNESFunction)&Poisson2DFunctionLocal,
(DMDASNESFunction)&Poisson3DFunctionLocal};

static DMDASNESJacobian jacobian_ptr[3]
= {(DMDASNESJacobian)&Poisson1DJacobianLocal,
(DMDASNESJacobian)&Poisson2DJacobianLocal,
(DMDASNESJacobian)&Poisson3DJacobianLocal};

typedef PetscErrorCode (*ExactFcnVec)(DMDALocalInfo*, Vec, PoissonCtx*);

static ExactFcnVec getuexact_ptr[3]
= {&Form1DUExact, &Form2DUExact, &Form3DUExact};
```

**Code 6.3.** *c/ch6/fish.c, part III. Function pointers for each dimension.*

An extract from the header (Code 6.4) includes a context struct declaring pointers to functions which evaluate  $f$  and  $g$ . These functions are regarded as having three spatial inputs, the maximum dimension. The header also declares the residual-evaluation functions `PoissonndDFunctionLocal()`, which compute the discretized function  $\mathbf{F}(\mathbf{u}) = \sigma(-c_x u_{xx} - c_y u_{yy} - c_z u_{zz} - f)$ , with a dimension-dependent scaling factor  $\sigma$  (see below).

```

typedef struct {
    // domain dimensions
    PetscReal Lx, Ly, Lz;
    // coefficients in - cx u_xx - cy u_yy - cz u_zz = f
    PetscReal cx, cy, cz;
    // right-hand-side f(x,y,z)
    PetscReal (*f_rhs)(PetscReal x, PetscReal y, PetscReal z, void *ctx);
    // Dirichlet boundary condition g(x,y,z)
    PetscReal (*g_bdry)(PetscReal x, PetscReal y, PetscReal z, void *ctx);
    // additional context; see example usage in ch7/minimal.c
    void *addctx;
} PoissonCtx;

PetscErrorCode Poisson1DFunctionLocal(DMDALocalInfo *info,
    PetscReal *au, PetscReal *aF, PoissonCtx *user);

PetscErrorCode Poisson2DFunctionLocal(DMDALocalInfo *info,
    PetscReal **au, PetscReal **aF, PoissonCtx *user);

PetscErrorCode Poisson3DFunctionLocal(DMDALocalInfo *info,
    PetscReal ***au, PetscReal ***aF, PoissonCtx *user);

```

**Code 6.4.** *c/ch6/poissonfunctions.h. Context and declarations.*

The FD scheme in `fish.c` is the same as in `ch3/poisson.c`, so we only show the 2D residual-evaluation function (Code 6.5), while Jacobian-evaluation functions `FormdDJacobianLocal()` are not shown at all.

```

PetscErrorCode Poisson2DFunctionLocal(DMDALocalInfo *info, PetscReal **au,
    PetscReal **aF, PoissonCtx *user) {
    PetscInt i, j;
    PetscReal xymin[2], xymax[2], hx, hy, darea, scx, scy, scdiag, x, y,
        ue, uw, un, us;
    DMGetBoundingBox(info->da, xymin, xymax);
    hx = (xymax[0] - xymin[0]) / (info->mx - 1);
    hy = (xymax[1] - xymin[1]) / (info->my - 1);
    darea = hx * hy;
    scx = user->cx * hy / hx;
    scy = user->cy * hx / hy;
    scdiag = 2.0 * (scx + scy); // diagonal scaling
    for (j = info->ys; j < info->ys + info->ym; j++) {
        y = xymin[1] + j * hy;
        for (i = info->xs; i < info->xs + info->xm; i++) {
            x = xymin[0] + i * hx;
            if (i==0 || i==info->mx-1 || j==0 || j==info->my-1) {
                aF[j][i] = au[j][i] - user->g_bdry(x,y,0.0,user);
                aF[j][i] *= scdiag;
            } else {
                ue = (i+1 == info->mx-1) ? user->g_bdry(x+hx,y,0.0,user)
                    : au[j][i+1];
                uw = (i-1 == 0) ? user->g_bdry(x-hx,y,0.0,user)
                    : au[j][i-1];
                un = (j+1 == info->my-1) ? user->g_bdry(x,y+hy,0.0,user)
                    : au[j+1][i];
                us = (j-1 == 0) ? user->g_bdry(x,y-hy,0.0,user)
                    : au[j-1][i];
                aF[j][i] = scdiag * au[j][i]
                    - scx * (uw + ue) - scy * (us + un)
                    - darea * user->f_rhs(x,y,0.0,user);
            }
        }
    }
}

```

```

        }
    }
}
PetscLogFlops(11.0*info->xm*info->ym);
return 0;
}

```

**Code 6.5.** *c/ch6/poissonfunctions.c*. Residual evaluation in 2D.

A crucial idea, new in *fish.c*, is that we discretize the problem on the particular grid supplied at call-back. This is needed to exploit the rediscretization form of GMG (below). The call-back from SNES supplies a `DMDALocalInfo` object to describe the grid (Code 6.5), and `DMGetBoundingBox()` gets domain dimensions for computing the mesh spacings  $h_x, h_y, h_z$ ; this works because we called `DMDASetUniformCoordinates()` in `main()`. Notice that *fish.c* does *not* save any grid-specific information in its context struct, as doing so would be incompatible with rediscretization.

In 2D the entries of the residual are scaled by the cell area  $h_x h_y$  so that all Jacobian entries are  $O(1)$  in the cell spacing. For all dimensions, entries scale as they would in a Galerkin finite element method (Chapters 9 and 10). For example, in the simplest case where  $c_x = c_y = c_z = 1$ , in 1D the diagonal entries are  $h(2/h^2) = 2/h$ , in 2D they are  $h_x h_y 2(1/h_x^2 + 1/h_y^2)$ , and in 3D they are  $h_x h_y h_z 2(1/h_x^2 + 1/h_y^2 + 1/h_z^2)$ . The equations for Dirichlet boundary conditions are also scaled so that the Jacobian has constant diagonal, thereby reducing its condition number. So that the Jacobian is symmetric, the residual evaluation always uses the values of  $g$  at boundary points to reduce the equations; see Chapter 3 on this point.

Compiling and checking for options gets us started:

```

$ cd c/ch6/ && make fish
$ ./fish -help | grep fsh_

```

Option `-fsh_problem` determines which exact solution is used. The default problem `manuexp`, which requires  $c_x = c_y = c_z = 1$ , is also solved in [134]. In 2D the PDE is  $-\nabla^2 u = xe^y$ , with solution  $u(x, y) = -xe^y$ . In 1D we set  $f(x) = e^x$  so  $u(x) = -e^x$ , while in 3D we set  $f(x, y, z) = 2xe^{y+z}$  so  $u(x, y, z) = -xe^{y+z}$ . Exact solution `manupoly` solves the same 2D problem as in *ch3/poisson.c*, with versions for any dimension or coefficients, and thus codes *ch3/poisson.c* and *ch6/fish.c* compute the same result when applied to the same problem:

```

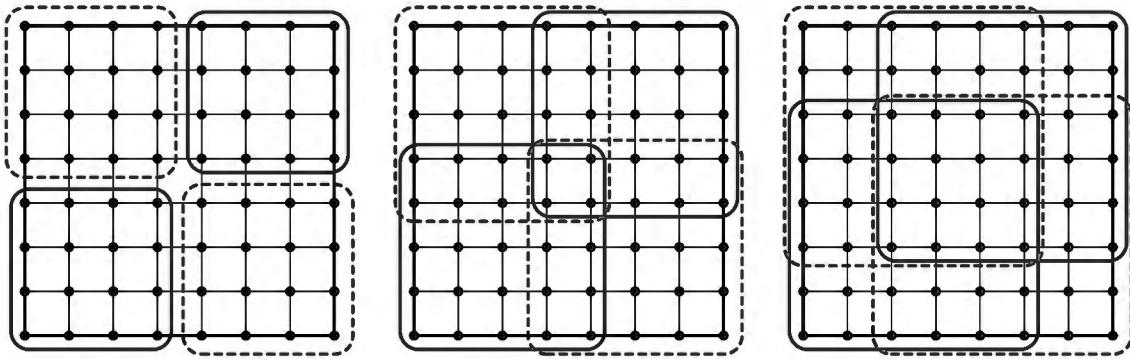
$ (cd ..//ch3/ && make poisson && ./poisson -ksp_converged_reason)
...
Linear solve converged due to CONVERGED_RTOL iterations 7
on 9 x 9 grid: error |u-uexact|_inf = 0.000763959
$ ./fish -fsh_problem manupoly -ksp_converged_reason -da_refine 2
Linear solve converged due to CONVERGED_RTOL iterations 7
problem manupoly on 9 x 9 point 2D grid:
error |u-uexact|_inf = 7.640e-04, |u-uexact|_h = 4.124e-04

```

The  $\infty$ -norm errors are the same (Exercise 6.10) but *fish.c* also computes the  $h$ -weighted  $L^2$  norm of the error, analogous to the continuous  $L^2$  norm.

## Single-level domain decomposition

We can now test subgrid-based preconditioners on the Poisson problem. First, domain decomposition (DD) methods approximately solve PDEs by dividing the domain into (generally)



**Figure 6.8.** Subgrid decompositions of a  $8 \times 8$  grid into  $p = 4$  equal-sized subgrids with overlaps of 0 (left), 1 (middle), and 2 (right) grid points.

overlapping subdomains,  $\Omega = \cup \Omega_i$ , solving the corresponding problem on each subdomain, and combining the results. In PETSC these methods are preconditioners for any finite-dimensional linear systems based on grids and meshes [134]. An original grid, which generates the linear system  $A\mathbf{u} = \mathbf{b}$ , is divided into  $p$  (generally) overlapping subgrids, as shown in Figure 6.8. Given the subgrid restrictions  $R_0, \dots, R_{p-1}$ , we then have subgrid matrices  $A_i = R_i A R_i^\top$  and corrections  $B_i = R_i^\top A_i^{-1} R_i$ . A preconditioner is chosen for each subgrid,  $M_i^{-1} = \mathcal{P}(A_i) \approx A_i^{-1}$ , which defines an approximate subgrid correction  $\tilde{B}_i = R_i^\top M_i^{-1} R_i$ . As usual, the  $M_i^{-1}$  matrices are never assembled; they represent pieces of code which solve systems  $M_i \mathbf{y} = \mathbf{c}$  on each subgrid.

The *alternating Schwarz method* [134], a grid-free iteration in which elliptic PDEs are (exactly) solved on each subdomain  $\Omega_i$  in turn, dates back to 1870 and is the original DD algorithm. The boundary values along  $\partial\Omega_i$  arise by evaluating (i.e., finding the *trace* [51] of) the solutions on the other subdomains  $\Omega_j$  for  $j \neq i$ . Such a domainwise method can be implemented either additively or multiplicatively. In the multiplicative implementation, for example, the subdomains are ordered, and each subdomain solve uses the latest values from the other subdomains.

The methods in PETSC are, however, finite-dimensional and based on subgrids and/or subsets of indices. The *additive Schwarz method* (ASM) preconditioner sums the corrections as in (6.15):

$$M^{-1} = \sum_{i=0}^{p-1} R_i^\top M_i^{-1} R_i. \quad (6.19)$$

Other than in Exercises 6.14 and 6.15, we will only use ASM in this book. (In PETSC the *multiplicative Schwarz method* [134] is only supported in serial.)

Basic ASM options are

```
-pc_type asm -pc_asm_overlap X -sub_pc_type Y
```

The default overlap is  $X = 1$ , and  $X = 0$  is the same as block Jacobi (`-pc_type bjacobi`). Any subdomain preconditioner  $Y$  can be chosen, including direct subdomain solves (`lu`, `cholesky`, `svd`), their incomplete versions (`ilu`, `icc`), and so on.

By default, `-pc_type asm` assigns one subgrid to each MPI rank, thus the number of processes equals the number of subgrids. However, one may increase the number of subgrids (blocks) per process above one with option `-pc_asm_blocks`, which sets the total number of subgrids, indirectly determining the number of subgrids per process. (Option `-pc_asm_print_subdomains` shows the decomposition.)

To reduce the amount of communication the subdomains should have relatively small boundaries. On DM DA structured grids, the number of processors should therefore be a composite

integer, and perfect squares in 2D or cubes in 3D are convenient. For example, the following run divides a  $65 \times 65$  grid into four equal-sized, overlapping  $33 \times 33$  subgrids and solves by CG+ASM+Cholesky:

```
| $ mpiexec -n 4 ./fish -da_refine 5 -pc_type asm -sub_pc_type cholesky
```

If the subdomains are of significant size—much larger than a single point—and if the `-sub_pc_type` method is direct or otherwise reasonably accurate, then many short- and medium-range influences are resolved by a single ASM application. However, one application can only propagate information from each subdomain to its overlapping neighbors, and long-range influences between variables must somehow be resolved if a Krylov iteration is to converge to the solution of the PDE. Thus a preconditioner of the form

```
-pc_type asm -sub_pc_type lu|cholesky|svd
```

can be a robust parallel approach for difficult problems, but the number of Krylov iterations will grow with the number of processors. On the other hand, as we will see below, a coarse grid covering the original domain can generate a correction which communicates the long-range influences and speeds convergence.

A first ASM performance question might ask how the amount of overlap relates to KSP iterations. Consider these runs with overlaps of  $X = 0, 1, 2, 4$  points:

```
| $ mpiexec -n P ./fish -fsh_dim D -da_refine L -ksp_converged_reason \
    -pc_type asm -sub_pc_type cholesky -pc_asm_overlap X
```

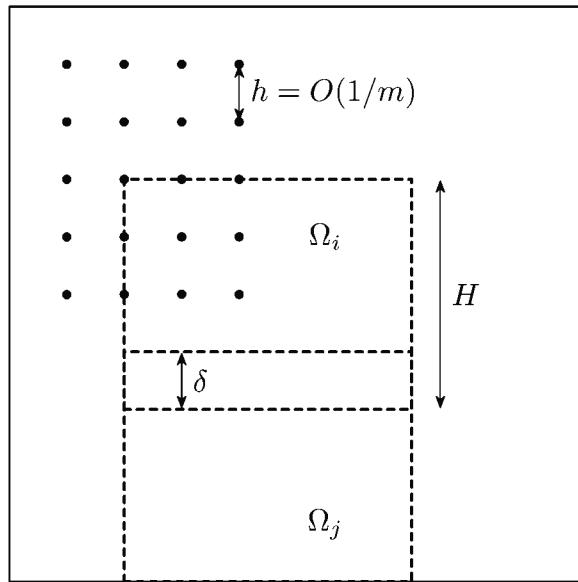
In 2D we use  $P = 4$  processes (subdomains), while in 3D we use  $P = 8$ , so the subdomains are geometrically similar to the original domain. In 2D we test refinement levels  $L = 3, 4, 5, 6, 7$ , but in 3D only  $L = 3, 4, 5$ . The finer-grid 3D runs are quite slow, but Table 6.1 shows the result.

**Table 6.1.** Number of CG iterations from ASM runs of `fish.c` with  $P$  processes, equal to the number of subdomains, and varying overlaps  $X$ . Grids are  $m \times m$  in 2D and  $m \times m \times m$  in 3D.

		Overlap X			
		0	1	2	4
(P = 4)	2D	17	13	11	11
		33	18	13	11
		65	25	18	15
		129	33	21	19
		257	44	28	23
	3D	17	16	11	10
(P = 8)		33	21	15	12
		65	29	20	16

Increasing overlap reduces the number of iterations, but a more subtle observation about Table 6.1 is that the diagonals are roughly constant. For example, there are 18 iterations for a  $65 \times 65$  grid with  $X = 1$ , 19 for a  $129 \times 129$  grid with  $X = 2$ , and 20 for a  $257 \times 257$  grid with  $X = 4$ . This is evidence for one of the established properties of overlapping additive Schwarz methods. Stating it precisely needs notation.

As shown in Figure 6.9, let  $h$  be the grid spacing,  $q$  the overlap in grid points, and  $\delta = qh$  the overlap as a distance. Assuming the domain  $\Omega$  has  $O(1)$  side length, suppose that the grid has  $m$  points along each side and that subdomains have side length  $H$ . Then  $h = O(1/m)$ , and in  $d$  dimensions there are  $m^d$  total grid points and  $(1/H)^d$  subdomains. Note that  $h, \delta, H$  are distances while  $q, m$  are counts.



**Figure 6.9.** Notation for overlapping subdomains and subgrids.

Returning to Table 6.1, the overlap distance  $\delta$  is constant along the diagonals (for  $X > 0$ ). That is, the subdomains and  $H$  are held fixed but  $q$  increases proportionally to  $m$  so  $\delta = qh$  is constant. With this form of refinement, the CG iterations are nearly constant, an indication that the spectra of the preconditioned matrices have comparable clustering as the grid is refined.

For comparison, consider refining the grid while holding each subgrid to a fixed number of points. For this experiment, Table 6.2 was generated by 2D runs as above, with grids of  $m = 17, 33, 65$  points on a side, corresponding to  $L = 3, 4, 5$ , but with a reduced accuracy goal (`-ksp_rtol 1.0e-3`) to avoid the confusing effects of rounding error on CG iterations. The number of processes  $P$ , equal to the number of subdomains, is set proportional to the total number of grid points, i.e.,  $P = O(m^2)$ . In this case  $H$  decreases at the same rate as  $h$ , namely  $O(1/m)$ . The number of grid points on each side of each subdomain,  $H/h$ , is constant. The result in Table 6.2 shows that the number of iterations increases roughly as  $O(1/H)$  if overlap  $X$  is fixed.

**Table 6.2.** Number of KSP iterations from 2D ASM runs of *fish.c*, on  $m \times m$  grids, with  $P$  processes (subdomains). The subgrid size is constant.

P	m	Overlap X			
		0	1	2	4
4	17	10	6	6	6
16	33	18	11	12	14
64	65	33	20	26	24

The results in Tables 6.1 and 6.2 supply evidence for three general properties of overlapping Schwarz methods [134]:

- (i) convergence is poor for block Jacobi (no overlap) but improves with increasing overlap,
- (ii) if distances  $\delta$  and  $H$  are fixed then the number of iterations is bounded independently of  $h$ , and
- (iii) the number of iterations grows as  $1/H$ , the number of subdomains along each side of the domain.

The first point can be understood by considering  $A^{-1}$  as the discrete Green's function of the elliptic operator, the effect of which is dominated by short-range interactions. Approximating a Green's function by a finite kernel of positive range in the overlap directions is much better than by one of zero range. Property (ii), which says that overlapping Schwarz methods have a kind of good scaling as the grid is refined, is proved by showing that the condition number of the preconditioned operator is bounded independently of  $h$  [134]. Regarding (iii), at each iteration ASM can communicate information from each subdomain only to its neighbors. Thus  $O(1/H)$  iterations, the number of subdomains along a side, are needed to communicate the influence of boundary conditions and source terms from one side of  $\Omega$  to the other.

Regarding the performance of ASM, results in Tables 6.1 and 6.2 already suggest that using direct solvers on the subdomains is not a good strategy for 3D grids. Note that adding nested dissection ordering (Chapter 2) to the Cholesky direct solver on each subgrid is a reasonably efficient direct subgrid solver:

```
-pc_type asm -sub_pc_type cholesky -sub_pc_factor_mat_ordering_type nd
```

For the  $33 \times 33 \times 33$  grids in Table 6.1 this is more than ten times faster than the natural-ordered solver, and the ratio increases for finer grids.

The above ASM+Cholesky solver scales much better than  $O(N^3)$  both because of banding and from variable reordering. Suppose we consider a refinement path of 3D runs from Table 6.1 with fixed overlap distance  $\delta$ , i.e., the runs with fixed  $p = 8$  subgrids and `-da_refine 4|5|6 -pc_asm_overlap 1|2|4`, respectively. For such runs, because of the general ASM properties above, the KSP iterations are essentially constant. The amount of work, measured by the total number of flops for the KSP solver stage (`-log_view | grep KSPSolve`), scales as  $O(N^{1.41})$ . When  $N$  increases by a factor of 8 (refinement by two in each direction) the work increases by a factor of about 18, instead of 500 for an  $O(N^3)$  algorithm. As the reader can confirm, however, memory usage soon brings further refinements to a halt because the reordered Cholesky factors fill-in a band of significant width around the diagonal. (The next refinement `-da_refine 7 -pc_asm_overlap 8` runs out of memory on the author's laptop.)

There is, however, no reason to only use direct solvers on the subgrids; the solve on each subgrid is merely preconditioning anyway. Incomplete subgrid solvers, in particular, yield significantly better overall memory usage and performance. Consider ICC subgrid solves:

```
-pc_type asm -sub_pc_type icc
```

We do the following 3D runs to test performance:

```
$ mpiexec -n 8 ./fish -fsh_dim 3 -da_refine L -ksp_converged_reason \
    -pc_type asm -sub_pc_type icc -pc_asm_overlap X
```

Here  $L$  and  $X$  are again chosen to give constant overlap distance  $\delta$  under refinement. Specifically, we do four runs with  $L = 4, 5, 6, 7$  and  $X = 1, 2, 4, 8$ , respectively. The number of KSP iterations grows, despite the fixed-distance overlaps, because the subgrid solves are no longer exact. The work (i.e., flops) in these ASM+ICC runs scales as  $O(N^{1.30})$ , a better rate than for ASM+Cholesky, but the most significant improvement is the reduced memory usage from avoiding fill-in, which also improves run time. In fact, the ASM+ICC run on a grid of  $N = 257^3$  points is about 10 times faster than the ASM+Cholesky run on the smaller  $N = 65^3$  grid.

ASM-preconditioned Krylov iterations are thus important solvers, especially in the context of parallel computations [134]. However, for 2D and 3D Poisson problems the DD solvers considered so far cannot achieve  $O(N)$  work for  $N$  unknowns, the property called “optimality” in Chapter 7. Actually, if optimality applied to the PC used on each subgrid then property (ii) above would allow ASM to scale as  $O(N)$  if the number of blocks  $P$  and the overlap distance  $\delta$

were held fixed as the grid was refined. However, for elliptic PDEs in 2D and 3D we have, for now, no candidate solver which is optimal on the subgrids.

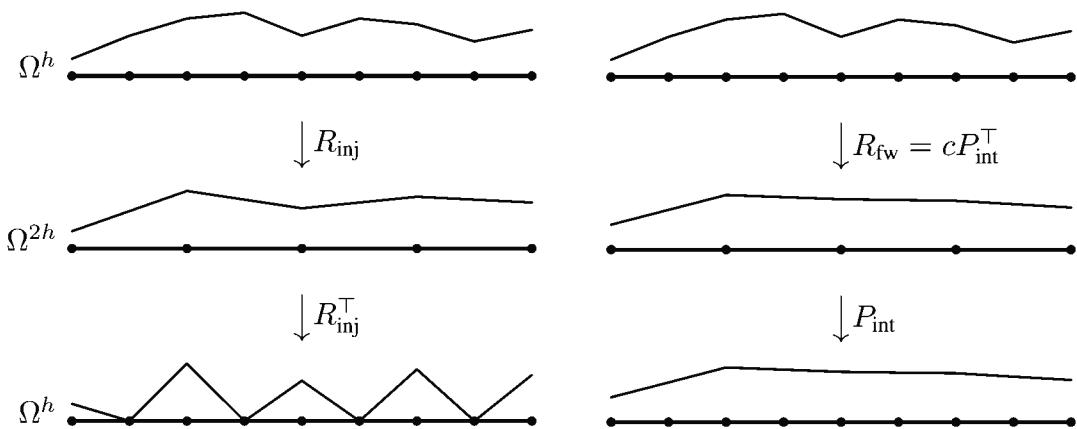
To get significantly better preconditioning than our current best performers for the Poisson equation, namely `-ksp_type cg -pc_type icc` in serial (Chapter 3), and `-ksp_type cg -pc_type asm -sub_pc_type icc` in parallel, we need to address the essential flaw in single-level DD. This is point (iii) above: an ASM preconditioner is slow to communicate long-range, low-frequency information about the solution. The key improvement is well known [21, 26, 45, 49] and addressed next.

## Coarse grids

Consider a structured grid  $\Omega^h$  covering the domain  $\Omega$  with  $N^h$  total points and spacing  $h$  in each direction. The *coarse grid*  $\Omega^{2h} \subset \Omega^h$  consists of every other point of  $\Omega^h$ , with spacing  $2h$ , from a *coarsening factor* of 2. In  $d$  dimensions the coarse grid has  $N^{2h} \approx N^h/2^d$  total points. Now that there is a coarse grid,  $\Omega^h$  is called the *fine grid*.

A few comments on this notation may help. We denote grid quantities using a grid-spacing superscript. The precise number of points in  $\Omega^h$  or  $\Omega^{2h}$  depends on how the boundary is handled, but the formula  $N^{2h} = N^h/2^d$  is exact if boundaries are periodic and the number of grid points in each direction is even. Though a coarsening factor of 2 is the most common case, and the default for DMDA, other ratios would not change the ideas below in any important way.

Restriction by injection  $R_{\text{inj}}$  takes a vector  $v$  of values on  $\Omega^h$  and extracts the values that live on  $\Omega^{2h}$ . Though injection is a reasonable way to transfer a vector to the coarse grid, its transpose  $R_{\text{inj}}^\top$  should not be used as the prolongation. It puts zeros at those points of the fine grid which are between the points of the coarse grid, and, considering vectors as piecewise-linear functions, this maps smooth functions to very rough ones (Figure 6.10 and Exercise 6.8; compare Figure 6.5).



**Figure 6.10.** Left: The transpose of injection  $R_{\text{inj}}$  is useless as a prolongation because it introduces high frequencies. Right: Linear interpolation  $P_{\text{int}}$  makes an effective prolongation, and its transpose (full-weighting) works for restriction.

However, the operation of piecewise *interpolation*  $P_{\text{int}}$  makes a well-behaved prolongation from  $\Omega^{2h}$  to  $\Omega^h$ . In 1D interpolation acts by averaging coarse-grid values to generate fine-grid values between. The coarse-grid values themselves are unaltered, and it follows that  $\|P_{\text{int}}\|_\infty = 1$  and that  $P_{\text{int}}$  has full rank  $N^{2h}$ .

For 2D and 3D grids, various low-order piecewise-polynomial interpolation schemes might generalize piecewise-linear interpolation in 1D. For example, on a DMDA structured grid the default operator  $P_{\text{int}}$ , in any dimension, is  $Q_1$  interpolation, with the same meaning as for the finite

element method of Chapter 9. Here we avoid the details in 2D and 3D, but show the 1D case concretely.

**Example 6.4.** In 1D, on a grid  $\Omega^h$  with  $N^h = 9$  points, with a coarse grid of  $N^{2h} = 5$  points, linear interpolation is the  $9 \times 5$  matrix

$$P_{\text{int}} = \begin{bmatrix} 1 & & & & \\ 1/2 & 1/2 & & & \\ & 1 & & & \\ & 1/2 & 1/2 & & \\ & & 1 & & \\ & & 1/2 & 1/2 & \\ & & & 1 & \\ & & & 1/2 & 1/2 \\ & & & & 1 \end{bmatrix}.$$

An example action of  $P_{\text{int}}$  appears in Figure 6.10.

By our definition, the transpose  $P_{\text{int}}^\top$  is a restriction because it has nonnegative entries and full row rank. If the column sums of  $P_{\text{int}}$  are used as row scalings then the resulting restriction is called “full-weighting” [26]. Specifically, if  $\text{diag}(\mathbf{c})$  denotes the matrix with vector  $\mathbf{c}$  on its diagonal, full-weighting restriction is

$$R_{\text{fw}} = \text{diag}(\mathbf{c}) P_{\text{int}}^\top \quad \text{where } c_j = \frac{1}{\sum_i (P_{\text{int}})_{ij}}, \quad (6.20)$$

which computes coarse-grid values by averaging. By construction,  $\|R_{\text{fw}}\|_\infty = 1$ .

**Example 6.5.** Continuing the above example,

$$R_{\text{fw}} = \begin{bmatrix} 2/3 & 1/3 & & & \\ & 1/4 & 1/2 & 1/4 & \\ & & 1/4 & 1/2 & 1/4 \\ & & & 1/4 & 1/2 & 1/4 \\ & & & & 1/3 & 2/3 \end{bmatrix}.$$

Figure 6.10 shows that  $R_{\text{fw}}$  is smoothing compared to injection  $R_{\text{inj}}$ .

Thus the multigrid algorithms used in this book, described in the next section, do not use injection  $R_{\text{inj}}$  as the fine-to-coarse restriction operation. Instead either an interpolation ( $P_{\text{int}}$ ) or a smoothing restriction (e.g.,  $R_{\text{fw}}$  or  $P_{\text{int}}^\top$ ) is formed as a `Mat`, and the sizes of this `Mat` indicate to PETSc whether it is a prolongation or a restriction,<sup>25</sup> and the transpose is used for the other operation. All this occurs internally for DMDA-based structured grids and DMplex unstructured grids (Chapter 13) once the grid hierarchy is established.

Now, given the fine-grid matrix  $A^h$ , and supposing interpolation and restriction operators are fixed, there are two ways to define the coarse-grid matrix:

- The *Galerkin* approach: Construct the subgrid matrix the same way as in DD. Use interpolation for prolongation and its transpose for restriction:

$$\hat{A}^{2h} = P_{\text{int}}^\top A^h P_{\text{int}}. \quad (6.21)$$

<sup>25</sup>The documentation for `PCMGSetRestriction()` and `PCMGSetInterpolation()` say “One can pass in the interpolation matrix or its transpose; PETSc figures out from the matrix size which one it is.”

- The *rediscretization* approach: Assume we already have code that converts a grid on  $\Omega$  into a discretization of the PDE problem. Apply this code on the coarse grid:

$$A^{2h} \text{ is computed by applying our code on } \Omega^{2h}. \quad (6.22)$$

Formula (6.21), option `-pc_mg_galerkin`, is called “Galerkin” because it projects onto a smaller space the same way a Galerkin finite element method (Chapters 9 and 10) projects a PDE problem into a finite-dimensional subspace [49]. The formula used by PETSc’s geometric multigrid (next section) is (6.21) and not  $R_{\text{fw}} A^h P_{\text{int}}$ , as the reader might expect, but this turns out to be an unimportant difference because only the coarse grid *correction* must be scaled like  $(A^h)^{-1}$ . (See formula (6.23) below and Exercises 6.17 and 6.18.) Instead of applying the factors in matrix-vector products, product (6.21) is multiplied out to generate a sparse matrix  $\hat{A}^{2h}$ . Note that  $\hat{A}^{2h}$  uses  $2^d$  times less memory than  $A^h$ , and that a coarse-grid Mat is formed in both approaches.

We will use rediscretization approach (6.22) most frequently. Typically our code includes a SNES call-back which generates the Jacobian, and, in structured-grid cases based on `DMDASNESSetJacobianLocal()`, an argument to the call-back is a `DMDALocalInfo` struct which describes the grid. The call-back never “knows” whether this grid is coarser than the original fine grid, but it forms the Jacobian based on the provided grid information.

Regarding symmetry, the Galerkin matrix  $\hat{A}^{2h}$  is symmetric if  $A^h$  is symmetric. On the other hand, if user code generated a symmetric matrix  $A^h$  for the original (fine) grid then it should also produce a symmetric coarse grid matrix  $A^{2h}$ .

**Example 6.6.** In 1D `fish.c` solves the Poisson equation  $-u_{xx} = f(x)$  using the simple FD scheme:

$$-u_{j-1} + 2u_j - u_{j+1} = h^2 f(x_j).$$

If  $\Omega^h$  is the 9-point grid with spacing  $h = 1/8$  then the coarse grid  $\Omega^{2h}$  has 5 points and spacing  $2h = 1/4$ . Using the Galerkin approach (6.21), and the prolongation  $P_{\text{int}}$  in Example 6.4, the coarse-grid matrix is

$$\hat{A}^{2h} = \begin{bmatrix} 20 & & & & \\ & 8 & -4 & & \\ & -4 & 8 & -4 & \\ & & -4 & 8 & \\ & & & & 20 \end{bmatrix}.$$

The coarse-grid matrix (6.22) from rediscretization differs only on the boundary:

$$A^{2h} = \begin{bmatrix} 8 & & & & \\ & 8 & -4 & & \\ & -4 & 8 & -4 & \\ & & -4 & 8 & \\ & & & & 8 \end{bmatrix}.$$

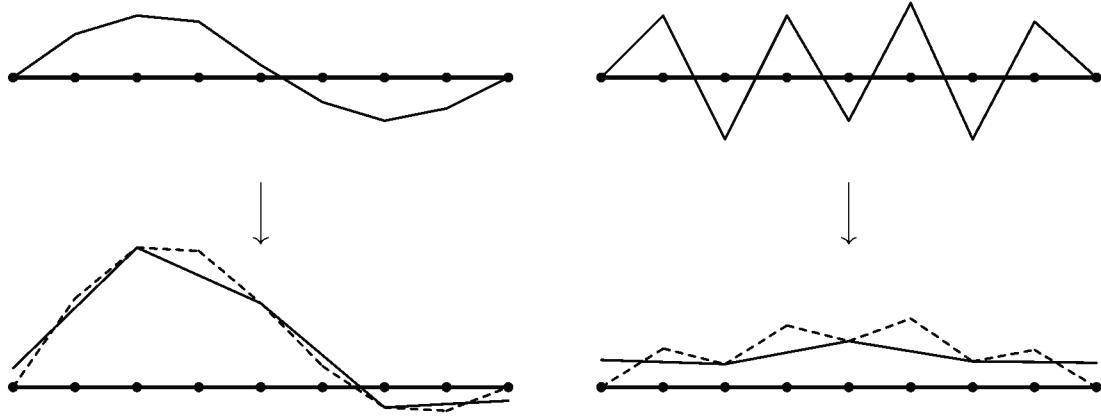
The latter has a smaller condition number.

The  $N^h \times N^h$  coarse-grid correction matrix is

$$B^{2h} = P_{\text{int}}(A^{2h})^{-1}P_{\text{int}}^\top. \quad (6.23)$$

We can use rediscretization  $A^{2h}$  or Galerkin  $\hat{A}^{2h}$ , but either way we denote the correction by  $B^{2h}$  from now on.

What does the result of applying  $B^{2^h}$  look like, relative to solving the problem by applying  $(A^h)^{-1}$ ? The answer depends on the smoothness of the vector, and in this case “smooth” means near the range of  $P_{\text{int}}$ , that is, like a piecewise-linear function on the coarse grid. As shown in Figure 6.11,  $B^{2^h}\mathbf{r}$  and  $(A^h)^{-1}\mathbf{r}$  are close to each other when  $\mathbf{r}$  is smooth, but significantly different otherwise. In the case shown, the relative difference  $\|B^{2^h}\mathbf{r} - (A^h)^{-1}\mathbf{r}\|_2 / \|(A^h)^{-1}\mathbf{r}\|_2$  is three times larger for the less smooth vector.



**Figure 6.11.** Given a smooth vector  $\mathbf{r}$  on the fine grid (left), the result of applying  $B^{2^h}$  (solid) is close to what we get from  $(A^h)^{-1}$  (dashed). For  $\mathbf{r}$  with strong high frequencies (right) the results are further apart.

Consider a step of the simple iteration

$$\mathbf{u}_{k+1} = \mathbf{u}_k + B^{2^h}(\mathbf{b} - A^h \mathbf{u}_k), \quad (6.24)$$

computed as a sequence of steps:

$$\begin{aligned} \mathbf{r}^{2^h} &= P_{\text{int}}^\top(\mathbf{b} - A^h \mathbf{u}_k) && \text{restrict the fine-grid residual to the coarse grid,} \\ A^{2^h} \mathbf{e}^{2^h} &= \mathbf{r}^{2^h} && \text{solve the coarse-grid error equation, and} \\ \mathbf{u}_{k+1} &= \mathbf{u}_k + P_{\text{int}} \mathbf{e}^{2^h} && \text{interpolate the solution as a fine-grid update.} \end{aligned}$$

In other words, we move the residual to the coarse grid, solve the error equation there, and bring back the result as a correction. Iteration (6.24) will be the key step in multigrid.

The first and last factors in a coarse-grid correction (6.23), namely local averaging ( $P_{\text{int}}^\top$ ) and piecewise-polynomial interpolation ( $P_{\text{int}}$ ), are  $O(N^h)$  operations with a small constant. Applying the middle factor  $(A^{2^h})^{-1}$ , i.e., solving the coarse-grid problem, can be done approximately by a preconditioner  $M^{2^h} \approx A^{2^h}$ . Relative to the fine grid, the number of unknowns has decreased by  $2^d$ , thus the coarse-grid solve should be fast. However, the benefits of a coarse-grid correction are at risk if the solver is expensive (e.g., direct) or if parallelizing the coarse-grid solve requires a large amount of communication [144]. (We return to the parallel implementation of multigrid in the next chapter.)

The rank of  $B^{2^h}$  is at most  $N^{2^h} = \text{rank}(P_{\text{int}})$  so the coarse-grid correction cannot be used alone as a preconditioner  $M^{-1}$ . To build a preconditioner we compose  $B^{2^h}$  with other corrections. The best-known example is multigrid, which multiplicatively composes the coarse-grid correction with smoothers; see the next section.

Another example is the *additive two-level overlapping* DD scheme of Dryja and Widlund [45]. It adds a coarse-grid correction to the ASM corrections (6.19),

$$M^{-1} = P_{\text{int}}(M^{2^h})^{-1}P_{\text{int}}^\top + \sum_{i=0}^{p-1} R_i^\top(M_i)^{-1}R_i. \quad (6.25)$$

In a two-level scheme like (6.25) the coarsening factor is a free parameter. The coarse grid might have spacing  $2h$ , as in (6.25), and as demonstrated later in this chapter, but at the other extreme the coarse grid might have only one grid point per ASM subdomain.

Two-level method (6.25) fixes a flaw of single-level DD. Suppose that the grid is refined ( $h \rightarrow 0$ ) such that the number of grid points per subdomain is held fixed and the number of subdomains increases ( $p \rightarrow \infty$ ). Then the number of Krylov iterations grows as  $O(1/H)$ , where  $H$  is the size of the subdomains; see page 150. Dryja and Widlund [45, 134] showed that, under reasonable conditions on the overlap, adding a coarse grid correction as in (6.25) causes the condition number of the preconditioned operator  $M^{-1}A$ , and thus the number of Krylov iterations, to be bounded independently of  $H$  and  $h$ . However, the disadvantage of a two-level scheme like (6.25) is that as the grid is refined either the coarse-grid solution or the subgrid (subdomain) solutions become expensive. If we increase the number of subdomains ( $p \rightarrow \infty$ ) but hold their number of points fixed ( $H/h$  fixed) then the size of the coarse-grid problem gets large as  $h \rightarrow 0$ . On the other hand, if we keep the coarse-grid size fixed ( $p$  fixed) then the size of the subgrid problems must get large ( $H/h \rightarrow \infty$ ) as  $h \rightarrow 0$ .

However, if we can have two levels of grids then we can have many. Also, we should exploit the smoothing property of classical iterations. Combining these ideas gives multigrid.

## Geometric multigrid

Multigrid combines three conceptual threads which are now in hand:

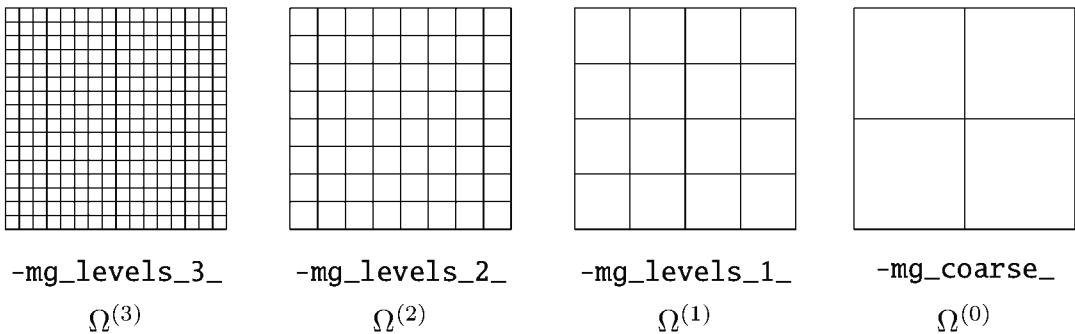
- (i) Inexpensive classical iterations like Jacobi, GS, SOR, and Chebyshev tend to smooth the residual in a few iterations.
- (ii) A coarse-grid correction does a good job of approximating the fine-grid solution when acting on a smooth residual.
- (iii) The restriction and prolongation parts of the coarse-grid correction are inexpensive.

These threads combine first into a preconditioner called the *two-grid scheme*; see [26, Chapter 3] or [144, section 2.23]. It approximately solves the fine-grid system  $A^h\mathbf{u} = \mathbf{b}^h$  by multiplicative composition of smoothing iterations and a coarse-grid correction, defining a fine-grid update  $\mathbf{v} = \text{TWOGRID}(A^h, \mathbf{b}^h, \mathbf{w})$ :

```
function TWOGRID( $A$ ,  $\mathbf{b}$ ,  $\mathbf{w}$ )
     $\mathbf{v} \leftarrow \mathbf{w}$ 
    for  $k = 1, 2, \dots, \nu_1$  pre-smoothing  $\nu_1$  times
         $\mathbf{v} \leftarrow \mathcal{S}_1(A, \mathbf{b}, \mathbf{v})$ 
     $\mathbf{r}^{2h} \leftarrow P_{\text{int}}^\top(\mathbf{b} - A\mathbf{v})$  coarse-grid
    solve  $A^{2h}\mathbf{z}^{2h} = \mathbf{r}^{2h}$  correction
     $\mathbf{v} \leftarrow \mathbf{v} + P_{\text{int}}\mathbf{z}^{2h}$ 
    for  $k = 1, 2, \dots, \nu_2$  post-smoothing  $\nu_2$  times
         $\mathbf{v} \leftarrow \mathcal{S}_2(A, \mathbf{b}, \mathbf{v})$ 
    return  $\mathbf{v}$ 
```

(6.24)

Note we suppress the “ $h$ ” superscript on fine-grid quantities.



**Figure 6.12.** The 2D grids used in `./fish -da_refine 3 -pc_type mg`.

Functions  $\mathcal{S}_1$  and  $\mathcal{S}_2$  denote *smoothers*, possibly distinct, which combine a Krylov method and a preconditioner. For example, the GS smoother (6.8) is a simple iteration (Chapter 2)

$$\mathcal{S}(A, \mathbf{b}, \mathbf{v}) = \mathbf{v} + M^{-1}(\mathbf{b} - A\mathbf{v}) \quad (6.26)$$

using  $M = D + L$ .

The reason for *pre-smoothing* ( $\mathcal{S}_1$ ) is evident: the coarse-grid correction accurately solves the fine-grid problem only if it acts on a smooth residual. However, the result of the coarse-grid correction is an update  $P_{\text{int}}\mathbf{z}^{2^h}$ , in the image of  $P_{\text{int}}$ , which generally can benefit from smoothing on the fine grid. Thus *post-smoothing* ( $\mathcal{S}_2$ ) is also used in multigrid cycles in this chapter.

The two-grid scheme can be better understood by examining how it propagates error. Suppose the smoothers are simple iterations (6.26), and let  $\mathbf{e} = \mathbf{w} - \mathbf{u}$  and  $\mathbf{e}' = \mathbf{v} - \mathbf{u}$  be the errors before and after the scheme is applied. If  $B^{2^h}$  denotes the coarse-grid correction (6.23) then (Exercise 6.19)

$$\mathbf{e}' = (I - M_2^{-1}A^h)^{\nu_2} (I - B^{2^h}A^h) (I - M_1^{-1}A^h)^{\nu_1} \mathbf{e}. \quad (6.27)$$

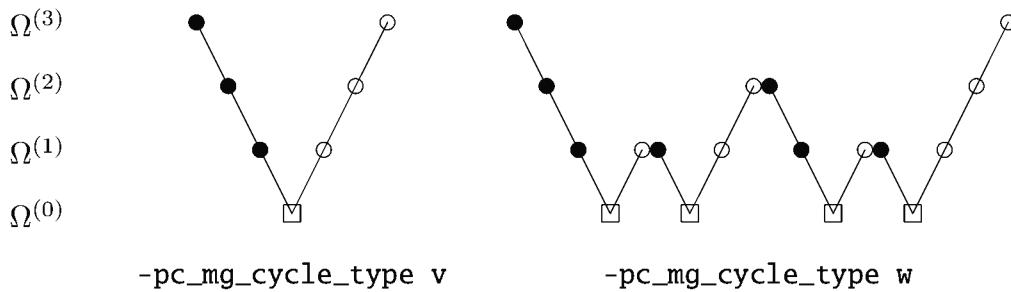
One can see from this product, at least informally, why there is a decrease in error norm. Suppose that the smoothers at least cause no norm increase, e.g.,  $\| (I - M_i^{-1}A^h)^{\nu_i} \| \leq 1$  in some matrix norm, and that they map into the subspace of smooth vectors, specifically those vectors that are in the range of  $P_{\text{int}}$ . Also suppose that the coarse-grid correction  $B^{2^h}$  nearly solves the problem for smooth vectors in the sense that  $B^{2^h}A^h \approx I$ . In particular, assume  $\|(I - B^{2^h}A^h)\mathbf{w}\| \leq \delta \|\mathbf{w}\|$  for some  $\delta < 1$  and all  $\mathbf{w}$  in the range of  $P_{\text{int}}$ . It follows that  $\|\mathbf{e}'\| \leq \delta \|\mathbf{e}\|$ .

Using Fourier analysis and specific smoothers one may prove for the Poisson equation that  $\delta$  is bounded below one, independently of the grid spacing  $h$ . For example, [144, section 2.1.3] shows this for red-black GS. We will not pursue such analysis further, instead preferring demonstrations with actual codes.

We have not addressed the cost of solving the linear system on the coarse grid. A direct solve would be acceptable only if the coarse grid has few points. On the other hand, the modification to the two-grid scheme needed to deal with large coarse grids is obvious: we apply a two-grid scheme to the coarse-grid problem as well, continuing recursively until the coarse grid is acceptably small for a direct solve. This idea is *geometric multigrid* (GMG), the preconditioner `-pc_type mg`.

Figure 6.12 shows the grids constructed when running `fish.c` with GMG and three levels of coarsening:

```
| $ ./fish -da_refine 3 -pc_type mg
```



**Figure 6.13.** Four-level V- and W-cycles. Solid dots are the down-smoother ( $S_1^{\nu_1}$ ), circles are the up-smoother ( $S_2^{\nu_2}$ ), and squares the coarse-grid solve.

The default method for recursively solving the coarse-grid problem is the *V-cycle* (Figure 6.13), which descends to the next-coarser grid once per level:

```
function VCYCLE(A, b, w, l)
  if l == 0
    solve Av = b, e.g., by a direct solver
  else
    v ←  $S_1^{\nu_1}(w)$ 
    rC ←  $P_{\text{int}}^\top(b - Av)$ 
    form  $A^C$ 
    zC ← VCYCLE( $A^C$ , rC, 0, l - 1)
    v ← v +  $P_{\text{int}}z^C$ 
    v ←  $S_2^{\nu_2}(v)$ 
  return v
```

Observe that  $\text{VCYCLE}(A, b, w, 1) = \text{TWOGRID}(A, b, w)$ , and note that  $\nu_i$  steps of smoothing using  $S_i$  are now denoted as powers. Also note that the total number of levels is  $l + 1$ .

One constructs a DMDA structured-grid hierarchy by defining a base grid and then refining a certain number of times. The base grid is determined by parameters to `DMDACreateXd()`, but these may be overridden by options `-da_grid_x` and `-da_grid_y`. The number of refinements is from option `-da_refine`.

The number of grid levels used in multigrid is set by `-pc_mg_levels`, but it defaults to the number of grids in the hierarchy. The value `-pc_mg_levels` determines how far we descend from the fine grid. Thus the coarsest grid  $\Omega^{(0)}$  in the multigrid cycle is defined by descending from the finest-constructed grid. It is common in multigrid usage for  $\Omega^{(0)}$  to not be the base grid in the hierarchy.

In terms of 2D PETSC options,

```
-pc_type mg -da_grid_x MX -da_grid_y MY -da_refine Z -pc_mg_levels L
```

implies a hierarchy with a base grid of  $\text{MX} \times \text{MY}$  points and a fine grid  $\Omega^{(L)}$  of  $2^Z \text{MX} \times 2^Z \text{MY}$  points. The GMG coarse grid  $\Omega^{(0)}$  has dimensions  $2^{Z-L+1} \text{MX} \times 2^{Z-L+1} \text{MY}$ . By default, if `-pc_mg_levels` is not specified then  $L=Z+1$  and thus  $\Omega^{(0)}$  is the  $\text{MX} \times \text{MY}$  base grid. Figure 6.12 shows an example in which defaults are used for `MX`, `MY`, and `L`, and `DMDACreate2d()` was called with a  $3 \times 3$  base grid. Note that grids can be viewed at run time by option `-dm_view draw`.

We have two choices for forming the matrix  $A^C$  at each level. The Galerkin approach (6.21) is option `-pc_mg_galerkin`, but the rediscretization approach (6.22) is the default. We also have choices for solving the coarsest-grid problem. One may use any combination of `-mg_coarse_ksp_type` and `-mg_coarse_pc_type`. In the examples in this chapter we only

consider direct solvers with `KSP=preonly` and `PC=lu` or `PC=cholesky`, but see the discussion of parallel multigrid in the next chapter.

## Multigrid cycle types and costs

However, one may instead apply a *W-cycle* (`-pc_mg_cycle_type w`) in which, after the finest level, the algorithm descends to the next-coarser level twice (Figure 6.13 and Exercises 6.20 and 6.21). Also, one may apply two or more cycles as the preconditioner. Because the residual is computed at each level, with correction from the next-coarser level, these are multiplicatively composed actions, and thus V- and W-cycle preconditioners correspond to options

```
-pc_mg_cycle_type X -pc_mg_type multiplicative -pc_mg_multiplicative_cycles K
```

where `X` is `v` or `w`. However, the default number of cycles per preconditioner application is `K = 1`.

W-cycles are effective preconditioners but each one is more expensive than a V-cycle. It is often the case that doing one extra V-cycle-preconditioned KSP iteration gives the same residual norm reduction as switching to W-cycles, but in less time [144]. Furthermore, in parallel W-cycles are usually a bad idea because many returns to the coarse grid (Figure 6.13) require copious communication between processes while doing little arithmetic. Thus V-cycles are the default for `-pc_type mg`.

On the other hand, how much work is done in these cycles? We adopt a straightforward model for computational cost from [144, section 2.4.3], as follows. Define

- $\gamma$  is the *cycle index*, with  $\gamma = 1$  for a V-cycle and  $\gamma = 2$  for a W-cycle;
- $W_l$  is the *work*, i.e., number of flops, from level  $l$  down;
- $W_k^{k-1}$  is the work done by smoothers at level  $k$ , and in restricting/interpolating to the next coarser level  $k - 1$ , excluding the solution of the problem at level  $k - 1$ ; and
- $N_k$  is the number of unknowns at level  $k$ , i.e.,  $|\Omega^{(k)}|$ .

The work of solving the coarsest-grid problem  $W_0$  will not be examined further here; it is a fixed cost. Let us also assume that

- $N_k = 2^d N_{k-1}$  in dimension  $d$ , and
- there is  $C > 0$  independent of  $k$  such that

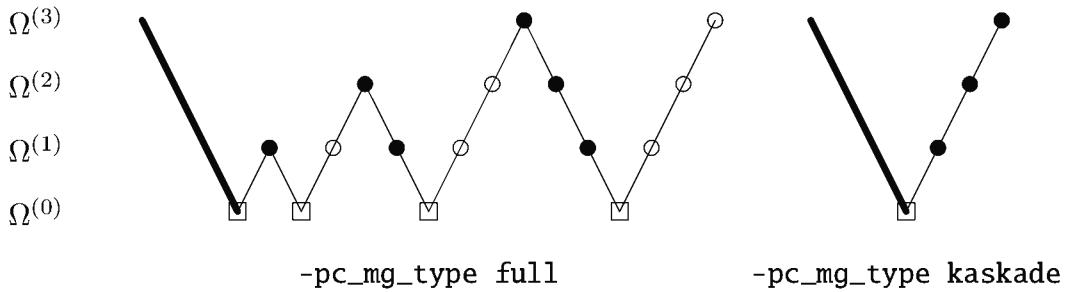
$$W_k^{k-1} \leq CN_k. \quad (6.28)$$

Assumption (6.28) says that the work of smoother iterations and restriction/interpolation operations scales linearly with the number of points in the level  $k$  grid. (Note (6.28) implies that the number  $\nu_k$  of smoother iterations is bounded;  $\nu_k \leq 3$  in most practical usage.)

From these definitions we can write  $W_1 = W_1^0 + W_0$ , and at finer levels  $W_k = W_k^{k-1} + \gamma W_{k-1}$ , from which it follows by induction that

$$W_l = \gamma^{l-1} W_0 + \sum_{k=1}^l \gamma^{l-k} W_k^{k-1}.$$

From our assumptions,  $N_l = (2^d)^{l-k} N_k$  for the finest grid. Now using (6.28) and summing the



**Figure 6.14.** One full (left) or Kaskade (right) cycle for a `-da_refine 3 -pc_type mg` run. The initial coarsening phase only restricts the residuals.

geometric series gives (Exercise 6.22)

$$W_l \leq \gamma^{l-1} W_0 + C N_l \frac{2^d}{2^d - \gamma}. \quad (6.29)$$

Suppose we fix the coarsest grid and make  $N_l \rightarrow \infty$  by increasing the cycle depth  $l$ . For V-cycles ( $\gamma = 1$ ) inequality (6.29) says

$$W_l = O(N_l) \text{ as } N_l \rightarrow \infty. \quad (6.30)$$

That is, the work of a V-cycle is  $O(N_l)$  where  $N_l$  is the number of grid points in the fine grid.

Expression (6.29) reveals the difference between V- and W-cycles. In W-cycles with  $\gamma = 2$ , from the leading term  $\gamma^{l-1} W_0 = 2^{l-1} W_0$  the number of returns to the coarsest grid is exponential in  $l$ . In most serial computations with  $d > 1$ , because  $W_0 \ll W_k^{k-1}$  for larger  $k$  (i.e., finer levels in the cycle), the leading term can be ignored. However, if the cost  $W_0$  is large, including the time needed to communicate in parallel (Chapter 7), then a W-cycle may be expensive.

Consider V-cycles in the case where  $W_0$  is negligible and assume  $W_k^{k-1} = CN_k$ . Then this model implies  $W_l \approx CN_l 2^d / (2^d - 1)$ . It follows that the cost of a V-cycle is very close to the cost of the work on the finest level only. In 3D we have  $W_l \approx (8/7)W_l^{l-1}$  and in 2D  $W_l \approx (4/3)W_l^{l-1}$ .

Multigrid V- and W-cycles each start at the finest grid and do smoothing steps before restricting the residual to coarser grids. A different strategy is to start at the coarsest grid, generate an inexpensive “first guess” there, and then interpolate the solution itself upward to finer grids, returning to coarser grids afterward to clean-up low frequency components of the error. This is called *grid sequencing*, and in PETSC it applies to nonlinear (and linear) problems at the level of the SNES solver, i.e., it is `-snes_grid_sequence`; see the next chapter.

On the other hand, in PETSC a multigrid cycle of the type considered here is a preconditioner for the linear equations on the fine grid. Thus, for `-pc_type mg` one must first restrict the fine-grid residual down to the coarsest grid before applying a “coarse-first” strategy. Two GMG types apply this kind of cycle, namely `-pc_mg_type full` and `-pc_mg_type kaskade` (Figure 6.14). These types work upward through finer grids in different ways. As it reaches each next-finer grid, `full` does a single V-cycle to resolve low-frequency components of the error. (One may add `-pc_mg_cycle_type w` to get a W-cycle.) The simpler upward-only version is `-pc_mg_type kaskade` [41]. Note PETSC only uses the down-smoother ( $S_1^{\nu_1}$ ) in `kaskade` (Exercise 6.21).

## Controlling multigrid

Consider the following example with `-pc_type mg` and three levels of refinement starting from the default  $3 \times 3$  grid in 2D (Figure 6.12):

```
$ ./fish -da_refine 3 -pc_type mg -ksp_converged_reason
  Linear solve converged due to CONVERGED_RTOL iterations 4
problem manuexp on 17 x 17 point 2D grid:
  error |u-uexact|_inf = 3.257e-05, |u-uexact|_h = 1.209e-05
```

This run solves the default Poisson problem using CG iteration and a GMG preconditioner with these defaults:

- $l = 4$  levels (`-pc_mg_levels 4`), one more than `-da_refine`,
- multiplicative composition (`-pc_mg_type multiplicative`) of V-cycles (`-pc_mg_cycle_type v`), applied once per PC application (`-pc_mg_multiplicative_cycles 1`),
- a nested-dissection-ordered (`-mg_coarse_pc_factor_mat_ordering_type nd`) LU direct solve (`-mg_coarse_ksp_type preonly -mg_coarse_pc_type lu`) on the coarsest grid, and
- a Chebyshev smoother (`-mg_levels_ksp_type chebyshev`), applied twice (`-mg_levels_ksp_max_it 2`; i.e.,  $\nu_1 = \nu_2 = 2$ ), using SSOR preconditioning (`-mg_levels_pc_type sor`).

This list, while impressive already, is quite incomplete!

Exposure of solver structure is again obligatory for understanding. In fact, the rest of this section depends on the reader having viewed the solver:

```
| $ ./fish -da_refine 3 -pc_type mg -ksp_view
```

(Option `-snes_view` gives nearly the same view because the SNES type is `ksponly`.)

One may explore multigrid-relevant options by adding `-help` to a `-pc_type mg` run and piping the result through `grep`, seeking one of the following prefixes: `pc_mg_`, `mg_levels_` or `mg_coarse_`. Control at a particular grid level  $z$  uses option prefix `-mg_levels_z_` (Figure 6.12), but to control all levels (except the coarsest) use `-mg_levels_`, and to control the coarsest use `-mg_coarse_`. For example, a clear view of multigrid V-cycles is revealed here by indentation:

```
$ ./fish -da_refine 3 -pc_type mg \
  -ksp_monitor -mg_{levels,coarse}_ksp_converged_reason
...
  [four V-cycles snipped]
  3 KSP Residual norm 1.000366698011e-03
  Linear mg_levels_3_ solve converged due to CONVERGED_ITS iterations 2
    Linear mg_levels_2_ solve converged due to CONVERGED_ITS iterations 2
      Linear mg_levels_1_ solve converged due to CONVERGED_ITS iterations 2
        Linear mg_coarse_ solve converged due to CONVERGED_ITS iterations 1
        Linear mg_levels_1_ solve converged due to CONVERGED_ITS iterations 2
      Linear mg_levels_2_ solve converged due to CONVERGED_ITS iterations 2
    Linear mg_levels_3_ solve converged due to CONVERGED_ITS iterations 2
  4 KSP Residual norm 3.635434047456e-05
problem manuexp on 17 x 17 point 2D grid:
  error |u-uexact|_inf = 3.257e-05, |u-uexact|_h = 1.209e-05
```

This reveals one V-cycle per preconditioner application, two smoother iterations per level, and one iteration for the direct solve on the coarse grid. Recall that the number of smoother applications is fixed, and no norm-based test for convergence is applied, so we see `CONVERGED_ITS` for the smoothers but `CONVERGED_RTOL` for the top-level KSP solver. Also, because we are using left

preconditioning, a V-cycle is applied before the first residual norm is reported. (Right preconditioning is not supported for the CG method, but compare `-ksp_type gmres -ksp_pc_side right`.)

One may also use X graphics to display the solution, i.e., the coarse-grid correction, at each level in the multigrid hierarchy (Exercise 6.23).

**Example 6.7.** As an exercise in understanding PETSc multigrid options, suppose we want to apply the following multigrid solver on an  $81 \times 81$  fine grid:

classical multigrid W-cycles, rediscretization on each grid, LU decomposition on the coarse grid,  $\nu_1 = 1$  and  $\nu_2 = 1$  classical GS smoothing steps, and three coarsenings.

To build the grids, note that an  $81 \times 81$  grid is 3 levels of refinement of an  $11 \times 11$  grid, so we start with `-da_grid_x 11 -da_grid_y 11 -da_refine 3`, and then the default `-pc_mg_levels 4` is what we want. We choose `-ksp_type richardson` for classical multigrid (with `-pc_type mg`). Rediscretization is the default (versus `-pc_mg_galerkin`) but we want `-pc_mg_cycle_type w`. The coarse grid solver defaults, namely `-mg_coarse_ksp_type preonly` and `-mg_coarse_pc_type lu`, are what we want. The default smoother is Chebyshev, so we switch to `-mg_levels_ksp_type richardson` for the classical GS iteration, but the default for `-mg_levels_pc_type sor` is SSOR, so we add `-mg_levels_pc_sor_forward` for GS. We set `-mg_levels_ksp_max_it 1` to limit the down- and up-smoothers to one iteration each. Here is the resulting command line:

```
$ ./fish -da_grid_x 11 -da_grid_y 11 -da_refine 3 -ksp_type richardson \
    -pc_type mg -pc_mg_cycle_type w -mg_levels_ksp_type richardson \
    -mg_levels_pc_sor_forward -mg_levels_ksp_max_it 1
```

To further clarify, add options `-ksp_view`, `-ksp_monitor`, or `-mg_{levels,coarse}_ksp_converged_reason`.

This solver turns out to be a bit faster than the default `mg` solver for this problem. For example, on a  $1281 \times 1281$  fine grid (`-da_refine 7`) it is about twice as fast. It is an excellent solver, but the defaults are good too.

Three further examples explore various `-pc_type mg` solvers, and they expose additional run-time options.

**Example 6.8.** For the 3D Poisson equation we compare the initial residual norm reduction from various multigrid types and cycles. To make it a relatively fair comparison we fix the number of smoother sweeps on the finest grid in each case. We use a  $129 \times 129 \times 129$  grid of  $N \approx 2 \times 10^6$  points, and each (serial) run takes a few seconds.

Specifically we run

```
$ ./fish -fsh_dim 3 -da_refine 6 -fsh_initial_type INIT \
    -ksp_type richardson -ksp_max_it 1 -ksp_monitor \
    -ksp_norm_type unpreconditioned -pc_type mg MORE
```

with initial conditions `INIT` equal to either `zeros` or `random`. This is a single multigrid-preconditioned Richardson iteration; there is no Krylov acceleration.

We compare six solvers by setting `MORE` as follows. First we try four V- or W-cycle solvers, each with one or two smoother sweeps at each level and one or two cycles per preconditioner application:

```
-pc_mg_cycle_type v -mg_levels_ksp_max_it 2 -pc_mg_multiplicative_cycles 1
-pc_mg_cycle_type v -mg_levels_ksp_max_it 1 -pc_mg_multiplicative_cycles 2
-pc_mg_cycle_type w -mg_levels_ksp_max_it 2 -pc_mg_multiplicative_cycles 1
-pc_mg_cycle_type w -mg_levels_ksp_max_it 1 -pc_mg_multiplicative_cycles 2
```

**Table 6.3.** Initial residual-norm ratios  $\|\mathbf{r}_1\|_2/\|\mathbf{r}_0\|_2$  for various multigrid types and cycles in Example 6.8.

	Multiplicative				Full	Kaskade
	V	2 V	W	2 W		
zeros	0.034	0.033	0.025	0.025	0.046	0.118
random	0.052	0.052	0.008	0.008	0.052	0.069

We also compare two coarse-grid-first solvers, with MORE equal to

```
-pc_mg_type full -mg_levels_ksp_max_it 2
-pc_mg_type kaskade -mg_levels_ksp_max_it 4
```

Every one of the 12 combinations corresponds to exactly four smoother sweeps on the finest grid, though with differing amounts of work on coarser grids. For each one we compute the ratio  $\|\mathbf{r}_1\|_2/\|\mathbf{r}_0\|_2$  of unpreconditioned (`-ksp_norm_type unpreconditioned`) residual norms for a single simple iteration. The results in Table 6.3 show that, for this Poisson problem, W-cycles are most effective and Kaskade iterations least so.

The zero initial condition yields an initial error (and residual) which is smooth. The random initial condition, i.e., white noise, gives nonsmooth initial error. The W-cycles, which spend a larger fraction of effort on coarser grids, are much more effective in the noisy case.

The above experiment used a fixed smoother, the default Chebyshev iteration. The next example measures the effect of different smoothers on the multigrid convergence rate. Regarding theory for this example, note that the smoothing factor of a relaxation scheme can be determined from the coefficients of the operator, i.e., the leading-order symbol [21], at least for classical iterations and structured-grid discretizations of linear, constant-coefficient elliptic equations. That is, see Exercises 6.26 and 6.27.

**Example 6.9.** We solve 2D cases of equation (6.18), namely

$$-c_x u_{xx} - c_y u_{yy} = f(x, y), \quad (6.31)$$

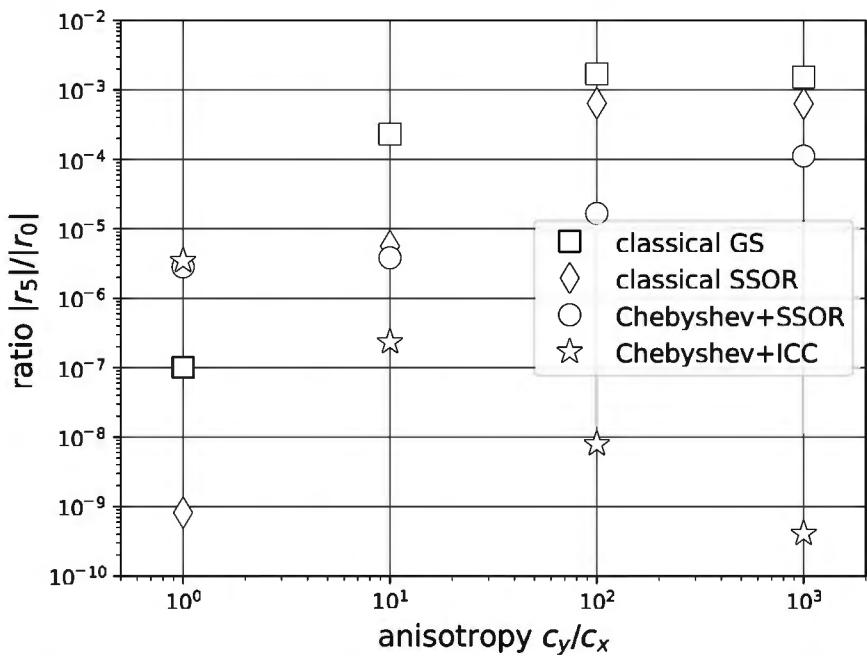
on domain  $\Omega = (0, 1)^2$ , for  $c_x = 1$  and different values  $c_y \geq 1$ . The ratio  $c_y/c_x$  is a measure of the *anisotropy* of the diffusion process, that is, of how the diffusion varies in different directions. When  $c_y/c_x$  is large then we expect that the error will retain significant oscillations in the relatively undamped direction, namely the  $x$  direction if  $c_y > 1$  (Exercise 6.25). As the residual is reduced the coarse-grid correction becomes inefficient because high-frequency error modes remain after smoothing and the multigrid iteration stagnates.

We use four values of  $c_y/c_x$  and four different smoothers (below), so there are 16 cases. In each case we do five classical V-cycles on a fixed  $257 \times 257$  grid and compute the reductions  $\|\mathbf{r}_5\|/\|\mathbf{r}_0\|$ :

```
$ ./fish -fsh_problem manopoly -fsh_initial_type random -da_refine 6 \
    -ksp_type richardson -ksp_norm_type unpreconditioned \
    -ksp_max_it 5 -ksp_rtol 0 -ksp_atol 0 -fsh_cy CY -pc_type mg SMOOTH
```

Here CY = 1, 10, 10<sup>2</sup>, 10<sup>3</sup> and SMOOTH is from the following list:

- `-mg_levels_ksp_type chebyshev -mg_levels_pc_type sor` (the default),
- `-mg_levels_ksp_type chebyshev -mg_levels_pc_type icc`,



**Figure 6.15.** When anisotropy  $c_y/c_x$  is strong in equation (6.31), multigrid performance degrades, though Chebyshev smoothing can adapt. Our best smoother for strong anisotropy uses ICC preconditioning.

- classical SSOR: `-mg_levels_ksp_type richardson -mg_levels_pc_type sor`,
- classical GS: same plus `-mg_levels_pc_sor_forward`.

Note we turn off KSP convergence tests so that the number of iterations is fixed.

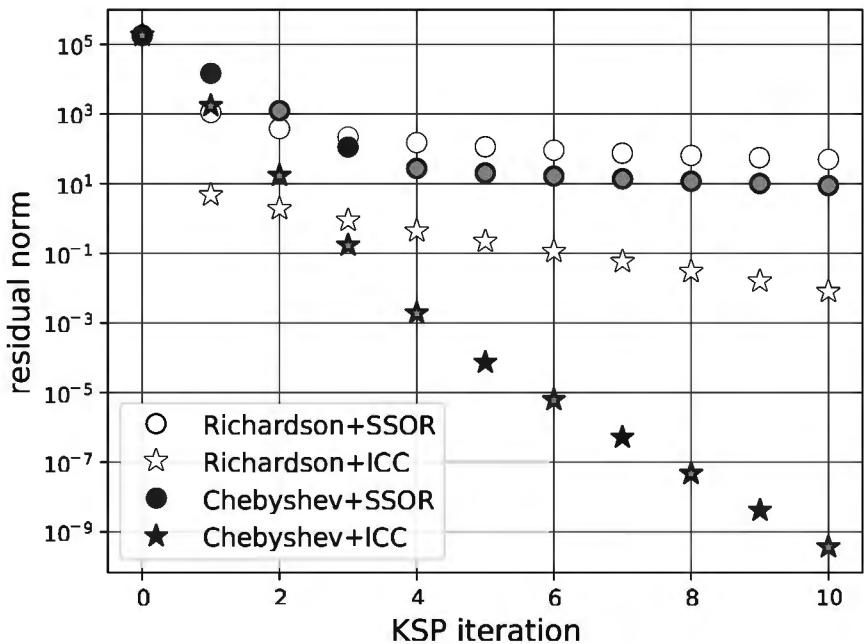
The result is in Figure 6.15. On the isotropic Poisson equation ( $c_y = 1$ ), classical SSOR and GS are the best smoothers in multigrid V-cycles. (SSOR does two GS sweeps per iteration so their efficiencies are comparable.) However, as the anisotropy grows the Chebyshev smoother can adapt to the growing range of eigenvalues of the matrix. Thus it becomes the best smoother for large anisotropy.

Switching to the goal of solving the equations fully, using option `-ksp_monitor` and showing the first ten iterations, in the strong anisotropic case our best performance is from Chebyshev iteration with an ICC-preconditioned matrix (Figure 6.16). An incomplete factorization can smooth the error effectively because it nearly solves the local interaction using the anisotropic coefficients.

Our last example below returns to the two-level Dryja and Widlund scheme (6.25). We show that with the addition of a coarse grid a domain decomposition method can have a bounded number of Krylov iterations as the grid is refined. This example also hints at the many subgrid-based preconditioners that can be composed at the PETSC command line.

**Example 6.10.** In equation (6.25) the coarse grid has spacing  $2h$ , the original grid has spacing  $h$ , and there are  $P$  subdomains. Suppose we solve the subdomain problems exactly by LU. The multigrid interpretation of this choice is to have two levels, i.e., a fine grid and a coarse grid, but then apply the additive Schwarz method on the fine grid *as a smoother*.

This is a parallel example because, by default, the number of ASM subdomains equals the number of processes. Suppose we solve the coarse grid problem exactly, but *redundantly*, by LU, so that each process solves the same coarse grid problem. (Note `-mg_coarse_pc_type redundant` is the parallel default; see Chapters 7 and 8.)



**Figure 6.16.** In the strong-anisotropy case with  $c_y = 10^3$ , multigrid V-cycles are fastest with a Chebyshev+ICC smoother.

We do the following runs:

```
$ mpieexec -n P ./fish -fsh_dim 2 -da_refine L -ksp_type gmres \
-ksp_rtol 1.0e-10 -pc_type mg -pc_mg_levels 2 -pc_mg_type additive \
-mg_levels_ksp_type preonly -mg_levels_pc_type asm \
-mg_levels_sub_pc_type lu -mg_coarse_ksp_type preonly \
-mg_coarse_pc_type redundant -mg_coarse_redundant_pc_type lu
```

We consider  $P = 4, 16, 64, 256$  processes and  $L = 6, 7, 8, 9$  refinement, respectively, so each subgrid has fixed size  $65 \times 65$  with  $N \approx 4 \times 10^3$  points. For comparison we also do single-level DD runs, namely `-pc_type asm -sub_pc_type lu`. Figure 6.17 shows the result. The two-level method does a fixed number of KSP iterations while the iterations for ASM+LU grow as  $O((1/H)^{1.37})$ .

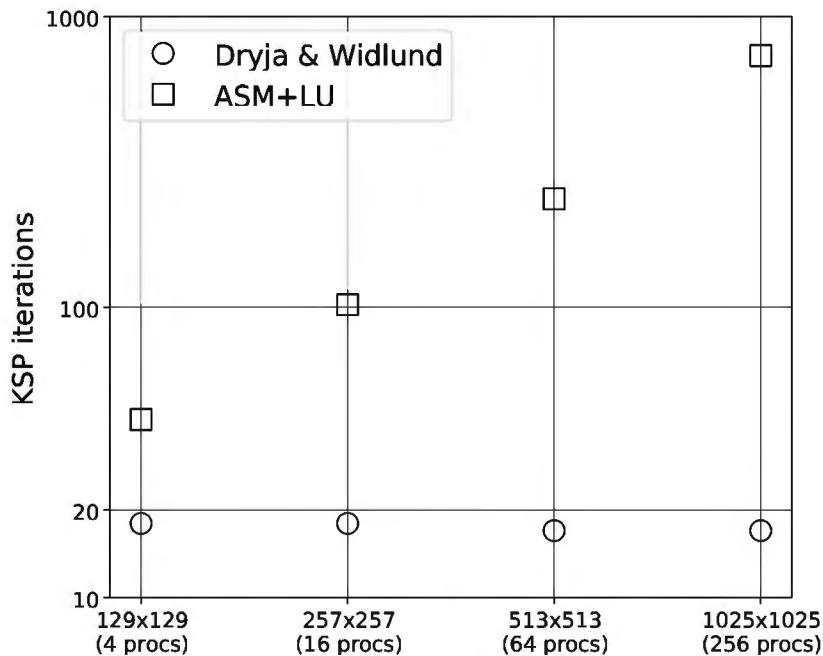
While the Dryja and Widlund two-level method (6.25) has grid-independent iterations, in the form here it is not recommended because at high resolution it requires large direct solves either on subgrids or on the coarse grid. A multigrid solver, by contrast, needs only low-cost smoothing on all levels, except possibly on a coarsest grid of (fixed) small size.

## Exercises

- 6.1. A scalar *weight*  $\omega$  may be used to modify the Jacobi iteration (6.5). Given splitting (6.4) this is written [64]

$$\mathbf{u}_{k+1} = (1 - \omega)\mathbf{u}_k + \omega D^{-1}(\mathbf{b} - (L + U)\mathbf{u}_k). \quad (6.32)$$

Show this is just simple iteration (6.3) with  $\alpha = \omega$  and  $M = D$ .



**Figure 6.17.** The two-level method (6.25), which adds a coarse grid to single-level DD, has constant KSP iteration counts under grid refinement.

- 6.2. Show that if  $\lim \mathbf{u}_k = \mathbf{u}$  exists in the Jacobi method (6.5) then  $A\mathbf{u} = \mathbf{b}$ . Show that if  $\rho(I - D^{-1}A) < 1$  then the method converges for all initial values  $\mathbf{u}_0$ . Show that if  $\rho(I - D^{-1}A) > 1$  then it diverges for some  $\mathbf{u}_0$ .
- 6.3. Decompose the matrix as for the Gauss-Seidel iteration (6.8), namely  $A = D + L + U$ , and suppose that  $D$  is invertible. Show that the linear system  $(D + L)\mathbf{y} = \mathbf{c}$  can be solved by the forward substitution formula

$$y[i] = \frac{1}{a_{ii}} (c[i] - a_{i,0}y[0] - \cdots - a_{i,i-1}y[i-1]).$$

Thereby explain the equivalence of (6.8) and (6.7). (Note that identifying the submatrices  $L$  and  $U$  requires ordering the variables.)

- 6.4. For a demonstration of the oddity identified on page 133, use `tri.c` from Chapter 2:

```
| $ ./tri -ksp_type richardson -pc_type sor -ksp_converged_reason
```

Then add `-ksp_monitor` and rerun. (Confirm a 1000-times difference in iterations!) Note that one sets a maximum number of KSP iterations by `-ksp_max_it`.

- 6.5. Show that the right-hand actions in Figure 6.7 are equivalent to applying  $M^{-1}$  from (6.16) in simple iteration (6.3) with  $\alpha = 1$ .
- 6.6. The difference between additive and multiplicative compositions can be seen in how they propagate error. Let  $\mathbf{e}_k = \mathbf{u}_k - \mathbf{u}$  where  $A\mathbf{u} = \mathbf{b}$ . Show that (6.15) implies

$$\mathbf{e}_{k+1} = (I - (B_0 + B_1)A)\mathbf{e}_k$$

while (6.16) implies

$$\mathbf{e}_{k+1} = (I - B_1 A)(I - B_0 A)\mathbf{e}_k.$$

- 6.7. Suppose  $R_i$  is a subgrid injection.
- Show that  $S = R_i^\top R_i$  is an orthogonal projection [143] in the usual inner product  $\langle \mathbf{u}, \mathbf{v} \rangle = \mathbf{u}^\top \mathbf{v}$ . That is, show  $S^2 = S$  and  $S^\top = S$ .
  - Show that if  $A$  is symmetric positive-definite, and if a subgrid correction matrix  $B_i$  is defined by (6.13), then  $S = B_i A$  is an orthogonal projection in the inner product  $\langle \mathbf{u}, \mathbf{v} \rangle_A = \mathbf{u}^\top A \mathbf{v}$ .
- 6.8. Consider the injection  $R$  from a fine grid to a coarse grid (Figure 6.10). Let  $S = R^\top R$  so that  $S$  is an orthogonal projection in the inner product  $\mathbf{u}^\top \mathbf{v}$ , with norm  $\|S\|_2 = 1$  (previous exercise). Argue that if  $A$  is the discretization of an elliptic operator then  $S$  nonetheless has large norm in the inner product  $\mathbf{u}^\top A \mathbf{v}$ , i.e., in energy norm.
- 6.9. Assume  $A$  is symmetric and suppose that the multiplicative composition in Figure 6.7 is followed by another residual update, then application of  $B_0$ , and then a final solution update, giving a three-step multiplicative composition [134]. Show that if the  $B_i$  are symmetric then this results in a preconditioner with symmetric matrix

$$M^{-1} = B_0 + B_1 - B_1 A B_0 - B_0 A B_1 + B_0 A B_1 A B_0.$$

- 6.10. Confirm that `ch6/fish.c` can do what `ch3/poisson.c` can do. For example, reproduce the results in Figure 3.10 and Table 3.1.
- 6.11. Code `fish.c` generates an SPD matrix  $A$ . Check this using option `-mat_is_symmetric`. Calculate its eigenvalues using options `-pc_type none -ksp_view_eigenvalues`.
- 6.12. In the text we demonstrate a factor-of-two difference in iteration count between classical Jacobi and GS methods using a tridiagonal matrix. Confirm that the same factor can be seen in iterations for this 3D Poisson problem:

```
| $ ./fish -fsh_dim 3 -da_refine 4 -ksp_converged_reason \
|   -ksp_type richardson
```

(Recall that classical iterations correspond to adding `-pc_type jacobi` or `-pc_type sor -pc_sor_forward`. You may want to do Exercise 6.4 first to avoid confusion.)

- 6.13. The ASM preconditioner does not require a DMDA. The decomposition can be done using `Vec` indices. Experiment with this Chapter 2 example:

```
| $ mpieexec -n P ./tri -tri_m M -ksp_type richardson -ksp_rtol 1.0e-12 \
|   -pc_type asm -sub_pc_type lu -pc_asm_overlap X
```

with various values of concurrency `P`, dimension `M`, and overlap `X`. Count KSP iterations and confirm the results on overlap in the text. (*However, in 2D and 3D PDE examples, because of the variable ordering, one wants the ASM to use the parallel decomposition generated by the DMDA.*)

- 6.14. The parallel overlapping ASM method requires communication between processes. In its classical form [134] the algorithm communicates overlap values of the residual before applying the preconditioner and then it communicates and interpolates the result after the application. (These communication stages correspond to column-wise and row-wise overlap in heuristic (6.17), respectively.) In PETSc this classical method is `-pc_asm_type basic`, but it is not the default. The default type `restrict` avoids the second communication stage by ignoring the overlap values. There is also type `interpolate` which avoids the first communication stage. Use `fish.c` to experiment and see why `restrict` is the default.

- 6.15. The ASM preconditioner is formula (6.19). The multiplicative Schwarz method (MSM) preconditioner, for  $p$  subgrids, is the following computation of  $\mathbf{v} = M^{-1}\mathbf{r}$  for a given vector  $\mathbf{r}$ :

$$\begin{aligned}\mathbf{v} &\leftarrow B_0\mathbf{r} \\ \mathbf{v} &\leftarrow \mathbf{v} + B_1(\mathbf{r} - A\mathbf{v}) \\ &\vdots \\ \mathbf{v} &\leftarrow \mathbf{v} + B_{p-1}(\mathbf{r} - A\mathbf{v}).\end{aligned}$$

PETSC supports this method in serial with options

```
-pc_type asm -pc_asm_blocks X -pc_asm_local_type multiplicative
```

(Classical MSM would also use `-pc_asm_type basic`; see Exercise 6.14.)

Using serial runs of `fish.c`, do your best to reproduce the first table of iteration counts on page 27 of [134]. Note that `fish.c` is set up to solve the same Poisson problem as in this table, but additional options `-ksp_type gmres -ksp_gmres_restart 10 -sub_pc_type lu -ksp_rtol 1.0e-2` will be needed. Comment on remaining differences.

- 6.16. Table 6.1 reports KSP iterations for ASM preconditioning runs using CG iteration with Cholesky direct solves on the subgrids. These are appropriate choices because  $A$  is symmetric (Exercise 6.11). Now compare results, including execution time, when you use GMRES+ASM+LU for the same runs. Recall that nested-dissection (ND) reordering is the default for PETSC's LU direct solver, so compare `-ksp_type cg -sub_pc_type cholesky -sub_pc_factor_mat_ordering_type nd` with `-ksp_type gmres -sub_pc_type lu`. Show that for GMRES+ASM+LU our conclusions about overlap are unaltered.
- 6.17. With `fish.c` we can check that formula (6.21), in which  $P_{\text{int}}^\top$  is used for restriction, is correct for the Galerkin coarse-grid matrix. Do

```
$ ./fish -fsh_dim 1 -da_refine 2 -pc_type mg -pc_mg_galerkin \
-mg_levels_1_ksp_view_mat ::ascii_dense
```

Compare the resulting  $5 \times 5$  matrix with the one shown in the text. Then remove option `-pc_mg_galerkin` and compare to (6.22).

- 6.18. Suppose  $A^h$  is the matrix on the fine grid  $\Omega^h$  and  $\mathbf{c}$  is a positive vector of length  $N^{2h}$ , the size of the coarse grid  $\Omega^{2h}$ . Suppose  $P$  denotes some prolongation from  $\Omega^{2h}$  to  $\Omega^h$  and define a restriction  $R = \text{diag}(\mathbf{c})P^\top$ . Show that  $P(RA^hP)^{-1}R = P(P^\top A^hP)^{-1}P^\top$ . Thus the scalings  $\mathbf{c}$  do not change the Galerkin coarse-grid correction  $B^{2h}$  in (6.23), though they do affect the Galerkin coarse-grid matrix  $\hat{A}^{2h}$  in (6.21).

- 6.19. Show (6.27). (Recall Exercise 2.5.)
- 6.20. Write out a pseudocode for the W-cycle, based on Figure 6.13 and the V-cycle pseudocode.
- 6.21. By default the PETSc GMG post-smoother (up-smoother)  $\mathcal{S}_2$  is the same KSP object as the pre-smoother (down-smoother)  $\mathcal{S}_1$ . These can be separated by declaring the up-smoother as distinct, but then one must control things carefully. Add the following options to a GMG run of `fish`:

```
-pc_mg_distinct_smoothup -mg_levels_up_ksp_max_it 3 \
-mg_{levels,levels_up,coarse}_ksp_converged_reason
```

(Note `-pc_mu_distinct_smoothup` generates new options with postfix `_up`.) This will show the GMG cycle structure rather clearly. Thereby show that the various solid dots and circles in Figures 6.13 and 6.14 are correct.

- 6.22. Show (6.29).
- 6.23. The internal states of a multigrid solver can be visualized from the PETSC command line. For example, to visualize the coarse grid solutions and the smoother iterates at all grid levels, add the following options to a GMG run of `fish`:

```
-mg_coarse_ksp_view_solution draw -mg_levels_ksp_view_solution draw \
-draw_pause 0
```

- 6.24. The default smoother in PETSC is Chebyshev iteration with SSOR preconditioning. Alternatives based on classical (weighted) Jacobi (6.32), GS (6.8), and SSOR (6.10) iterations are worth comparing because they are ubiquitous in the multigrid literature [26, 49, 144]. For 2D Poisson the recommended  $\omega$  value for a weighted-Jacobi smoother is  $\omega = 4/5$  [144].

Using GMG runs of `fish.c`, and various levels of refinement, compare numbers of iterations using the following smoothers:

```
-mg_levels_ksp_type chebyshev -mg_levels_pc_type sor      # default
-mg_levels_ksp_type richardson -mg_levels_pc_type jacobi
-mg_levels_ksp_type richardson -mg_levels_pc_type sor \
    -mg_levels_pc_sor_forward
-mg_levels_ksp_type richardson -mg_levels_pc_type sor
-mg_levels_ksp_type richardson -mg_levels_pc_type jacobi \
    -mg_levels_ksp_richardson_scale 0.8
```

- 6.25. In the runs which produced Figure 6.15 we only consider equation (6.31) with  $c_y \geq 1$ . What about anisotropy with  $c_y \leq 1$ ? It is actually equivalent to solving  $-c_x u_{xx} - u_{yy} = f(x, y)$  with  $c_x \geq 1$ , in the sense that in both cases the smoother becomes inefficient at removing oscillations from the error in the  $y$  direction. Visualize this effect.
- 6.26. In the text we visualize the smoothing properties of the classical iterations but avoid analysis. One may use Fourier analysis for 2D Gauss-Seidel iteration ([21, pp. 10–11] and [144, Chapter 4]) as follows.

Consider the 2D and  $c_x = 1$  case of (6.18) with square cells  $h = h_x = h_y$ . Apply our FD scheme, equation (3.5):

$$-(u_{i-1,j} - 2u_{ij} + u_{i+1,j}) - c_y(u_{i,j-1} - 2u_{ij} + u_{i,j+1}) = h^2 f(x_i, y_j).$$

Suppose we apply the GS iteration by traversing the grid in lexicographic order. If  $u_{ij}^{(k)}$  is the  $k$ th GS iterate, the errors  $v_{ij}^{(k)} = u_{ij}^{(k)} - u_{ij}$  satisfy

$$-(v_{i-1,j}^{(k+1)} - 2v_{ij}^{(k+1)} + v_{i+1,j}^{(k)}) - c_y(v_{i,j-1}^{(k+1)} - 2v_{ij}^{(k+1)} + v_{i,j+1}^{(k)}) = 0. \quad (6.33)$$

We expand the errors in a finite Fourier series using  $I = \sqrt{-1}$ :

$$v_{ij}^{(k)} = \sum_{(s,t)} V_{st}^{(k)} e^{I(sj+ti)}. \quad (6.34)$$

This sum is over frequencies  $(s, t)$  which can be supported on a grid with spacing  $h = 1/m$ , thus over certain points in the square  $[-\pi, \pi]^2$ . For example,  $s \in \{2\pi q/m\}$  where  $q$  is an integer with  $|q| \leq m/2$ , and similarly for  $t$ .

Combine equations (6.33) and (6.34) to show that the coefficients in the Fourier expansion satisfy

$$(-e^{-Is} - c_y e^{-It} + 2 + 2c_y) V_{st}^{(k+1)} + (-e^{Is} - c_y e^{It}) V_{st}^{(k)} = 0.$$

Thus the *amplification factors* of the frequencies satisfy

$$\mu(s, t) := \frac{|V_{st}^{(k+1)}|}{|V_{st}^{(k)}|} = \left| \frac{e^{Is} + c_y e^{It}}{2 + 2c_y - e^{-Is} - c_y e^{-It}} \right|.$$

Observe that  $\mu(s, t) \rightarrow 1$  as  $(s, t) \rightarrow (0, 0)$ .

- 6.27. (*Continues Exercise 6.26.*) Define a 2D frequency  $(s, t)$  to be *high* if either component is above half the maximum frequency:  $\max\{|s|, |t|\} \geq \pi/2$ . The *smoothing factor* is the worst amplification factor considering only high frequencies,

$$\bar{\mu} = \max_{\max\{|s|, |t|\} \geq \pi/2} \mu(s, t).$$

First, consider GS smoothing for the isotropic case of (6.18) with  $c_x = c_y = 1$ . Reference [21] claims a “simple calculation” shows that  $\bar{\mu} = 0.5$ , attained at  $(\pi/2, \arccos(4/5))$ . Using the result of Exercise 6.26, confirm this, perhaps numerically via a contour map of  $\mu(s, t)$ . A smoothing factor of 0.5 is satisfactory; a few smoothing sweeps generates errors  $v_{ij}^{(k)}$  which are well approximated on the next-coarsest grid.

Next, for anisotropic cases with  $c_x = 1$  and  $c_y \ll 1$  or  $c_y \gg 1$ , generate additional contour plots of  $\mu(s, t)$ . Confirm that  $\mu(0, \pi/2)$  and  $\mu(\pi/2, 0)$  are the maxima in these cases, respectively, and that  $\bar{\mu} \approx 1$ , which is not so good.

# Interlude: Quadrature

In the chapters ahead we will need to numerically approximate integrals, i.e., do *quadrature*, in planar regions. Here we pause to present a bit of code which computes integrals over *reference elements* for the upcoming finite element (FE) methods, namely a square for Chapter 9 and a triangle for Chapter 10. (Integrals over the square also make an appearance in Chapter 7.) While the description here is deliberately basic and limited, the Firedrake library (Chapter 13) automates quadrature and extends these methods far beyond what is covered here. As this interlude is really just calculus, readers can skip forward and return to it when needed.

Quadrature rules for one-dimensional integrals [67] are of the form

$$\int_{-1}^1 f(x) dx \approx \sum_{r=0}^{n-1} w_r f(\xi_r). \quad (\text{I.1})$$

Other intervals of integration are handled by a linear change of variable. Generally a nonnegative *weight function*  $\rho(x)$  is allowed; the left side of (I.1) becomes  $\int_{-1}^1 f(x) \rho(x) dx$ , but the right side is unchanged.

In *Gaussian* rules the *nodes*  $\xi_r$  and *weights*  $w_r$  are chosen to maximize the degree of polynomials  $f(x)$  for which the integral is exact, that is, to maximize the *degree* of the rule. The trapezoid rule, Simpson's rule, and other *Newton-Cotes* rules [125] are less accurate than Gaussian rules, for a given number of function evaluations, because the nodes are equally spaced.

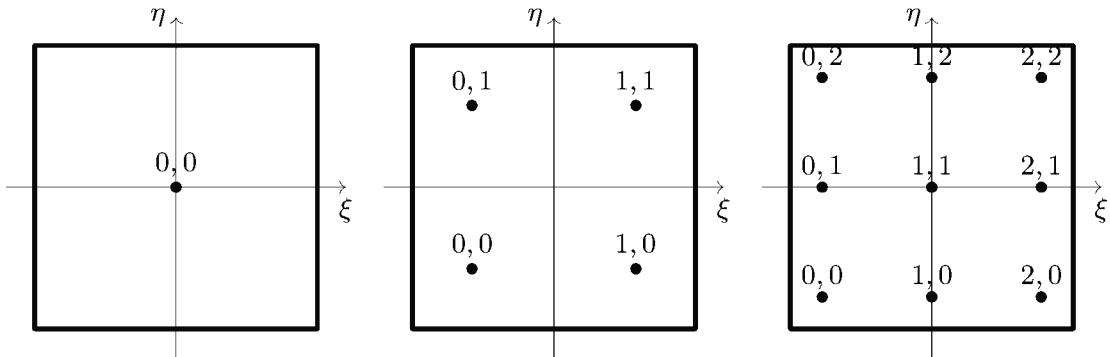
*Gauss-Legendre* quadrature is a Gaussian rule over interval  $[-1, 1]$  with  $\rho(x) = 1$ . The rules with  $n = 1, 2, 3$  points are given in Table I.1; the  $n = 1$  case is simply the midpoint rule. The elegant manner in which the nodes and weights are determined is left for the references (e.g., [67, 125]), but note that the degree of the  $n$ -point rule is  $2n - 1$  (Exercise I.1).

**Table I.1.** Nodes and weights for low-degree Gauss-Legendre quadrature.

$n$	nodes $\xi_r$	weights $w_r$
1	0	2
2	$-\frac{1}{\sqrt{3}}, +\frac{1}{\sqrt{3}}$	1, 1
3	$-\sqrt{\frac{3}{5}}, 0, +\sqrt{\frac{3}{5}}$	$\frac{5}{9}, \frac{8}{9}, \frac{5}{9}$

For now we merely record nodes and weights in a `struct`, defined in the C header `c/quadrature.h` and displayed in Code I.1. A program using the  $n$ -point rule would include this header file and then choose the rule:

```
#include "quadrature.h"
const Quad1D q = gausslegendre[n-1];
```



**Figure I.1.** Tensor product Gauss-Legendre rules (I.2) on the square  $\square_*$  for  $n = 1, 2, 3$ .

The calling program then uses values `q.n`, `q.xi[r]`, `q.w[r]`. For examples see `c/ch7/minimal.c` and `c/ch9/phelm.c`.

```
#define MAXPTS 3

typedef struct {
    PetscInt n;           // number of quadrature points for this rule
    PetscReal xi[MAXPTS], // locations in [-1,1]
              w[MAXPTS]; // weights (sum to 2)
} Quad1D;

static const Quad1D gausslegendre[3]
= { {1,
      {0.0,          NAN,          NAN},
      {2.0,          NAN,          NAN}} ,
{2,
  {-0.577350269189626, 0.577350269189626, NAN},
  {1.0,           1.0,          NAN}} ,
{3,
  {-0.774596669241483, 0.0,          0.774596669241483},
  {0.555555555555556, 0.888888888888889, 0.555555555555556}} };
```

**Code I.1.** `c/quadrature.h`, part I. Implementation of (I.1).

Formula (I.1) can be extended to integrals over the square  $\square_* = [-1, 1] \times [-1, 1]$ , the reference element for the  $Q^1$  FE method in Chapter 9:

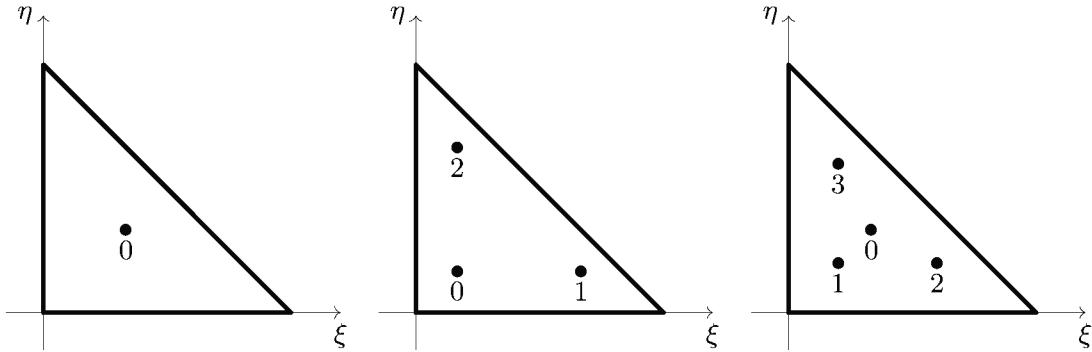
$$\int_{\square_*} v(\xi, \eta) d\xi d\eta \approx \sum_{r=0}^{n-1} \sum_{s=0}^{n-1} w_r w_s v(\xi_r, \xi_s). \quad (\text{I.2})$$

The  $n = 1, 2, 3$  cases of this 2D rule are pictured in Figure I.1; it is simply the product (*tensor product* [87]) of 1D rules.

For rectangles a change of variables transfers the integral to  $\square_*$  as follows. Suppose  $v(x, y)$  is any integrable function on  $R = [a, b] \times [c, d]$ . The linear map  $x(\xi, \eta) = a + \frac{1}{2}(b-a)(\xi + 1)$ ,  $y(\xi, \eta) = c + \frac{1}{2}(d-c)(\eta + 1)$  takes  $\square_*$  to  $R$ . The Jacobian of this map is the constant  $(b-a)(c-d)/4$ , the ratio of areas, so

$$\int_R v(x, y) dx dy = \frac{(b-a)(c-d)}{4} \int_{\square_*} v(\xi, \eta) d\xi d\eta, \quad (\text{I.3})$$

where  $v(\xi, \eta) = v(x(\xi, \eta), y(\xi, \eta))$ .



**Figure I.2.** Symmetric Gaussian quadrature rules on the reference triangle  $\Delta_*$  for  $n = 1, 3, 4$  points. Points  $(\xi_r, \eta_r)$  are indexed by  $r = 0, \dots, n - 1$ .

Integrals over triangles will be approximated using symmetric quadrature rules [46]. Here the triangle  $\Delta_*$  with vertices  $(0, 0), (1, 0), (0, 1)$  is the reference element (Figure I.2) for the  $P^1$  FE method in Chapter 10. For  $n$  quadrature points  $(\xi_r, \eta_r) \in \Delta_*$  with weights  $w_r$  we have

$$\int_{\Delta_*} f(\xi, \eta) d\xi d\eta \approx \sum_{r=0}^{n-1} w_r f(\xi_r, \eta_r). \quad (\text{I.4})$$

(Nonsymmetric rules based on tensor products of one-dimensional integrals are an alternative [87].) Table I.2 shows the degree  $k = 1, 2, 3$  rules with  $n = 1, 3, 4$  points, respectively. Note that the sum of the weights is  $1/2$  because  $|\Delta_*| = 1/2$  is the area of the triangle. The  $k = 3$  rule has a negative coefficient so its stability is suspect.

**Table I.2.** Symmetric Gaussian quadrature rules on the reference triangle  $\Delta_*$ .

Degree $k$	$n$	Nodes $(\xi_r, \eta_r)$	Weights $w_r$
1	1	(1/3, 1/3)	1/2
		(1/6, 1/6)	1/6
2	3	(2/3, 1/6)	1/6
		(1/6, 2/3)	1/6
3	4	(1/3, 1/3)	-27/96
		(1/5, 1/5)	25/96
		(3/5, 1/5)	25/96
		(1/5, 3/5)	25/96

We implement (I.4) similarly to the tensor product case, but now we have a single index  $r$  for the quadrature points so we store the locations as pairs (Code I.2). An application program would declare

```
#include "quadrature.h"
const Quad2DTri q = symmgauss[k-1];
```

then use values  $q.n, q.xi[r], q.eta[r], q.w[r]$ ; see code `c/ch10/unfem.c` for an example. Finally, as illustrated in Chapter 10, general triangles are easily handled using a linear change of variables.

```
#define MAXPTS_TRI 4

typedef struct {
    PetscInt n;           // number of quad. points for this rule
    PetscReal xi[MAXPTS_TRI], // locations: (xi,eta) in ref. triangle
              eta[MAXPTS_TRI], // with vertices (0,0), (1,0), (0,1)
              w[MAXPTS_TRI];   // weights (sum to 0.5)
} Quad2DTri;

static const Quad2DTri symmgauss[3]
= { {1,
      {1.0/3.0,    NAN,      NAN,      NAN},
      {1.0/3.0,    NAN,      NAN,      NAN},
      {1.0/2.0,    NAN,      NAN,      NAN} },
  {3,
      {1.0/6.0,    2.0/3.0,  1.0/6.0,  NAN},
      {1.0/6.0,    1.0/6.0,  2.0/3.0,  NAN},
      {1.0/6.0,    1.0/6.0,  1.0/6.0,  NAN} },
  {4,
      {1.0/3.0,    1.0/5.0,  3.0/5.0,  1.0/5.0},
      {1.0/3.0,    1.0/5.0,  1.0/5.0,  3.0/5.0},
      {-27.0/96.0, 25.0/96.0, 25.0/96.0, 25.0/96.0} };
```

**Code I.2.** *c/quadrature.h, part II. Implementation of (I.4).*

## Exercises

I.1 Write a short code using (I.2) to do the integrals

$$\int_{\square_*} (1 + \xi)^k + (1 + \eta)^k d\xi d\eta = \frac{2^{k+3}}{k+1}$$

for  $k = 0, 1, \dots, 6$ . Confirm that the  $n = 1, 2, 3$  Gauss-Legendre quadrature formulas exactly integrate these degree  $2n - 1$  polynomials, but not degree  $2n$  polynomials.

I.2 The claimed degree of accuracy  $k = 1, 2, 3$  in Table I.2 can be checked by comparing to the exact integral

$$\int_{\Delta_*} \xi^i \eta^j d\xi d\eta = \frac{i! j!}{(i+j+2)!}.$$

Confirm exactness for all cases with  $0 \leq i + j \leq n$ , and inexactness for some cases with  $i + j = n + 1$ .

## Chapter 7

# Optimal solvers for elliptic PDEs

**Definition.** A solution method for systems of equations in  $N$  real unknowns is said to be *optimal* if it solves these problems in  $O(N)$  floating-point operations (*flops*) as  $N \rightarrow \infty$  [109].

Because PDE problems are posed in infinite-dimensional spaces, but then discretized into  $N$ -variable algebraic systems in which  $N$  wants to be large, this definition should make sense as an aspiration. In this chapter, on several elliptic PDE problems, we achieve it by using the multigrid methods introduced in the last chapter. In fact, the rest of the book will demonstrate, or nearly demonstrate, an optimal solver for each PDE problem. Note that while this chapter focuses on flops, in the next chapter we take seriously actual run time, not to mention parallelization. Because algorithms generally change slightly when run in parallel (see Chapter 8), here we only consider the one-process case.

## Solver complexity

We call the asymptotic growth rate of flops for a solution method the *solver complexity*. An optimal solver has the lowest possible solver complexity because any method for solving nontrivial equations for  $N$  unknowns must do at least  $O(N)$  operations of some kind.

In the above definition we should really refer to a “family of” systems of equations with  $N \rightarrow \infty$ , and furthermore  $N \rightarrow \infty$  asymptotics only make sense on a hypothetical computer with infinite memory; in practice we will always be memory limited. Also, the phrase “solves these problems” means that a residual norm is reduced to a predetermined tolerance which is independent of  $N$ , so the definition depends on a choice of norm on  $\mathbb{R}^N$ .

Optimality is an impossible goal for families of linear systems with generic dense matrices having  $O(N^2)$  nonzero entries. Any such method would need to do some operations on each non-zero matrix entry. Polynomial or trigonometric spectral methods [142], for example, discretize a PDE into nontrivial dense matrix equations, and solver complexity often exceeds  $O(N^2)$  in that case. However, spectral methods are extraordinarily effective in many cases—Chapter 13 includes an example—and optimality as we have defined it is *not* the relevant issue if the order of the discretization error also grows with increasing  $N$ . Thus we will only use the adjective “optimal” to describe a solution method for the discrete equations for finite difference/element/volume methods if the discretization error order is  $O(h^k)$  where  $k$  is fixed.

An easy but important class of optimal methods for linear algebraic equations are the LU and Cholesky direct factorizations for tridiagonal or otherwise banded linear systems  $A\mathbf{u} = \mathbf{b}$ . For systems, with bandwidth bounded independently of  $N$ , i.e.,  $O(1)$  bandwidth as  $N \rightarrow \infty$ , these

direct methods are optimal as long as pivoting is not required for stability (see Chapter 2). Thus optimal solvers often exist for the discretizations of so-called “two-point” problems, i.e., ODE boundary value problems.

However, suppose we consider an FD discretization of the Poisson PDE on a  $d$ -dimensional structured grid with  $N$  unknowns and  $m = O(N^{1/d})$  grid points in each dimension (Chapter 6). This yields a matrix  $A$  with bandwidth which grows with  $N$ . In 2D, with the usual ordering of the variables, the bandwidth is  $O(m) = O(N^{1/2})$  and in 3D it is  $O(m^2) = O(N^{2/3})$ . If  $A$  is also symmetric positive definite (SPD) we can apply Cholesky decomposition and this (banded) algorithm has solver complexity  $O(N^2)$  in 2D and  $O(N^{7/3})$  in 3D [64]. While much better than generic Gauss elimination at  $O(N^3)$ , such solvers are not optimal. More sophisticated variable orderings—see discussion of nested-dissection ordering in Chapters 2 and 6—can substantially reduce the solver complexity, but not to  $O(N)$  except in special cases.

In any case, FD and FE discretizations of linear elliptic PDEs generate sparse matrices  $A$  with a bounded number of nonzeros per row, so these matrices have  $O(N)$  nonzero entries in total. Multiplying a vector  $\mathbf{v}$  by such a matrix, i.e., a single sparse “mat-vec” operation  $A\mathbf{v}$ , takes  $O(N)$  work. This is the starting point for seeking optimality in any “sparse method.”

A sketch of the good case where optimality *is* achieved goes like this. We suppose that the preconditioner (Chapter 2) requires  $O(N)$  work per application of  $M^{-1}$ , and we choose a Krylov method for which the work in each iteration, usually including a few sparse mat-vecs  $A\mathbf{v}$ , is  $O(N)$ . Then we suppose that the number of preconditioned Krylov iterations to achieve a small residual norm is independent of  $N$ , that is, it is  $O(1)$ . (Equivalently, the number of iterations is  $O(1)$  as  $h \rightarrow 0$  if the problem comes from discretizing a PDE.) Then this solver requires  $O(N)$  total work:

$$\text{optimal: } O(1) \begin{pmatrix} \text{number of iterations} \\ & \begin{matrix} \text{preconditioner application} & \text{Krylov step} \end{matrix} \\ O(1) & O(N) + O(N) \end{pmatrix} = O(N).$$

Many preconditioned-Krylov methods do not achieve optimality, under the above or any other strategy, but at least the assumption of  $O(N)$  work for each Krylov step includes most KSP types, including Richardson iteration, CG, MINRES, and bi-orthogonalization methods [66]. (See Chapter 11 for an example using `-ksp_type bcgs`.) It even includes restarted GMRES if the restart count is fixed, but not the true GMRES iteration which has growing work and memory for each iteration.

Critically, however, Krylov iteration counts are tied to the spectral properties of the preconditioned matrix  $M^{-1}A$  (Chapter 2), and this is the single most important barrier to optimality. The preconditioned matrices must have spectra which are appropriately bounded—e.g., clustered into disks away from the origin of the complex plane—*independently* of  $N$ . For discretized PDEs this means that the spectral bounds on the preconditioned operators must be independent of the mesh spacing  $h$ .

Our definition of optimality applies equally to nonlinear algebraic systems solved by Newton-Krylov methods. Such a solver is optimal if a fixed number of Newton iterations, independent of  $N$ , gives the desired accuracy, *and* if each linear Newton step problem is solved optimally. (As a detail, if the Newton iteration is implemented with a line search then the number of residual evaluations per line search must also be bounded independently of  $N$ .) For example, the well-behaved nonlinear ODE boundary value problem in `reaction.c` in Chapter 4 was solved optimally by a fixed number of Newton steps using a direct tridiagonal linear solver.

For a large class of discretized PDE problems in 2D and 3D, multigrid preconditioning is the key to constructing optimal solvers. In fact, a slightly stronger goal than optimal complexity is sometimes sought for multigrid solvers. “Textbook multigrid efficiency” [4, 28, 140] means that

the solution method does at most the work of 10 residual evaluations, but in this book we usually overlook the constant in “ $O(N)$ .” Optimality, defined above, is our goal.

It is worth observing that standard explicit time-stepping schemes for advection equations (Chapter 11), and generally the FD and FV methods used for hyperbolic time-dependent PDEs, are also optimal in the following sense. In these cases the  $N$  unknowns of the problem are understood to be all the points of a space-time grid. Because of CFL-type restrictions associated to well-defined maximum wave speeds (Chapter 11), the grid refines at comparable rates in all dimensions, but also  $O(1)$  operations suffice to compute the value at each space-time grid point. While we do not want to press this perspective on the reader, it is one way of explaining the dominance of explicit methods for hyperbolic PDEs.

This discussion is raising the bar quite high. Not every PDE has a known optimal solver. However, on a variety of linear and nonlinear elliptic PDEs we will show clear evidence of optimality, namely  $O(N)$  work for large  $N$ , by using multigrid-preconditioned Newton-Krylov methods. In this chapter we demonstrate it for the 2D and 3D Poisson equation, the nonlinear minimal surface equation, and the fourth-order biharmonic (plate) equation.

## An optimal solver for the Poisson equation

We start by demonstrating the serial (single-process) optimality of geometric multigrid (GMG) for the Poisson equation in 2D and 3D, using `fish.c` from the last chapter. Our primary measure of solver complexity will be flops, as a function of the number of degrees of freedom, but we demonstrate optimality in run time as well.

Consider these 2D runs using grid-refinement levels  $L = 3, \dots, 9$ , the CG method, and  $\text{PC} = \text{none}, \text{icc}, \text{mg}$ :

```
$ cd c/ch6/
$ ./fish -ksp_monitor_singular_value -da_refine L -pc_type PC
```

Option `-ksp_monitor_singular_value` prints the ratio of maximum and minimum singular values for the preconditioned matrices, i.e., the 2-norm condition number  $\kappa = \kappa(M^{-1}A)$ . In the above runs, using the last  $\kappa$  value printed once the KSP has converged, we get the results in Figure 7.1.

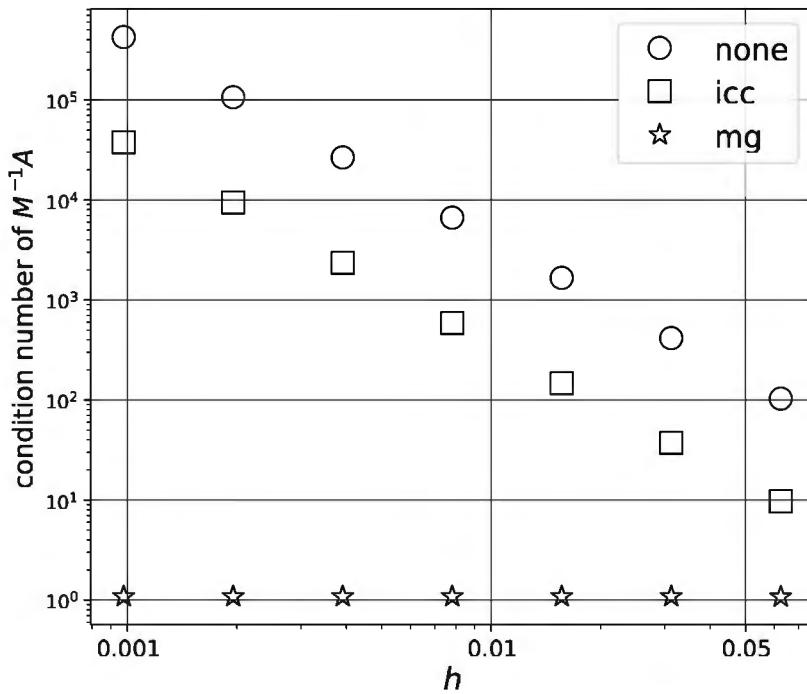
Now, Theorem 3.4 shows that if  $\kappa$  is bounded independently of  $h$  then the number of CG iterations will be also, and the solution method will be optimal. For the GMG-preconditioned matrix,  $\kappa$  is impressively constant, bounded by 1.2 as  $h$  decreases. This reflects that a multigrid V-cycle nearly solves the Poisson problem, unassisted by the CG iteration. By contrast, for the unpreconditioned matrices  $\kappa$  grows from  $10^2$  to  $4 \times 10^5$ . For ICC preconditioning  $\kappa$  grows at the same rate, though it is an order of magnitude smaller.

Now consider these runs using default GMG settings:

```
$ ./fish -fsh_dim D -ksp_rtol 1.0e-10 -pc_type mg -da_refine L \
-ksp_converged_reason -log_view
```

For dimension  $D = 2$  we use levels  $L = 5, \dots, 11$ , while for  $D = 3$  we use  $L = 3, \dots, 7$ , so that in each case  $N$ , the total number degrees of freedom, spans more than three orders of magnitude. The maximum refinement levels give  $N > 10^7$ , on grids of  $4097^2$  and  $257^3$  points, respectively. These finest-grid runs took less than a minute, but refining one more level generated “out-of-memory” error messages on the author’s workstation; compare with Exercise 7.1. (In fact, with regard to achieving high resolutions, a *feature* of optimal solution methods is that one can view them as essentially memory limited, not run-time limited.)

To measure algorithmic work we can use either the flops or the wall clock time for the SNESolve event reported by `-log_view`. Note that inside PETSc solver kernels, and in the



**Figure 7.1.** Condition numbers  $\kappa(M^{-1}A)$  of preconditioned Poisson matrices. For GMG, it is independent of  $h$ , and close to one.

residual/Jacobian evaluation routines in `poissonfunctions.c`, flops are counted through calls to `PetscLogFlops()`. On the other hand, timings are noisier because they are subject to delays from memory-hierarchy transfers (“cache misses”), and from competition with other jobs.

When we plot flops versus  $N$  for the above runs, optimality is very clear. Figure 7.2 shows that  $\text{flops} = O(N^1)$  to near perfection, reflecting the simplicity of the Poisson problem on a square/cube and the power of a single GMG V-cycle to nearly solve the problem. While the 2D and 3D results look rather the same, by plotting flops-per- $N$  versus  $N$  as in Figure 7.3 we see that the 3D computations are consistently more expensive. Each KSP iteration did a few hundred flops per unknown in all cases. On the other hand, plotting time-per- $N$  as in Figure 7.4 reveals that this workstation needs about 3 microseconds per unknown for larger problem sizes.

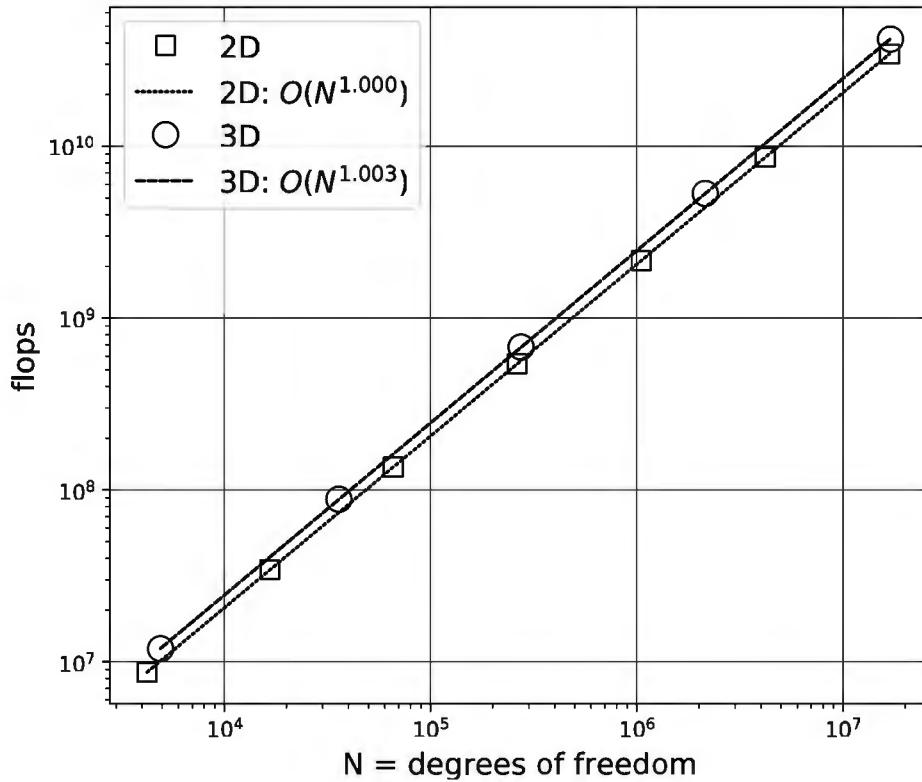
There are too many possibilities to allow testing all GMG option combinations (Chapter 6), but on the 3D Poisson problem we can also compare

- W-cycles (`-pc_mg_cycle_type w`),
- full multigrid cycles (`-pc_mg_type full`), and
- Galerkin coarse-grid operator construction (`-pc_mg_galerkin`).

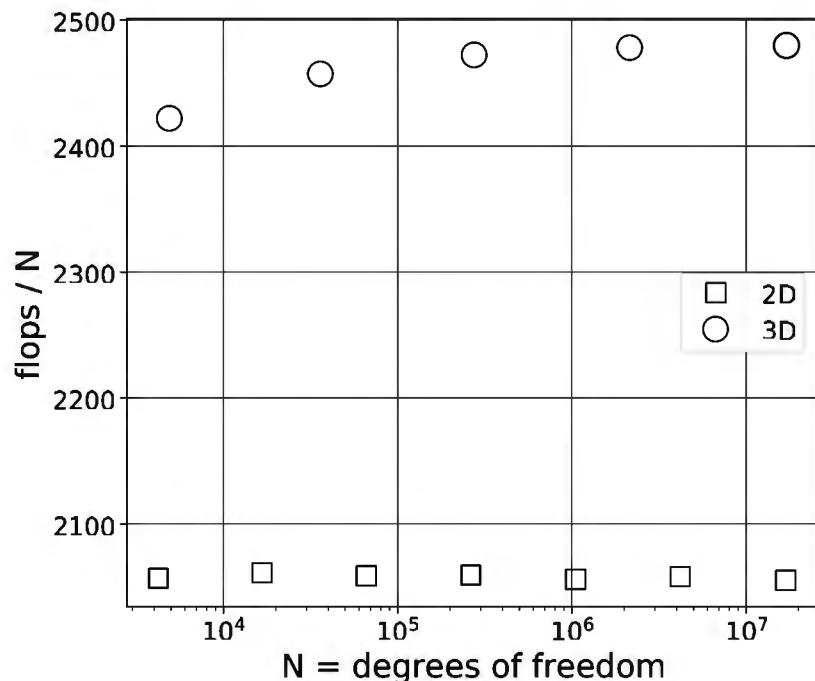
In every case, the result is between 2400 and 3700 flops per degree of freedom across all of the above grids (not shown). There is some variation between the methods, primarily because the number of KSP iterations varies a little, but the evidence is clear that, on this clichéd Poisson problem, all of these GMG solvers are optimal.

## Parallel multigrid and the coarse-grid problem

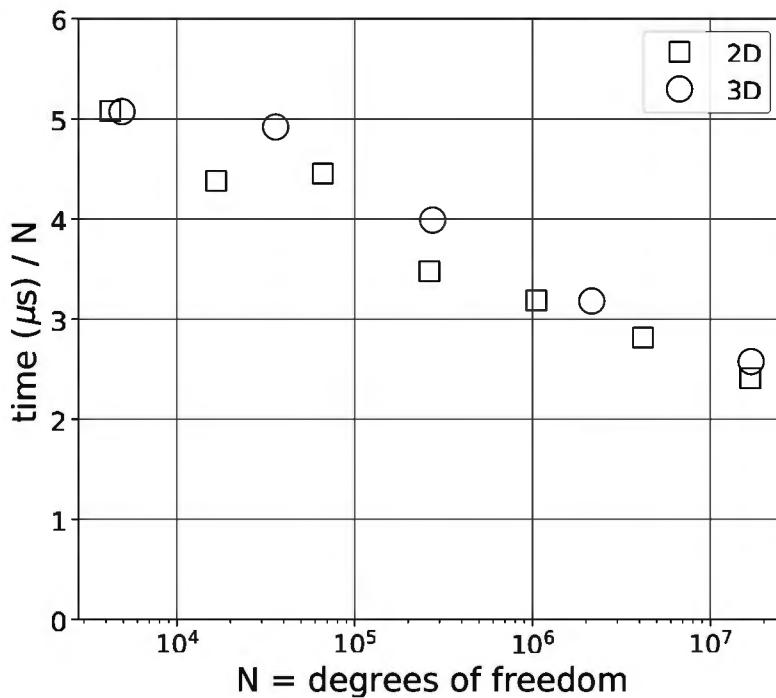
Though our demonstrations have been serial so far, multigrid is parallelized in PETSc. This assertion has at least two meanings:



**Figure 7.2.** Measured by total flops versus  $N$ , GMG-preconditioned CG is clearly an optimal solver for 2D and 3D structured-grid Poisson problems.



**Figure 7.3.** Plotting flops-per- $N$  gives a flat graph for an optimal solver. 3D solutions require more work per degree of freedom.



**Figure 7.4.** Wall clock time per degree of freedom gives a relatively flat graph.

- *viability* in parallel: Does the multigrid algorithm run on  $P > 1$  MPI processes, and if so, how? What are the PETSc options to control it?
- *scaling* in parallel: For a fixed number of processes  $P > 1$  how does the work—flops or run time—grow with the number of unknowns  $N$ ? (Is it still  $O(N)$ ?) What happens when  $P$  increases or both  $P$  and  $N$  increase?

Parallel viability for the Poisson equation starts with understanding the coarse-grid solve, addressed here, while the question of scaling is deferred to Chapter 8.

On  $P > 1$  processes and a structured (fine) grid of  $N$  points, what does the GMG solver `-pc_type mg` actually do? Consider the 2D Poisson run

```
| $ mpiexec -n P ./fish -pc_type mg -da_refine L
```

First `DMDACreate2d()` is called with arguments corresponding to a  $3 \times 3$  coarsest grid  $\Omega^{(0)}$  (Chapter 6),<sup>26</sup> so `-da_refine L` will generate a fine grid  $\Omega^{(L)}$  with  $m = 2^{L+1} + 1$  points in each direction. The refinement process creates DMDA objects for each level of a *grid hierarchy*, refining from  $\Omega^{(0)}$  to  $\Omega^{(L)}$  by repeatedly calling `DMRefine()`. Each level is partitioned into  $P$  subgrids and distributed across the  $P$  processes. The number of points of each subgrid is not perfectly balanced across processes, but we will assume that on the fine grid  $\Omega^{(L)}$  each process owns roughly  $N/P$  unknowns. As the hierarchy is created, the rest of the PC is set up, as in serial, with interpolation/restriction operators for each level.

Observe that the coarsest-grid problem depends globally on the fine-grid PDE data. For example, in the above runs the  $\Omega^{(0)}$  problem has only one nontrivial equation, at the center of the  $3 \times 3$  grid. Through restriction of the residuals from finer grids, the right side of this equation depends on all the data of the problem, namely the source function  $f$  on the whole domain and the values of  $g$  along the entire boundary.

<sup>26</sup>In other words, corresponding to `-da_grid_x 3 -da_grid_y 3`.

By default the problem on the coarsest grid  $\Omega^{(0)}$  is solved exactly by LU decomposition. However, noting PETSC only implements LU in serial (Chapter 2), the coarsest-grid problem is copied to and solved *redundantly*, in serial on each process. That is, the above run includes these implied defaults:

```
-mg_coarse_ksp_type preonly -mg_coarse_pc_type redundant \
-mg_coarse_redundant_ksp_type preonly -mg_coarse_redundant_pc_type lu
```

Note that **redundant** is a type of PC but it defines a communication pattern, not a computation *per se*. An alternative approach would be to transfer the problem to the rank 0 process and only solve it there, but the **redundant** approach saves a communication stage. That is, though it would require all-to-all communication when copying the  $\Omega^{(0)}$  problem to each processor, the **redundant** solution is available on each process without further communication.

However, the DMDA type requires each process to always *own at least one grid point*, including on the coarsest grid, and this constrains GMG usage. For instance, while the above run succeeds with  $P = 9$ , for  $P > 9$  it fails with a “Partition ... is too fine!” message. By contrast, the PETSC algebraic multigrid type **-pc\_type gamg** (Chapter 10), which does not use a DM object, permits a process to own zero degrees of freedom, as does the unstructured grid DM type **DMPlex** (Chapter 13).

An obvious solution here, using DMDA, is to make the coarsest grid finer. For instance, consider either of these equivalent formulations which set a  $5 \times 5$  coarsest grid:

```
$ mpiexec -n P ./fish -pc_type mg -da_grid_x 5 -da_grid_y 5 -da_refine L-1
$ mpiexec -n P ./fish -pc_type mg -da_refine L -pc_mg_levels L
```

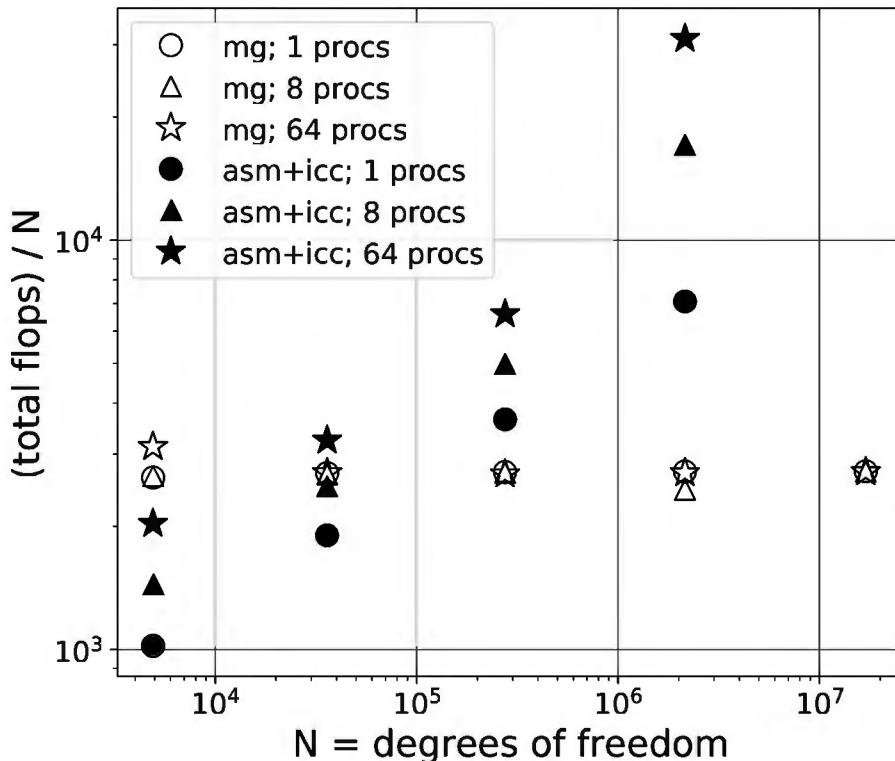
These runs succeed with  $P = 10$  or  $P = 25$ , for example, but fail whenever  $P > 25$ . (The prime factorization of  $P$  determines which  $P \leq 25$  are in fact allowed.)

In other words, for any given coarsest grid, using **redundant** coarse-grid solves, PETSC’s parallel implementation of GMG on a structured DMDA grid imposes an upper bound on  $P$ . As  $P$  increases the user will need to increase the size of the coarsest grid, but doing so comes at a cost. A coarser coarsest grid is more effective at removing low-frequency error components, and making it finer degrades the performance of the solver in either of two ways (Exercise 7.2).

- If the coarsest-grid problem is solved accurately by a suboptimal direct method (LU, Cholesky, SVD), for example, then the number of flops for each multigrid cycle will increase as the cost of the coarsest-grid solve becomes dominant.
- If the coarsest-grid problem is solved inexactly, but by  $O(N)$  work, for example using a fixed number of smoother iterations, then the convergence factor of the multigrid cycle gets worse as the cycles become shallower and less effective at removing low-frequencies in the error.

We introduce the **telescope** PC type [109], an alternative to **redundant** which transcends this performance barrier, in Chapter 8. These considerations also imply that in parallel the W-cycle must be treated with caution. Its many visits to the coarsest grid (Figure 6.13) require copious interprocess communication while doing relatively little arithmetic. Thus V-cycles are the default in PETSC.

In addition to the choice of solution method on the coarsest grid, GMG needs smoothing and interpolation/restriction operations at each level. All of these components are based on **Mat** objects “inside” the GMG PC object, which makes its structure quite complicated (Exercise 7.3). Generally speaking, GMG components are implemented to have a high degree of parallelism [144], at least for DM-based grids and meshes [10], but adjustments are made for the parallel case:



**Figure 7.5.** The work for parallel GMG scales as  $O(N)$  if we fix the number of processes. By contrast, single-level DD (`asm + icc`) is far from optimal.

- SOR (e.g., Gauss-Seidel and SSOR) smoothers are, by default, modified to their “processor-block” versions (page 133). The off-process entries of the matrix do not contribute to the smoother, thus some smoother performance is sacrificed to avoid communication.
- Smoothers use a fixed number of iterations instead of a norm-based convergence test. In fact this applies even in serial, but in parallel it is significant because it avoids norm evaluations (global reductions) which require communication [72]. Thus in `-mg_levels_ksp_converged_reason` output the smoother iterations always succeed with a `CONVERGED_ITS` message.

Regarding the second point, one can completely avoid norm evaluations in multigrid smoothers by choosing `-mg_levels_ksp_type richardson`. This may compromise the convergence rate, for example if the coefficients in (6.31) are highly anisotropic, but `richardson` works well in isotropic Poisson and advection-diffusion (Chapter 11) cases. In any case, parallel performance analysis suggests that Chebyshev iteration (page 137) remains a good default choice in parallel, at least for symmetric problems [3], even though eigenvalue estimation for the `chebyshev` smoother requires norm evaluations.

Because of how the defaults work, the performance of our GMG method for the Poisson equation is relatively independent of  $P$ , beyond the requirement that the coarse grid is large enough for the given  $P$  values. In fact, Figure 7.5 shows the results from 3D Poisson runs with concurrency  $P = 1, 8, 64$ . The runs use a  $5 \times 5 \times 5$  coarsest grid;  $64 < 125$  so  $\Omega^{(0)}$  has at least one point per process. We consider  $L = 2, 3, 4, 5, 6$  levels of grid refinement:

```
$ mpiexec -n P ./fish -fsh_dim 3 -da_grid_x 5 -da_grid_y 5 -da_grid_z 5 \
    -ksp_rtol 1.0e-10 -da_refine L -pc_type mg
```

(The  $L = 6$  grid is  $257 \times 257 \times 257$  with  $N = 1.7 \times 10^7$  unknowns.) The only significant  $P$  dependence in the above runs is inside the smoother, which has a Chebyshev KSP and an SSOR PC. Processor-block SSOR (Chapter 6) degrades as the number of degrees of freedom per process becomes small.

Figure 7.5 shows the total flops per  $N$  as a function of  $N$ , compared to a single-level DD solver (-pc\_type `asm` -sub\_pc\_type `icc`). Parallel GMG does optimal  $O(N)$  work for each fixed  $P$  on this easy example. In Chapter 8 such an analysis using fixed  $P$  and increasing  $N$  is called a *static scaling* study.

## The minimal surface equation

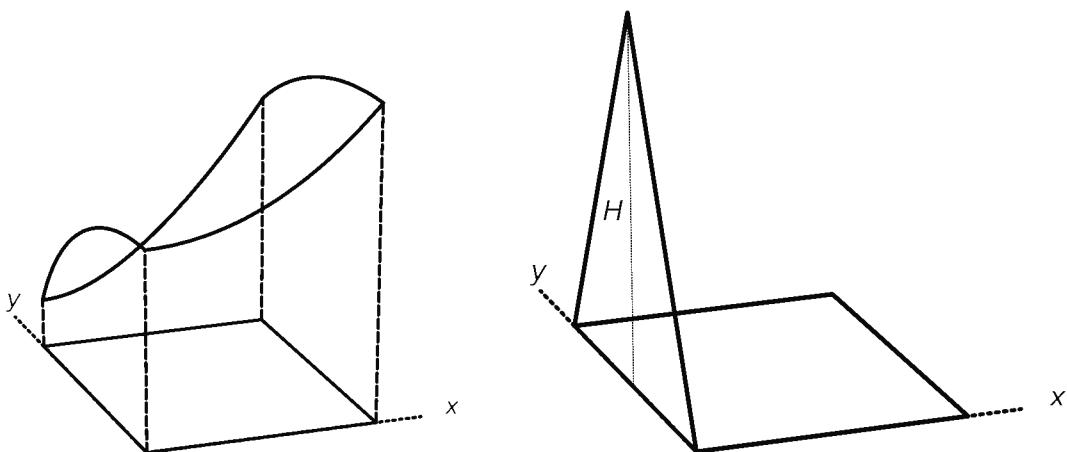
From calculus we know that the area of a differentiable surface  $z = v(x, y)$  on a domain  $\Omega \subset \mathbb{R}^2$  is

$$I[v] = \int_{\Omega} \sqrt{1 + |\nabla v|^2} dx dy. \quad (7.1)$$

The *minimal surface equation* (MSE) [51] is the Euler-Lagrange equation of this functional, a well-known nonlinear and elliptic PDE,

$$-\nabla \cdot \left( \frac{\nabla u}{\sqrt{1 + |\nabla u|^2}} \right) = 0. \quad (7.2)$$

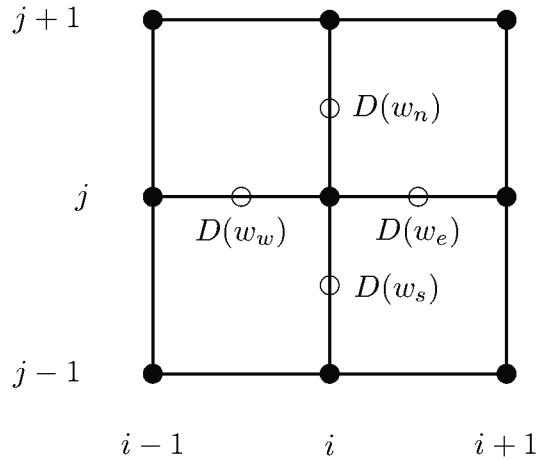
One derives (7.2) by computing the directional derivative of  $I$  at the minimizer  $u$ , in the direction of a test function  $v$ , requiring this to be zero for all  $v$  (Exercise 7.4). A standard problem for (7.2), well posed if  $\Omega$  is convex [60], includes continuous Dirichlet boundary conditions  $u|_{\partial\Omega} = g$ . In zero gravity the function  $u$  would determine the shape of a soap film or elastic membrane spanning the rigid wire frame at height  $g$  along  $\partial\Omega$  [119]. Two such frames are shown in Figure 7.6. Note that the *catenoid* (left) provides an exact solution (Exercise 7.5).



**Figure 7.6.** Catenoid and tent wire frames, i.e., boundary conditions  $g$ .

Equation (7.2) is a *quasilinear* elliptic PDE [60] with solution-dependent coefficient  $(1 + |\nabla u|^2)^{-1/2}$  on the leading-order derivative. It is uniformly elliptic only if  $|\nabla u|$  is bounded above on  $\Omega$ ; this depends on the particular boundary value problem. By contrast, the Liouville-Bratu equation (Exercises 4.5 and 7.12), for example, is a uniformly elliptic *semilinear* PDE with only a zeroth-order nonlinearity.

Our strategy for solving (7.2) is to discretize by finite differences on a structured grid (DMDA). When solving such nonlinear PDE boundary value problems using Newton iterations (SNES),



**Figure 7.7.** A *BOX* stencil FD scheme for equation (7.3).

multigrid (GMG or AMG) preconditioners can be applied to the linear systems for each step, so-called *Newton-multigrid* methods [18, 26, 144]. When CG or GMRES is used as an accelerator (Exercise 7.7), such methods could also be called “Newton-Krylov multigrid.” In any case, an initial Newton iterate is needed. At first we will set all interior values to zero, but we have more to say about this below.

Our residual-evaluation function will generalize (7.2) to the form

$$-\nabla \cdot (D(|\nabla u|^2) \nabla u) = 0. \quad (7.3)$$

The diffusivity  $D$  is thus solution dependent, and we implement

$$D(w) = (1 + w)^q. \quad (7.4)$$

While  $q = -1/2$  in (7.2), the  $q = 0$  case allows testing on the Laplace (Poisson) equation.

Our centered FD scheme for (7.3) uses a structured grid with cell dimensions  $h_x$  by  $h_y$ . The scheme is scaled the same way as (3.7) for the Poisson equation,

$$\begin{aligned} F_{ij}(\mathbf{u}) = & -\frac{h_x}{h_y} (D(w_e)(u_{i+1,j} - u_{i,j}) - D(w_w)(u_{i,j} - u_{i-1,j})) \\ & - \frac{h_y}{h_x} (D(w_n)(u_{i,j+1} - u_{i,j}) - D(w_s)(u_{i,j} - u_{i,j-1})), \end{aligned} \quad (7.5)$$

where subscripts  $e, w, n, s$  indicate cardinal directions (Figure 7.7). A centered scheme for a linear diffusion equation  $-\nabla \cdot (D(x, y) \nabla u) = 0$  would evaluate the diffusivity  $D(x, y)$  at the same “staggered” points (open circles) as in the figure (Exercise 7.9). To maintain  $O(h_x^2 + h_y^2)$  local truncation error [115] the partial derivatives have FD approximations which are centered at these staggered points. For example, at the “east” point  $(x_i + \frac{h_x}{2}, y_j)$  we compute

$$w_e = \left( \frac{u_{i+1,j} - u_{i,j}}{h_x} \right)^2 + \left( \frac{u_{i,j+1} + u_{i+1,j+1} - u_{i,j} - u_{i+1,j}}{4h_y} \right)^2, \quad (7.6)$$

and the other staggered points have similar formulas (Exercise 7.6). Note that this scheme requires `DMDA_STENCIL_BOX` when calling `DMDACreate2d()`.

If  $w = |\nabla u|^2$  is small then  $1 + |\nabla u|^2 \approx 1$  and (7.2) is nearly the Laplace equation. This observation suggests that we may try reusing the Jacobian from the Poisson equation, namely from `ch6/poissonfunctions.h|c`. In fact, in the absence of an exact Jacobian we should choose between `-snes_fd_color` and `-snes_mf_operator`, and the latter can exploit an inexact Jacobian as a preconditioner in the matrix-free numerical derivative. As we will see, both approaches work reasonably well though we will prefer the former. (Exercise 7.11 compares the implementation of an exact Jacobian.)

The above ideas, and little else, go into a new code called `minimal.c` (not shown). Start by compiling it and then looking at options from `-help`:

```
| $ cd c/ch7/ && make minimal
| $ ./minimal -help | grep ms_
```

A first concern is to demonstrate convergence in the default `catenoid` verification case. Consider these runs on  $m \times m$  grids for  $m = 9, 17, \dots, 129$ :

```
| $ for LEV in 2 3 4 5 6; do
|   ./minimal -snes_converged_reason -snes_fd_color -da_refine $LEV; done
```

On the finest grids  $\text{LEV}=5, 6$  this initial attempt fails with DIVERGED errors. The default initial iterate seems to be outside of the domain of convergence of the Newton iteration in these cases.

Option `-ms_exact_init` allows us to initialize using the gridded values of the exact continuum solution. This is not the exact solution of the discrete system, but it should be close on fine grids. Using this option results in convergence to default tolerances in two Newton iterations on each grid. Furthermore the numerical errors  $\|u - u_{\text{exact}}\|_\infty$  decrease by a factor of four at each reduction of  $h$  by two (not shown), so we have good evidence for  $O(h^2)$  convergence of the scheme.

Note that the Jacobian is not symmetric in nonlinear cases  $q \neq 0$ , so GMRES may be a good KSP choice. However, when  $q = 0$  in (7.3) the problem is the Poisson equation and the residual-evaluation code should generate a symmetric Jacobian. Testing this might find bugs in the boundary condition implementation in particular, so we consider the runs

```
| $ ./minimal -snes_fd_color -da_refine 5 -ms_q 0 -mat_is_symmetric TOL
```

with tolerances  $TOL = 10^{-6}, 10^{-7}, 10^{-8}$ . These report “`Matrix is symmetric`,” but for tolerances  $10^{-9}, 10^{-10}$  the matrix is not symmetric, as expected for a finite-differenced Jacobian only accurate to  $O(\sqrt{\epsilon}) \approx 10^{-8}$  (Chapter 4). At this point, based on the runs so far, we propose that the residual evaluation in `minimal.c` is correct.

Regarding preconditioning the Newton steps, GMG is promising for performance on fine grids, but for it to work the residual-evaluation code must be correctly rediscretized on the provided grid using the `DMDALocalInfo` input for `FormFunctionLocal()`. To check that this is correctly done in any SNES- and DMDA-using code, note the solver combination `-pc_type mg -snes_fd_color` should not throw an error. In fact, using `-ms_exact_init` we can confirm that GMG generates three to six KSP iterations per Newton iteration in most cases solved by `minimal.c`. By contrast, for `-pc_type ilu` the number of KSP iterations grows rapidly to over 100 (Exercise 7.8).

Thus we have only two significant convergence issues in solving this PDE problem:

1. The initial iterate  $\mathbf{u}_0 = 0$  is not in the domain of quadratic convergence on fine grids. Consider the runs

```
| $ for LEV in 2 3 4 5 6 7; do
|   ./minimal -snes_converged_reason -snes_fd_color -pc_type mg \
|     -da_refine $LEV -ms_problem PROBLEM; done
```

For catenoid the three finest grids DIVERGE. All six tent runs CONVERGE, but the number of Newton iterations grows rapidly. One might be tempted to code an *ad hoc* scheme for the initial iterate, but an excellent and general-purpose approach is shown in the next section.

2. Newton convergence rates are slower when the solution is less smooth. Using the tent problem to demonstrate, note that solution gradients (slopes)  $|\nabla u|$  are large near the tent door if  $H$  is large (Figure 7.6). Thus the runs

```
$ for HH in 0.1 1 10 100; do
  ./minimal -da_refine 6 -snes_converged_reason -snes_fd_color \
  -pc_type mg -ms_problem tent -ms_tent_H $HH; done
```

require increasing Newton iterations, 7, 11, 13, 15, respectively. While larger nonlinearities are indeed intrinsically harder, these convergence rates are also improved by the next idea.

## Grid sequencing

Efficient solutions of nonlinear PDEs using Newton's method require initial iterates within the domain of convergence of the method. For a good initial iterate, quadratic convergence (Chapter 4) will set in immediately. Finding such a good initial iterate on a coarse grid is apparently much easier than on a fine grid, a hint we now pursue.

With *grid sequencing* one starts on a coarse grid and solves the nonlinear problem to some accuracy. The solution is interpolated to become the initial iterate on a finer grid. After solving and interpolating we generate the initial iterate on a next finer grid, and so on. Like GMG, this strategy needs a grid hierarchy, but grid sequencing is not multigrid. One uses coarse grids only for initial iterates, not as a tool for removing components from the error.

We propose to use *both* grid sequencing and GMG preconditioning. This powerful strategy could be called “grid-sequenced Newton-Krylov multigrid,” undoubtedly a flabby name. However, as it is comparable to full-cycle GMG preconditioning (`-pc_mg_type full`; Chapter 6) we instead call it a *nonlinear full multigrid cycle*.

Grid sequencing in PETSc is an outer iteration executed by SNES. To exploit it, codes must provide a DM for an initial coarsest grid or mesh, create a SNES, attach these with a call to `SNESSetDM()`, and then call `SNESolve()`. All of this is just as expected for GMG and DD preconditioners, but the code must also be prepared to use a solution returned by `SNESolve()` which is on a finer grid than the original DM. As shown in Code 7.1, after the solve the code gets the fine-grid DM and solution Vec by calls to `SNESGetDM()` and `SNESGetSolution()`, respectively. (There is no need to restore or Destroy these objects.)

```
SNESSolve(snes,NULL,u_initial);
DMRestoreGlobalVector(da,&u_initial);
DMDestroy(&da);
SNESGetDM(snes,&da);
SNESGetSolution(snes,&u);
```

**Code 7.1.** `c/ch7/minimal.c`. For grid sequencing a code must get the updated DM and solution Vec after calling `SNESolve()`.

At the command line the key idea is that the new option `-snes_grid_sequence` replaces `-da_refine`. More precisely, DMDA options like `-da_refine` and/or `-da_grid_x` determine the initial, coarsest grid and then `-snes_grid_sequence` adds levels.

Consider the following serial MSE solution which is better than any so far:

```
$ ./minimal -snes_fd_color -snes_converged_reason -pc_type mg -snes_grid_sequence 8
      Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE ... 5
      Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE ... 3
      Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 3
done on 513 x 513 grid and problem catenoid: error |u-uexact|_inf = 2.27604e-07
```

The initial DM<sub>A</sub> holds a  $3 \times 3$  grid. Five Newton iterations are needed there because the initial iterate is poor, but the domain of convergence is large and forgiving, and this solve is fast because the grid is so coarse. Then the SNES asks the DM<sub>A</sub> for the next-finer grid (`DMRefine`), and an interpolant of the current solution for use as the next initial iterate. This is repeated at each level. That only three Newton iterations are needed on all of the finer levels shows that the interpolant of the previous converged solution is a good initial iterate. Indented output displays the grid sequencing. Adding

```
-mg_{levels,coarse} ksp_converged_reason
```

gives the nonlinear full cycle indentation profile, similar to a linear full cycle (Figure 6.14), but with control from both SNES and PC objects.

Before looking at optimality we should consider the smoothness of MSE solutions. Recalling Figure 7.6, and based on Exercise 7.5, if  $c > 1$  then the `catenoid` solution has bounded derivatives of all orders on  $\Omega = (0, 1)^2$ . However, as  $c \rightarrow 1$  the maximum magnitude of the solution gradient diverges. Problem `tent` is even worse. The parameter  $H$  controls the maximum size of the gradient, but for any  $H$  the magnitude of second derivatives, such as  $|u_{yy}|$  near the top of the tent door, is unbounded on  $\Omega$ .

The nonlinear full cycle strategy gives optimal complexity up to  $N \approx 10^6$ , at least, if the solution gradient is not too large. Consider runs of the form

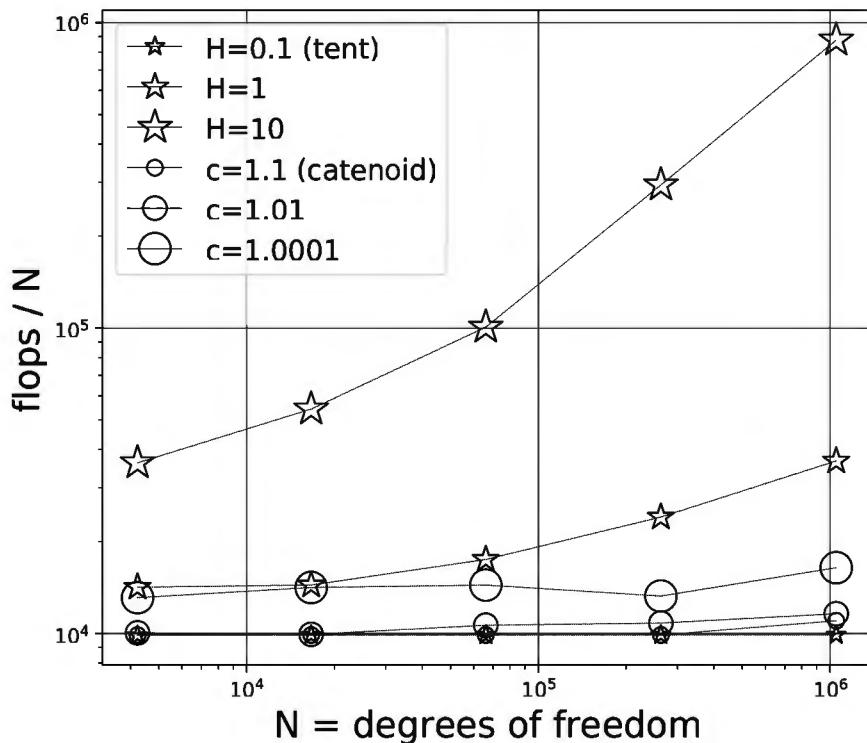
```
| $ ./minimal -snes_fd_color -pc_type mg -snes_grid_sequence LEV
```

for  $LEV = 5, 6, 7, 8, 9$ ; the last has  $N = 1.05 \times 10^6$ . Using `-log_view` output to count flops, Figure 7.8 shows the flops-per- $N$  result for cases  $c = 1.1, 1.01, 1.0001$  for `catenoid` and  $H = 0.1, 1, 10$  for `tent`. For these parameter ranges, measured solver complexity is no worse than  $O(N^{1.03})$  for `catenoid`, but it increases from  $O(N^{1.00})$  to  $O(N^{1.58})$  for `tent` as  $H$  grows. Evidently, a lack of smoothness can have a performance impact.

In terms of the two convergence issues listed above (page 185), we have addressed the first by grid sequencing. We will not make further progress with the second, but one approach would use *adaptive mesh refinement* [19, 24, 144] (AMR) near locations of large gradients. Note that while Chapters 10, 13, and 14 consider unstructured meshes, AMR itself is outside the scope of this book.

Regarding parallel runs, this nonlinear full cycle approach has similar parallel performance to that already seen from CG+GMG solvers for the linear Poisson equation. We will address the parallel performance of `minimal.c` at higher resolution in the next chapter.

As a final note let us return to the Jacobian. Instead of using `-snes_fd_color` we may apply preconditioned Jacobian-free Newton-Krylov (JFNK) using option `-snes_mf_operator`. The



**Figure 7.8.** For the minimal surface equation, a nonlinear full cycle solver, i.e., grid-sequenced Newton-Krylov multigrid, has optimal or near-optimal complexity except when unbounded second derivatives degrade performance.

assembled analytic Jacobian for the Poisson equation is used as preconditioner material. There is, however, a large performance difference between these approaches. For example, the runs

```
$ ./minimal -pc_type mg -{snes,ksp}_converged_reason \
-snes_grid_sequence 9 JAC
```

with  $JAC = -snes\_fd\_color, -snes\_mf\_operator$  each converge in three Newton iterations on the finest grid. With  $-snes\_mf\_operator$  there are substantially more KSP iterations at each Newton step so that the run time is about four times longer. A custom monitor, described next, helps to more precisely diagnose the difficulties of JFNK on this example.

## A SNES monitor for the minimal surface equation

Equation (7.2) determines the surface of minimal area spanning a given wire frame, so it would be nice to see the surface area decrease during the Newton iteration. Also, to address the second convergence issue in our list on page 185, we want to compute bounds on the solution-dependent diffusivity  $D(|\nabla v|^2) = (1 + |\nabla v|^2)^{-1/2}$ .

To compute solution area and diffusivity bounds at every SNES iteration we need to treat each Newton iterate  $\mathbf{v}_k$  as a function  $v(x, y)$  on  $\Omega$  with a well-defined gradient. A disadvantage of the FD paradigm is that functions are only represented by their grid values, but, looking forward, the next chapter introduces a  $Q_1$  finite element (FE) method in which  $\mathbf{v}_k$  represents a function. We exploit this idea in the monitor code as follows.

Let

$$\square_{ij} = [x_{i-1}, x_i] \times [y_{j-1}, y_j]$$

be a rectangular cell with dimensions  $h_x$  and  $h_y$ . If  $\mathbf{v} \in \mathbb{R}^N$  corresponds to grid values  $v_{i,j}$  then

the interpolant

$$\begin{aligned} v(x, y) = & v_{i,j} \frac{(x - x_{i-1})(y - y_{j-1})}{h_x h_y} + v_{i-1,j} \frac{(x_i - x)(y - y_{j-1})}{h_x h_y} \\ & + v_{i-1,j-1} \frac{(x_i - x)(y_j - y)}{h_x h_y} + v_{i,j-1} \frac{(x - x_{i-1})(y - y_{j-1})}{h_x h_y} \end{aligned} \quad (7.7)$$

is a smooth function on  $\square_{ij}$  with the given values at the corners. One may then compute partial derivatives  $\partial v / \partial x, \partial v / \partial y$  (Exercise 7.10) to get an expression for  $|\nabla v|^2$  on  $\square_{ij}$ . The vector  $\mathbf{v} \in \mathbb{R}^N$  corresponds to a function on  $\bar{\Omega} = [0, 1]^2$  which is continuous and piecewise smooth, with bounded, but discontinuous, gradient.

Next, each integral in the sum

$$I[v] = \sum_{i,j} \int_{\square_{ij}} \sqrt{1 + |\nabla v|^2} dx dy$$

is evaluated accurately by Gauss-Legendre quadrature, namely formulas (I.2) and (I.3) from the Interlude (page 171).

The above formulas go into a function `MSEMonitor()` (not shown), and a call-back via `SNESMonitorSet()` is set by option `-ms_monitor`. (Compare with `TSMonitorSet()` in Chapter 5.) The result looks like the following on the `tent` problem and a  $129 \times 129$  grid:

```
$ ./minimal -ms_problem tent -da_refine 6 -pc_type mg -snes_fd_color \
-snes_converged_reason -ms_monitor
area = 1.49258333; 0.0078 <= D <= 1.0000
area = 1.34061346; 0.0125 <= D <= 1.0000
...
area = 1.32348541; 0.0279 <= D <= 1.0000
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 11
done on 129 x 129 grid and problem tent ...
```

Note that the surface areas decrease monotonically. Replacing `-da_refine` with `-snes_grid_sequence` shows that in the nonlinear full multigrid cycles the initial states on each grid level also become close to the final solution in surface area.

The bounds on diffusivity  $D = (1 + |\nabla v|^2)^{-1/2}$  range over almost two orders of magnitude in the above case. It follows that the constant-coefficient Laplace operator, i.e., the matrix used with `-snes_mf_operator`, does not match the spectrum of the true Jacobian at all well; the preconditioned operator is not close to the identity. Adding `-ksp_view_singularvalues`, with either `-snes_fd_color` or `-snes_mf_operator`, confirms that the preconditioned operator for JFNK has large condition number. Among other conclusions, it follows that adding code for an exact Jacobian is reasonably justified (Exercise 7.11).

In summary, we have resolved our MSE convergence issues identified earlier—or at least understood them better. On the other hand, the parallel scaling of the solver remains unexplored; see Chapter 8.

## The biharmonic equation as a coupled system

We have not yet applied one of the preconditioning strategies mentioned in Chapter 6, namely `-pc_type fieldsplit` (Figure 6.4). To introduce it we use the fourth-order biharmonic equation:

$$\nabla^4 u = u_{xxxx} + 2u_{xxyy} + u_{yyyy} = f. \quad (7.8)$$

Note that the biharmonic operator  $\nabla^4 = (-\nabla^2)^2$  is nonnegative.

Equation (7.8) is a model for small deflections of thin plates under distributed loads  $f$  [24, 36]. We consider homogeneous boundary conditions such that the edge of the plate is “simply supported” at height zero, with no resistance to bending along the direction normal to the boundary:

$$u = 0 \quad \text{and} \quad \nabla^2 u = 0 \quad \text{on } \partial\Omega. \quad (7.9)$$

Consider also the problem which combines (7.8) with boundary conditions  $u = 0$  and  $\nabla^2 u + (1 - \nu)u_{\tau\tau} = 0$ , where  $u_{\tau\tau}$  is the second derivative of  $u$  taken tangentially to the boundary  $\partial\Omega$ . The weak form of this problem is well posed for  $0 < \nu < 1$  [24, section 5.9]. For boundary conditions (7.9) on the unit square  $\Omega = (0, 1)^2$ ,  $u = 0$  along  $\partial\Omega$  implies  $u_{\tau\tau} = 0$  along  $\partial\Omega$  almost everywhere. Thus, at least in the case we actually solve, conditions (7.9) are also well posed.

If an FD method is applied directly to (7.8), via an approximation of fourth-order derivatives, then the stencil is large and the implementation of boundary conditions requires thought. While this can certainly be done—Exercise 7.13 requests the 1D version—we may instead replace (7.8) with a coupled system of two second-order equations [135]. In fact, with the substitution  $v = -\nabla^2 u$  the system has triangular block form:

$$(7.8) \iff \begin{aligned} -\nabla^2 v &= f \\ -\nabla^2 u &= v \end{aligned} \iff \begin{bmatrix} -\nabla^2 & 0 \\ -I & -\nabla^2 \end{bmatrix} \begin{bmatrix} v \\ u \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}. \quad (7.10)$$

The boundary conditions are Dirichlet for both components,  $u = 0$  and  $v = 0$  on  $\partial\Omega$ , so this approach is easy to implement.

Our implementation of system (7.10) in a SNES-based code `ch7/biharm.c` is straightforward and not shown. We manufacture an exact solution  $u(x, y) = c(x)c(y)$ , where  $c(x)$  is a 6th-degree polynomial which satisfies boundary conditions (7.9). Then we create a 2D DMDA structured grid with `dof = 2`. Calling `DMDASetFieldName()` allows us to refer to components by names “`u`” and “`v`” in `fieldsplit` usage below. The Vecs associated to this DM, including the solution, are stored in interleaved form  $v_{0,0}, u_{0,0}, v_{1,0}, u_{1,0}, \dots$ . Local evaluation routines use

```
typedef struct {
    double v, u;
} Field;
```

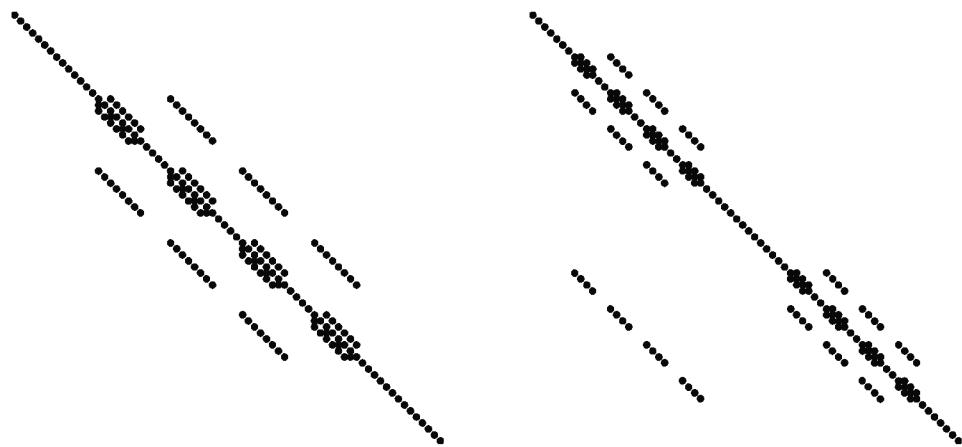
to refer to  $v_{i,j}$  as `W[j][i].v`, for example, where `W` has type `Field**`; compare with `pattern.c` in Chapter 5. The residual evaluation `FormFunctionLocal()` (not shown) computes scalar residuals  $F[j][i].v \approx -\nabla^2 v - f$  and  $F[j][i].u \approx -v - \nabla^2 u$  at each interior grid point  $(x_i, y_j)$ . Trivial equations apply at each boundary point and we scale the rows just as we did in Chapter 6.

The familiar five-point stencil FD discretization of the scalar Laplacian  $-\nabla^2$  (Chapter 3) generates block  $A$ , an SPD matrix, in the system

$$Tw = g \quad \text{where} \quad T = \begin{bmatrix} A & 0 \\ -I & A \end{bmatrix}, \quad w = \begin{bmatrix} v \\ u \end{bmatrix}, \quad g = \begin{bmatrix} f \\ 0 \end{bmatrix}. \quad (7.11)$$

The matrix  $T$  is nonsymmetric so we set the default KSP type to GMRES; we also set the SNES type to KSPONLY because the problem is linear. Note that option `-ksp_view_mat` displays a sparsity pattern shown on the left in Figure 7.9, reflecting the interleaved storage order.

Because the DMDA allocates ten nonzeros per (generic) row in the discrete system (7.11), i.e., based on `dof = 2` and a star stencil with (half-)width one, the cost of a single `-snes_fd_color` Jacobian approximation (Chapter 4) is more than 10 residual evaluations. It follows that programmer effort to implement an analytical Jacobian implementation, as we have done



**Figure 7.9.** Sparsity of  $T$  in the original, interleaved form (left) and reordered into blocks (right; equation (7.11)) with scalar Laplacians on the diagonal.

(`FormJacobianLocal()`; not shown), is justified. Performance profiling of flops or time also justifies this.

When using the exact solution for verification, norm errors are somewhat larger for the  $u$  component than for the  $v$  component, though both converge at the expected rate. In fact, adding `-ksp_rtol 1.0e-10`, to runs on grids of  $33^2$  up to  $2049^2$  points, yields rates  $\|v - v_{\text{exact}}\|_\infty = O(h^{2.0002})$  and  $\|u - u_{\text{exact}}\|_\infty = O(h^{2.0004})$  (not shown).

## Block-structured preconditioning

Our purpose in building a biharmonic equation solver is to introduce `-pc_type fieldsplit`. This preconditioner type exploits information from a DM, namely that `dof = 2` in this case. The matrix  $T$  in (7.11) has block size two so, in the interleaved storage form,  $2 \times 2$  blocks are allocated in the same pattern as the nonzeros of the scalar Laplacian. However, `fieldsplit` permits us to view  $T$  blockwise as in equation (7.10) and the right side of Figure 7.9. No memory needs to be copied or moved for this interpretation, but run-time control of `fieldsplit` is based on this blockwise view.

For an example of the kind of high-level, blockwise action made possible by `fieldsplit`, consider the following calculation to invert  $T$ :

$$\underbrace{\begin{bmatrix} I & 0 \\ 0 & A^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ I & I \end{bmatrix} \begin{bmatrix} A^{-1} & 0 \\ 0 & I \end{bmatrix}}_{T^{-1}} \underbrace{\begin{bmatrix} A & 0 \\ -I & A \end{bmatrix}}_T = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}. \quad (7.12)$$

One may regard this either as blockwise forward substitution to solve a lower triangular system or as a single step of blockwise Gauss-Seidel.

If we compute the action of  $A^{-1}$  by a direct LU solver then the following command line implements (7.12) to solve the PDE problem (7.10) on a  $33 \times 33$  grid:

```
$ ./biharm -da_refine 4 -pc_type fieldsplit \
    -fieldsplit_v_pc_type lu -fieldsplit_u_pc_type lu
```

This is a direct solution method, composed from a step of  $2 \times 2$  block Gauss-Seidel, because the default `fieldsplit` type is `multiplicative` (below), plus direct factorization on the blocks, thus  $M^{-1}T = I$ . This solver does not (yet) have good complexity as  $N \rightarrow \infty$ , however.

It is important to identify some unstated defaults in the above command line. First, the default KSP type is GMRES, but one may check with `-ksp_monitor` that in one iteration the residual norm is reduced by a factor of  $10^{16}$ , so this is indeed a direct solver; `-ksp_type preonly` gives the same solver without norm checks. Second, there are both KSP and PC objects for each field component, but the default is to precondition only; options `-fieldsplit_{v,u}_ksp_type preonly` are implied.

An analogy from Chapter 6, regarding composition of corrections, applies here to `fieldsplit` types. Namely, Jacobi  $\sim$  additive and Gauss-Seidel  $\sim$  multiplicative. In fact, only three choices of `fieldsplit` type are considered in this book:

```
-pc_fieldsplit_type multiplicative|additive|schur
```

The third type `schur` is critical to solving the Stokes equations in Chapter 14, but it does not make sense here because the diagonal blocks are invertible; we are in the easier case here.

If option `-pc_fieldsplit_type additive` is added to the above run, along with `-ksp_monitor`, we see that the method takes exactly two iterations of GMRES, and the same is true with `-ksp_type richardson`. That is, `multiplicative` gives a direct solver but `additive` is apparently *also* a direct solver when one takes an additional iteration. The reason for this is worth pursuing, in part because a related idea will precondition the Stokes equations in Chapter 14.

The following calculation, assuming that the blocks are inverted exactly, shows the action of `-pc_fieldsplit_type additive`:

$$M^{-1}T = \begin{bmatrix} A^{-1} & 0 \\ 0 & A^{-1} \end{bmatrix} \begin{bmatrix} A & 0 \\ -I & A \end{bmatrix} = \begin{bmatrix} I & 0 \\ -A^{-1} & I \end{bmatrix}. \quad (7.13)$$

We have not solved the system with this one preconditioner application, because  $M^{-1}T$  is not yet the identity, but now the preconditioned matrix has only a single eigenvalue  $\lambda = 1$ . In fact,

$$(M^{-1}T - I)^2 = \begin{bmatrix} 0 & 0 \\ -A^{-1} & 0 \end{bmatrix}^2 = 0, \quad (7.14)$$

so  $M^{-1}T$  has a minimal polynomial of degree two. In terms of equation (7.11), and given initial iterate  $\mathbf{w}_0 = 0$  and Richardson iteration, we would compute  $\mathbf{w}_1 = M^{-1}\mathbf{g}$  and then (7.14) implies  $T\mathbf{w}_2 = \mathbf{g}$  (Exercise 7.14).

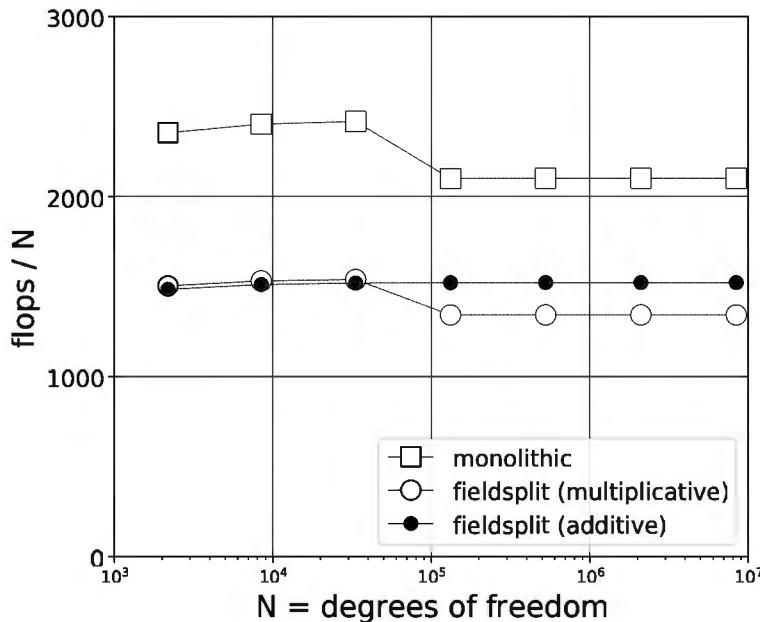
Such on-paper calculations apply  $A^{-1}$  on each block, but in fact we use a preconditioned iteration, denoted  $\text{ksp}(A, P)$ , to approximate the action of  $A^{-1}$ . (Recall that a KSP holds two potentially different matrices  $A$  and  $P$ , with  $P$  denoting the preconditioner material. See Chapter 6.) Thus the correct description of the matrix action of `-pc_fieldsplit_type multiplicative`, in the special case where the system matrix is  $T = \begin{bmatrix} A & 0 \\ -I & A \end{bmatrix}$ , is

$$M^{-1} = \begin{bmatrix} I & 0 \\ 0 & \text{ksp}(A, P) \end{bmatrix} \begin{bmatrix} I & 0 \\ I & I \end{bmatrix} \begin{bmatrix} \text{ksp}(A, P) & 0 \\ 0 & I \end{bmatrix},$$

while type `additive` is

$$M^{-1} = \begin{bmatrix} \text{ksp}(A, P) & 0 \\ 0 & \text{ksp}(A, P) \end{bmatrix}.$$

For the general case see [10].



**Figure 7.10.** Measured by work per degree of freedom, all three GMG solvers are optimal, with advantage to the *fieldsplit* methods.

Using direct solvers on the blocks will yield poor solver-complexity asymptotics. We can, however, construct optimal solvers by switching to GMG preconditioners. Consider this GMRES+GMG solver using the default *multiplicative*-type *fieldsplit*:

```
$ ./biharm -da_refine X -pc_type fieldsplit \
    -fieldsplit_v_pc_type mg      -fieldsplit_u_pc_type mg \
    -fieldsplit_v_pc_mg_levels LEV -fieldsplit_u_pc_mg_levels LEV \
    -fieldsplit_v_pc_mg_galerkin  -fieldsplit_u_pc_mg_galerkin
```

This nontrivially composed preconditioner does a V-cycle on the  $v$  block, then (nearly) eliminates the  $v$  variables, and then does a V-cycle on the  $u$  block. (Note that *fieldsplit* currently only allows Galerkin coarsening (Chapter 6) on the blocks, and that the number of levels on the blocks must be set explicitly.) If a V-cycle were the exact inverse  $A^{-1}$  then this multiplicative solver would converge in one iteration. In reality each V-cycle reduces the residual norm by a factor of  $10^2$  to  $10^3$ , so just a few iterations are needed.

If we were to change to *-pc\_fieldsplit\_type additive* then the first iteration makes much less progress. However, at the second iteration there is a large drop in residual norm.

On the other hand, an alternative to *fieldsplit* is to apply GMG to the whole system, essentially disregarding the block structure. We call this the *monolithic* [27] GMG choice:

```
| $ ./biharm -da_refine X -pc_type mg -pc_mg_levels LEV
```

Now we have three GMG-based solvers of the biharmonic problem: multiplicative *fieldsplit*, additive *fieldsplit*, and monolithic. On  $33^2$  to  $2049^2$  grids, with  $N = 10^3$  to  $N = 10^7$  degrees of freedom, respectively, these solvers all converge in two or three KSP iterations using the default *-ksp\_rtol 1.0e-5* (not shown). Figure 7.10 reveals that their work per degree of freedom is very steady across this  $10^4$  increase in  $N$ . The advantage is to *-pc\_type fieldsplit*, but all three are optimal solvers. Algebraic multigrid (AMG; Chapter 10) also generates (nearly) optimal solvers in the same three modes (Exercise 7.15).

## Exercises

- 7.1. Reproduce the runs in Figures 7.2–7.4. (Note that `fish.c` is in `ch6/`, use a `-with-debugging=0` configuration, and watch memory usage on the finest grids.) Then replace `-pc_type mg` with simpler preconditioners like Jacobi and ICC to feel how much it hurts the convergence rate for the finer grids.
- 7.2. Consider the following serial V-cycle runs for a 3D Poisson problem, in which the coarsest grid problem is solved directly by LU with nested-dissection ordering (Chapter 2):

```
| $ ./fish -fsh_dim 3 -ksp_rtol 1.0e-10 \
|   -pc_type mg -da_refine LEV -pc_mg_levels DEPTH
```

Note that the coarsest grid is  $3 \times 3 \times 3$  if  $\text{DEPTH} = \text{LEV} + 1$ , but otherwise it is finer, with  $3^3, 5^3, 9^3, \dots$  points as  $\text{DEPTH}$  decreases. Alternatively we could add options such as

```
-mg_coarse_ksp_type cg -mg_coarse_pc_type jacobi \
-mg_coarse_ksp_max_it 2 -mg_coarse_ksp_convergence_test skip
```

so that two steps of Jacobi-preconditioned CG are used to (inexactly) solve the coarse problem. Use the above runs to demonstrate, in serial, the two modes of degradation of performance, as  $\text{DEPTH}$  decreases, described in the discussion of parallel multigrid. For example, if you use  $\text{LEV} = 6$  for a  $129 \times 129 \times 129$  grid then try  $\text{DEPTH} = 7, 6, 5, 4, 3$ . Graph flops and KSP iterations.

- 7.3. When run in parallel, the variety of Mats inside a GMG PC is remarkable. For example, the run

```
| $ mpiexec -n 4 ./fish -pc_type mg -da_refine 2
```

generates a very modest grid hierarchy:  $\Omega^{(0)}$  is  $3 \times 3$ ,  $\Omega^{(1)}$  is  $5 \times 5$ , and  $\Omega^{(2)}$  is  $9 \times 9$ . Add `-dm_view` to the above run to see the way the grids  $\Omega^{(i)}$  are distributed. Now add `-info` to the above run and pipe the output through `grep "Matrix size"` to see the many sequential (i.e., process-owned) Mat sizes. Some of these are the diagonal blocks of the matrices  $A^{(i)}$  but others are for interpolation or for the coarse grid problem. Identify as many of these Mats as you can.

- 7.4. For this problem you will show that (7.2) follows if  $u$  minimizes  $I[\cdot]$  in (7.1). Assume that  $u$  is a smooth function on  $\Omega$  with boundary values  $g$ , and assume that  $v$  is a smooth function with zero boundary values. Follow these steps:

- (i) For  $\epsilon \in \mathbb{R}$ , compute  $I[u + \epsilon v] - I[u]$  and simplify, using the binomial theorem for  $(1+z)^{1/2}$ , to

$$I[u + \epsilon v] - I[u] = \epsilon \int_{\Omega} \frac{\nabla u \cdot \nabla v}{\sqrt{1 + |\nabla u|^2}} + O(\epsilon^2).$$

- (ii) Integrate by parts to conclude that, for all  $v$ ,

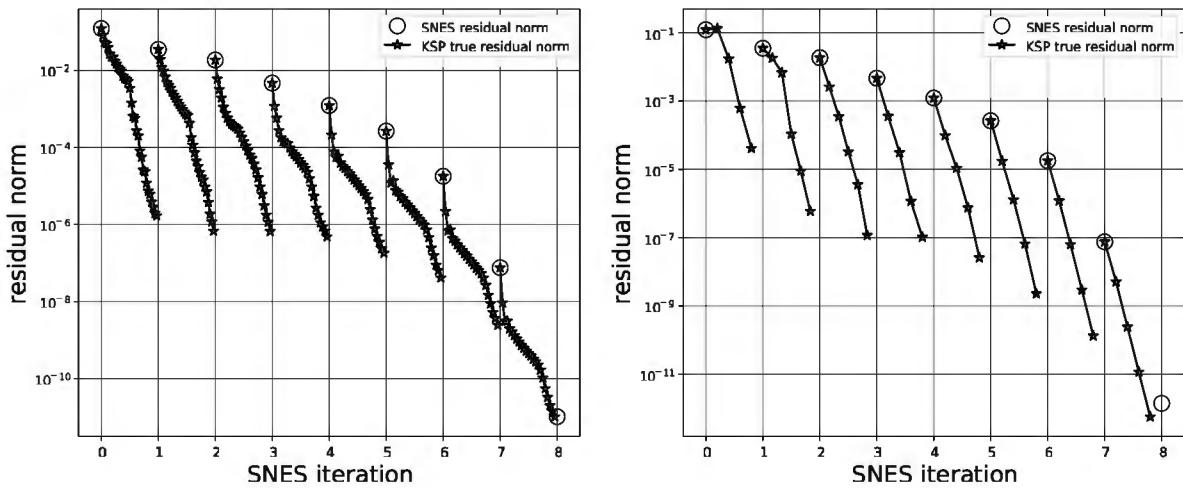
$$I'[u](v) = \lim_{\epsilon \rightarrow 0} \frac{I[u + \epsilon v] - I[u]}{\epsilon} = - \int_{\Omega} \nabla \cdot \left( \frac{\nabla u}{\sqrt{1 + |\nabla u|^2}} \right) v.$$

- (iii) Conclude that (7.2) applies at all points in the open set  $\Omega$ .

The same kind of calculation is done at the beginning of Chapter 9.

- 7.5. For any  $c \geq 1$ , a catenoid surface on  $\Omega = (0, 1)^2$  is given by the formula

$$u(x, y) = c \cosh\left(\frac{x}{c}\right) \sin\left(\arccos\left(\frac{y}{c \cosh(x/c)}\right)\right). \quad (7.15)$$



**Figure 7.11.** Newton-Krylov solutions of (7.2) exhibit quadratic convergence (both). ILU(0) preconditioning stagnates (left) while GMG is fast (right).

(The restriction of  $u$  to  $\partial\Omega$  is shown in Figure 7.6, for  $c = 1.1$ .) Show that  $u$  satisfies (7.2), and that if  $c = 1$  then  $|\nabla u|$  is unbounded on  $\Omega$ .

- 7.6. Write down the finite difference formulas analogous to formula (7.6) for the other (cardinal) staggered-grid points. Show that these formulas, combined with (7.5), give a  $O(h_x^2 + h_y^2)$  scheme for (7.3).
- 7.7. If multigrid tools are used effectively then the Krylov accelerator is often unimportant. For example, the runs

```
$ ./minimal -snes_fd_color -pc_type mg \
-snes_grid_sequence LEV -ksp_type KSP
```

give results relatively independent of the KSP choice, at least among GMRES, CG, and Richardson. (Note that Richardson corresponds to Newton-multigrid, i.e., without acceleration.) What are the differences? Add `-snes_monitor -ksp_converged_reason` and explain.

- 7.8. When working to diagnose Newton-Krylov performance, or bugs, the option combination  
`-snes_monitor -ksp_monitor_true_residual`

is often useful. This prints, using comparable norms, both nonlinear residual norms  $\|\mathbf{F}(\mathbf{u}_k)\|$  in the outer iteration and linear residual norms  $\|\mathbf{b} - \mathbf{A}\mathbf{s}^{(j)}\|$ , where  $\mathbf{b} = -\mathbf{F}(\mathbf{u}_k)$  and  $\mathbf{A} = J(\mathbf{u}_k)$ , in the inner iteration.

The script `c/sneskspplot.py` visualizes these numbers. For example, consider these runs of `minimal.c` using `X=ilu,mg`:

```
$ ./minimal -ms_problem tent -da_refine 5 -snes_fd_color \
-snes_monitor -ksp_monitor_true_residual -pc_type X >& X.txt
```

What the numbers show is that the ILU(0) preconditioner is causing stagnation of the GMRES iteration, while GMG is not—it generates constant residual norm reductions. In both cases the Newton iteration exhibits quadratic convergence (Chapter 4). Generate graphs like Figure 7.11:

```
$ ./sneskspplot.py --ksptrue -o X.pdf X.txt
```

- 7.9. Write a code `diffusion.c` for the linear 2D equation

$$-\nabla \cdot (D(x,y) \nabla u) = f(x,y), \quad (7.16)$$

with Dirichlet boundary conditions  $g(x, y)$ , on a rectangle. The following strategy is recommended:

- (i) Start from `fish.c`. Implement a new residual evaluation by reusing the FD scheme (7.5) for the MSE; see Figure 7.7.
- (ii) Test for correctness in the case  $D(x, y) = 1$ , using the same domain and boundary conditions as in `fish -fsh_problem manuexp`, and recover existing convergence results. Then test on a discontinuous diffusivity such as

$$D(x, y) = \begin{cases} \alpha, & (x - 1/2)^2 + (y - 1/2)^2 < (1/4)^2, \\ 1 & \text{otherwise.} \end{cases}$$

Using `-ksp_monitor_solution draw`, visualize the solution for  $\alpha = 1, 10$ . (Additional testing might use a manufactured solution.)

- (iii) For runs with `-snes_fd_color` and `-snes_mf_operator`—the latter reusing the Jacobian from the Poisson equation—do you observe optimality of the solver?
  - (iv) Implement a Jacobian. Was your labor justified?
- 7.10. Verify that (7.7) defines  $v(x, y)$  so that  $v(x_k, y_l) = v_{k,l}$ . Then compute expressions for  $\partial v / \partial x, \partial v / \partial y$ . Check the correctness of `MSEMonitor()` in `minimal.c`.
- 7.11. (*This exercise requires a nontrivial amount of error-prone work.*) Extend `minimal.c` by implementing an exact Jacobian for FD scheme (7.5). Compare performance with `-snes_fd_color`.
- 7.12. (*Compare Exercise 4.5.*) The Liouville-Bratu equation in 2D is

$$-\nabla^2 u - \lambda e^u = 0 \quad (7.17)$$

for  $\lambda > 0$  [23, 106]. This semilinear [51] elliptic PDE can be solved by the same strategies we used for equation (7.2).

- (i) Start a new code called `bratu2D.c`, based on `minimal.c`. Suppose  $\Omega = (0, 1)^2$  and implement only the minimum, namely a residual-evaluation function for (7.17) based on the obvious FD approximations. Again reuse the Poisson Jacobian. Allow arbitrary Dirichlet boundary conditions  $u = g(x, y)$  along  $\partial\Omega$ .
- (ii) The expected critical parameter is  $\tilde{\lambda} \approx 6.81$  if  $g(x, y) = 0$  [144]. Confirm this by demonstrating convergence of the Newton iteration for  $\lambda < \tilde{\lambda}$ , and divergence for  $\lambda > \tilde{\lambda}$ , on sufficiently fine grids. Test with multiple Jacobian strategies: `-snes_fd_color`, the Poisson Jacobian used as preconditioner material (`-snes_mf_operator`), or the Poisson Jacobian used as-is. (Which yield quadratic convergence?)
- (iii) Using `-pc_type mg` and some of the above Jacobian strategies, demonstrate optimal  $O(N)$  scaling.
- (iv) One should be careful to solve the PDE correctly. An exact solution can be built by manufacturing a solution, i.e., adding a right-hand side to (7.17). However, if you are Joseph Liouville in 1853 [106] then you just write down an infinite family of exact solutions to equation (7.17) itself. For example, let

$$\omega(x, y) = \frac{x^2 + y^2}{(x^2 + y^2)^2 + 1)^2}.$$

Show this function satisfies  $\nabla^2(\log \omega) = -32\omega$  and thereby show that

$$u(x, y) = \log(32\omega(x+1, y+1))$$

is a smooth exact solution of (7.17) in the case  $\lambda = 1$ . Use this exact solution to verify `bratu2D.c`.

- 7.13. Construct a SNES-based code `biharm1.c` which uses a five-point central difference scheme

$$u''''(x) = \frac{u(x-2h) - 4u(x-h) + 6u(x) - 4u(x+h) + u(x+2h)}{h^4} + O(h^2)$$

directly on the boundary value problem

$$u'''' = f, \quad u(0) = u(1) = u''(0) = u''(1).$$

If correctly implemented, the system matrix will be pentadiagonal and SPD. Find an exact solution suitable for testing, and demonstrate using very fine grids that both a direct linear algebra method and a multigrid-preconditioned Krylov method are optimal solvers.

- 7.14. Show that if  $\mathbf{w}_0 = 0$  then in exact arithmetic the second iterate  $\mathbf{w}_2$  from

```
$ ./biharm -da_refine LEV -ksp_type richardson \
    -pc_type fieldsplit -pc_fieldsplit_type additive \
    -fieldsplit_v_pc_type lu -fieldsplit_u_pc_type lu
```

solves  $T\mathbf{w} = \mathbf{g}$ , i.e., equation (7.10). (That is, fill in the details of the argument following (7.14).) Then confirm this in practice for modest settings of LEV; note `-ksp_view` says “initial guess is zero.”

- 7.15. Replace GMG with AMG, i.e., use `gamg` or `hypre`, in the three solvers which produced Figure 7.10, and generate the corresponding figure. You will find that AMG is nearly as fast as GMG for this simple, linear problem in which we are discretizing the usual Laplacian operator.

# Chapter 8

# Parallel scaling

Many users arrive at PETSC wanting to do large-scale parallel computations, knowing that the mathematical concepts in this library can lead to high performance parallel solutions of nonlinear PDEs and multiphysics models. Thus we need a vocabulary for quantifying parallel performance, and this chapter defines several measures of how solver performance scales as the number of processes increases. However, the scaling of algorithmic work done by the solver, as the number of degrees of freedom increases, i.e., the solver complexity (Chapter 7), is at least as important and will remain prominent.

High performance computing (HPC) as a subject is not really covered here, however. For example, only MPI parallelism [72] is considered at all, so texts on HPC, such as [35, 48, 75], are the place to learn about threads, OpenMP, GPUs, CUDA, and so on. Indeed, our model of parallel computation in this chapter is very basic. We will run examples in parallel with only minimal consideration of the physical processors they live on, and the actual design of supercomputers (clusters) is not taken seriously. Practical batch-system commands—e.g., how to request resources or observe the state of jobs on clusters—are not addressed. Finally, we only explore parallel scaling at modest levels of concurrency. Nonetheless, well-written PETSC codes will run in parallel with little effort, and experiments with parallel scaling are straightforward when HPC resources allow. In later chapters, when our focus returns to the mathematics and numerical analysis of interesting PDE problems, parallel performance measurements appear in several cases.

## Consumable resources on clusters

How much does a numerical PDE solution *cost*? The programmer hours spent learning the mathematics and writing the code are often the dominant cost, but, after the code is debugged, processor time gets consumed in “production runs” for a given engineering or scientific purpose. For PETSC codes, such runs should be done in a configuration with compiler optimizations turned on and without debugging symbols, that is, timing measurements should only occur in a `-with-debugging=0` configuration.

Consider a production run to solve a PDE on a given grid or mesh, seeking a given accuracy, namely some specified level of residual norm reduction relative to that of the initial iterate. Setting up such a computation will determine the following two variables:

### Definition.

- $N$ , the *problem size*, the number of real degrees of freedom.
- $P$ , the *concurrency*, the number of MPI processes.

Precise operational meanings of these variables are appropriate. Since all of our solvers use PETSC SNES, we may define  $N$  as the result of applying `VecGetSize()` to the input `Vec` of the residual function. In simple cases on a grid or mesh with  $n$  nodes, and for a solution with values in  $\mathbb{R}^d$ ,  $N = nd$  is expected. However, more generally,  $N$  may have a complicated relationship to the mesh, for example, in higher-order FE discretizations on unstructured meshes. The concurrency  $P$  is set by the command “`mpiexec -n P`.” For the examples in this book it is also the result of calling `MPI_Comm_size()` on `PETSC_COMM_WORLD` (Chapter 1).

We now focus on two “consumable” quantities, regarded as functions of  $N$  and  $P$ , which we would want to minimize when designing a solver.

### Definition.

- The *flops*  $f(N, P)$  are the total number of floating-point operations counted during the run. It is the last number (`Total` column) returned by

```
mpiexec -n P ./program ... -log_view | grep "Flop: "
```

- The *run time*  $t(N, P)$  is the maximum wall clock time used by any of the processes. It is the first number (`Max` column) from

```
mpiexec -n P ./program ... -log_view | grep "Time (sec):"
```

The other numbers in the “Flop:” line of `-log_view` output are the *maximum flops* over the various ranks, the *imbalance ratio*, the *average flops*  $f(N, P)/P$ , and the total flops  $f(N, P)$ . Observe that we use plural “flops” as a number of operations, but other sources use the same word for what we call the *flops rate*  $f(N, P)/t(N, P)$ , or flops per second.

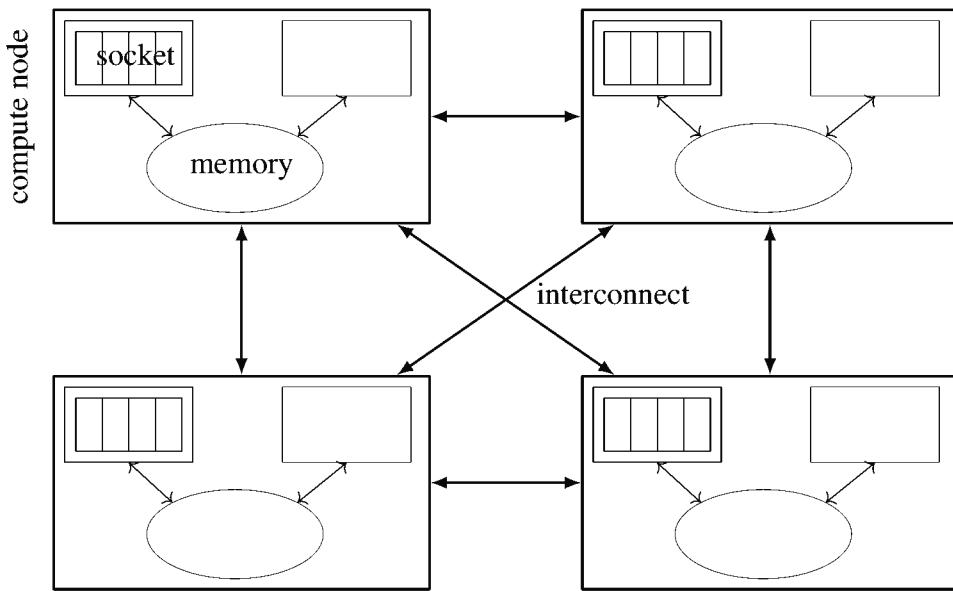
Given that all of the code examples in this book correspond to deterministic algorithms, the flops  $f(N, P)$  is a repeatable measurement. However, the run time  $t(N, P)$  should be understood as a random variable from a distribution. The spread of this distribution depends on the other activities on the machine, so  $t(N, P)$  might best be redefined as the expected value. In any case, it is natural to rerun timing measurements several times to get a sense of  $t(N, P)$  as an average or even as a confidence interval.

Before discussing other consumables, we must acknowledge the nontrivial architecture of supercomputers. Actually we will say “cluster” instead of “supercomputer”; a cluster is any machine of a certain architecture, whether small or big, and Figure 8.1 shows this architecture in simplified form. *Compute nodes*, typically blades in a rack with a common power supply and cooling system, communicate via an *interconnect*, the details of which are beyond our scope. Each compute node has a small number of CPU *sockets*,<sup>27</sup> each with a chip (die) containing multiple *cores*, the actual processors. In our simplified view, at most one MPI process can be running at any time on one core. (This view ignores the instruction-level parallelism of modern processors, e.g., pipelining, branch prediction, and hyperthreading [48].) The notional cluster in the figure can run at most  $P = 8$  processes simultaneously on each compute node. For such an intranode run, MPI messages are passed between processes via the shared memory on that node. At most  $P = 32$  processes can run simultaneously on the whole cluster, with messages between distinct nodes passing through the interconnect.

However, to acknowledge even more complications, the vague “memory” blobs in the figure actually consist of nontrivial levels of *cache* [48] and *main* memory, details of which will indeed affect the reader’s achievable performance, but which are also beyond our scope. (Modern architectures involve some degree of *nonuniform memory access* (NUMA) [48].) A key idea is

---

<sup>27</sup>Other processor sockets may exist on a node, such as graphics processing unit (GPU) sockets [10, 48].



**Figure 8.1.** A notional, simplified cluster with four compute nodes and an interconnect. Each compute node has two sockets holding 4-core processors and a shared memory.

that cores on a compute node compete for access to memory, and the latency and bandwidth of a memory access will depend on how “close” a core is to the memory being accessed. Competition with other memory requests accessing the same cache and memory channels complicates all realistic performance analyses.

Thus a parallel run on a cluster is subject to the following overlapping and nontrivial performance concerns:

- the latency of the interconnect,
- variable-speed access to memory depending on cache levels, and
- competition between cores for access to shared memory.

Furthermore, all of these issues relate to *process placement*, that is, to the user’s ability, via the installed MPI library and/or batch scheduler, to determine which processes are bound to which cores, sockets, or nodes. (See the section “Maximizing Memory Bandwidth” of the *PETSc Users Manual* [10] for further information.) For this chapter we often lump these important HPC issues into a nebulous source of uncertainty in run times.

Flops and run time are by no means the only consumables. For example, the watts consumed or, equivalently, the heat dissipated by the cluster’s cooling system, might be the important consumable for a conscientious modern user, but it is not easily measured with command-line tools. Certainly the maximum amount of memory which each process can allocate is a critical consumable, or at least it imposes a limit on capability. In fact the amount of memory available on each compute node is often the limiting factor which stops further solution improvement (further increases in problem size  $N$ ). Note that when a program issues a request too large for memory it will be killed by the operating system or become slow through disk swapping, so one should leave a margin between the total memory on a compute node and the amount needed by the run.

Memory usage of PETSC programs can be extracted from `-log_view` output, but the option `-memory_view` gives a convenient summary. A Unix utility, such as `top`, or a graphical system activity monitor, will show memory usage for each process and/or total memory usage in a

dynamic manner, but these tools require user attention or sampling, and they may not be easy to use on a cluster. The `valgrind` utility will also measure total memory (heap) usage, but it may not scale to large, parallel jobs. Because of these various complications, and though it would be natural to add total memory to our discussion of  $(N, P)$ -dependent consumables, it is impractical to do this precisely, and we forgo such analysis.

The number of *bytes transferred* (BT) to the processor from the fastest level of cache, and, more generally, the bytes transferred between levels of cache and memory, are important consumables. Measuring BT involves, for example, counting cache misses, a detail which is not included in `-log_view` output. The *arithmetic intensity* [48] of an algorithm is the ratio of flops over bytes transferred,  $f(N, P)/\text{BT}$ , and this quantity will help determine whether the algorithm's performance is limited by the cores' peak flops rate or by the memory bandwidth to those cores [155]. Arithmetic intensity is one of a three-quantity "spectrum" proposed by [33] for evaluating the performance of parallel PDE-solving algorithms; the others are the run time  $t(N, P)$  and the computational rate  $N/t(N, P)$ , namely the degrees of freedom per time.

As already mentioned, the capacity of the interconnect, both *latency* and *bandwidth*, is a consumable. Measuring this capacity is made complicated by different modes for passing MPI messages (above), and it is outside our scope.

Finally, regarding consumables generally see the "Profiling" and "Hints for performance tuning" sections of the *PETSc Users Manual* [10], including "Interpreting `-log_view` Output: Parallel Performance."

## The streams benchmark

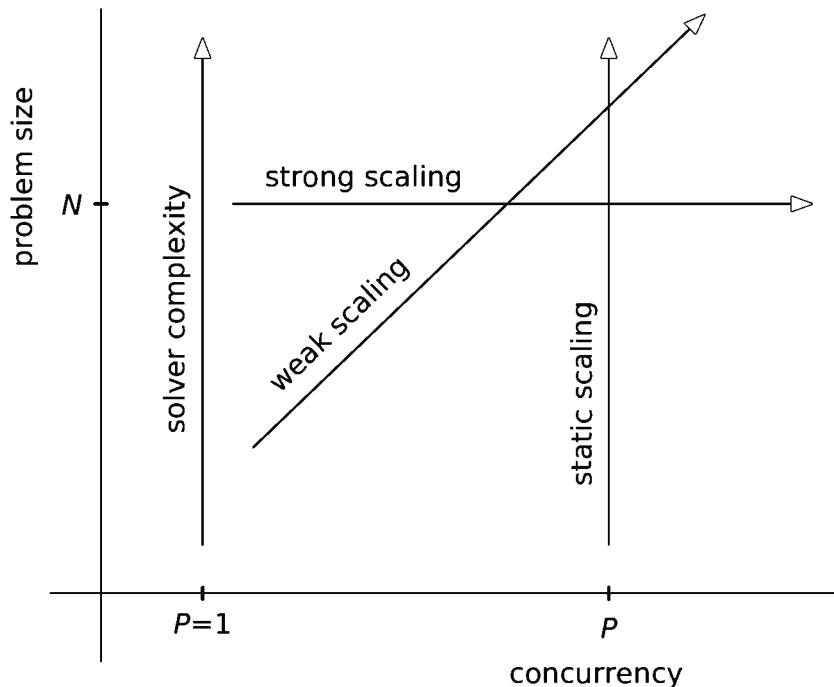
The `streams` benchmark [110] measures sustained bandwidth for transferring data from main memory to the CPU, while doing minimal arithmetic on this data. The reported rates, in bytes per second, are the speed at which the machine can add two relatively large vectors which fit into the main memory. Of course such jobs are part of any PDE solution, and a PETSc solver is often limited by memory-channel bandwidth rather than the flops rate of the available cores. The concurrency at which best `streams` performance occurs is often the best concurrency for solver applications. In particular, the reported `streams` bandwidth multiplied by the solution run time is an upper limit on bytes transferred (BT) for a solver.

PETSc makes it easy to run `streams`. On the author's laptop (2.2 GHz Intel Core i7-4770HQ) with four physical cores,

```
$ cd $PETSC_DIR
$ make streams NPMAX=4
Running streams with 'mpiexec' using 'NPMAX=4'
1 19582.3939 Rate (MB/s)
2 31048.5553 Rate (MB/s) 1.58553
3 26754.7595 Rate (MB/s) 1.36627
4 17390.6227 Rate (MB/s) 0.888074
```

When we go from  $P = 1$  to  $P = 2$  the bandwidth increases by less than 60%, but increasing the concurrency further actually reduces the rate. While plenty of parallel algorithm development can (and does) occur on this machine, production runs with  $P > 2$  are not worthwhile.

Compared to a laptop, clusters are built with CPUs designed for higher concurrency, e.g., Intel Xeon instead of Core i7. Also, deliberate process placement can improve performance further on clusters. The user can, and often should, ask `mpiexec` or the batch system to either bind processes to cores or impose limits on the number of cores per compute node (and/or socket), so as to get better memory bandwidth.



**Figure 8.2.** Strong, weak, and static scaling, as well as serial solver complexity, are directions to infinity in the  $(N, P)$  plane.

For example, suppose we run `streams` on three compute nodes of a cluster,<sup>28</sup> each with two 12-core Intel Xeon processors (2.6 GHz E5-2685 v3), but imposing a limit of four MPI processes on each node:

```
$ make streams NPMAX=12
Running streams with 'mpiexec' using 'NPMAX=12'
 1 14209.5910    Rate (MB/s)
 2 26424.4449    Rate (MB/s) 1.85962
 3 38519.8476    Rate (MB/s) 2.71083
...
11 141210.8168   Rate (MB/s) 9.93771
12 154423.6243   Rate (MB/s) 10.8676
```

(See Exercise 8.1 and the SLURM option `-tasks-per-node`.) That is, with  $P = 12$  processes there is an almost 11 times increase in `streams` bandwidth. This is a configuration for which it is worthwhile to develop a parallel PDE solver and ask how it scales with  $P$ .

## The classic language of speedup

Each computation is a point in the  $(N, P)$  plane of Figure 8.2. On the other hand, we regard a scaling analysis as the asymptotics of the consumables defined above, as  $N \rightarrow \infty$ ,  $P \rightarrow \infty$ , or both—hence the arrows in the figure. One such analysis, *strong scaling*, measures performance for fixed problem size as the concurrency increases, that is, it supposes a sequence of computations with increasing  $P$  and fixed  $N$ .

<sup>28</sup>Chinook, at the Geophysical Institute, University of Alaska Fairbanks. See [www.gi.alaska.edu/services/research-computing-systems](http://www.gi.alaska.edu/services/research-computing-systems).

**Definition.** [47, 74] Fix the problem size  $N$ .

- The parallel *speedup* is relative to the run time using one process:

$$s_N(P) = \frac{t(N, 1)}{t(N, P)}.$$

- The parallel *efficiency* is the actual speedup relative to the desired speedup:

$$e_N(P) = \frac{s_N(P)}{P} = \frac{t(N, 1)}{P t(N, P)}.$$

The expected ranges are  $1 \leq s_N(P) \leq P$  and  $0 \leq e_N(P) \leq 1$ . That is, it is natural to assume that a job of fixed size divided among  $P$  workers can, at best, be done in  $(1/P)$ th the time. However,  $s_N(P) > P$  can occur, and the explanation is that there will be some first value of  $P$  such that the  $N/P$  unknowns on each process can fit into fast cache memory (of some level). At that concurrency the time  $t(N, P)$  drops more than it otherwise would, so  $s_N(P)$  exceeds  $P$  if the speedup is already good. The fact that  $s_N(P) > P$ , “superlinear speedup,” is even possible reminds us that  $t(N, P)$  includes the time for data to transit from memory to the processor, not just for arithmetic. In this regard flops are simpler than run time. For the algorithms in this book, if  $N$  is fixed then the flops  $f(N, P)$  increase monotonically with  $P$ . In particular, our parallel algorithms do at least as much arithmetic as their serial counterparts.

A fundamental assumption in the classic language of speedup is that the algorithm is the same for different  $P$ , since  $N$  is fixed, but this assumption is violated for most PDE solvers. For example, it fails for the basic CG+GMG solver for the Poisson equation (Chapters 6 and 7):

```
mpiexec -n P ./fish -da_refine LEV -pc_type mg
```

The serial smoother here is symmetric Gauss-Seidel (GS), which changes to processor-block GS for  $P > 1$ , so as  $P$  increases the smoother uses fewer off-diagonal elements. The resulting loss of smoothing efficiency can increase the KSP iteration and flops counts.

Despite such violations, parallel speedup and efficiency are worth considering. A solver is regarded as strong scaling if the speedup is close to  $P$  or, equivalently, if the efficiency is close to one. For example, if the study reveals that there is a small  $\delta > 0$  so that  $e_N(P) \geq 1 - \delta$  for large  $P$  then the solver has good strong scaling.

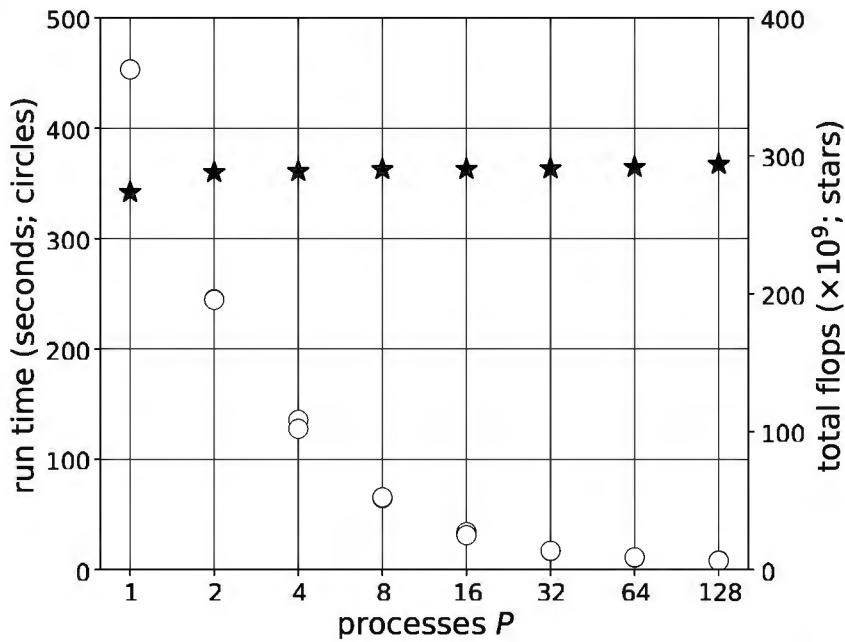
A key idea for parallel PDE solutions is that the number of degrees of freedom per process  $N/P$  must remain large enough so that the processes are doing significant work. For instance,  $N/P \geq 10^5$  might be needed for reasonable results. In fact, the serially optimal solvers in this book have good strong-scaling performance when  $N/P \geq 10^5$ .

**Example.** Consider `minimal.c` (Chapter 7) which solves the nonlinear minimal surface equation for the shape of a soap film. Fix a  $4097 \times 4097$  grid with  $N = 1.6 \times 10^7$  degrees of freedom. Our solver combines Newton iteration, a finite-differenced Jacobian, GMRES, V-cycle GMG, and grid sequencing starting with an initial solution on a  $33 \times 33$  grid, namely

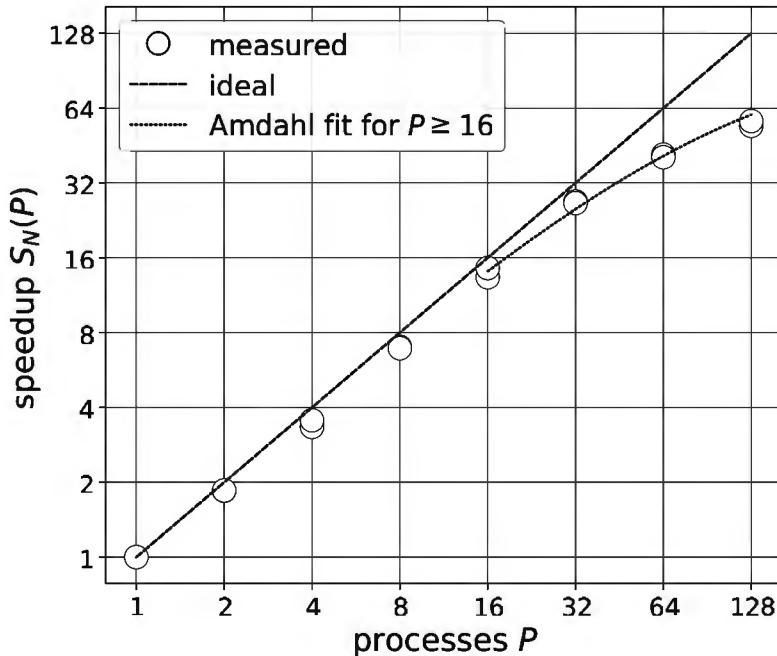
```
$ mpiexec -n P ./minimal -da_grid_x 33 -da_grid_y 33 \
-snes_fd_color -snes_grid_sequence 7 -pc_type mg
```

for  $P = 1, 2, 4, \dots, 128$ . We add monitoring options (e.g., `-log_view`) and do each run twice to observe variability in run time. The raw timing and flops data, shown in Figure 8.3, reveal a fundamental success: the  $P = 1$  job takes 453 seconds, the  $P = 128$  job takes about 8 seconds, and parallel computation is effective!

For this solver the total flops  $f(N, P)$  increase only slightly as  $P$  increases (Figure 8.3) because the algorithmic changes are small, so strong scaling is a reasonable goal. Note that the



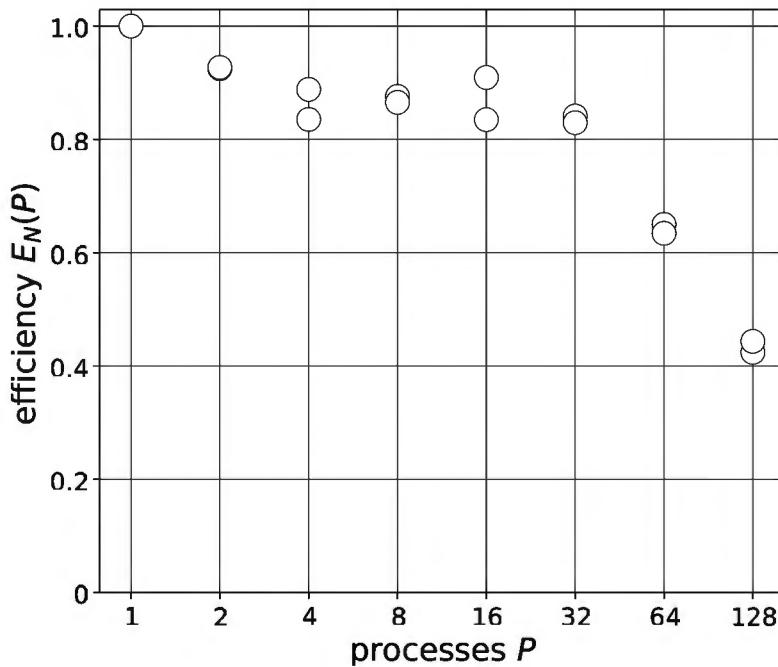
**Figure 8.3.** Run times  $t(N, P)$  and flops  $f(N, P)$  for a minimal surface equation (see Chapter 7) solution with  $N = 1.6 \times 10^7$ .



**Figure 8.4.** For the same runs as in Figure 8.3, speed-up is close to ideal up to  $P = 32$ .

job must fit in memory for  $P = 1$ ; otherwise,  $t(N, 1)$  and  $S_N(P)$  cannot be evaluated. Also,  $N/P = 1.3 \times 10^5$  when  $P = 128$ ; it would make little sense to run this job with  $P = 1000$ .

A typical strong-scaling analysis for these runs might display the results as in Figures 8.4 and 8.5. The observed speedup is close to ideal (the dashed line) up to roughly  $P = 32$ . However, efficiency shows noticeable drops at  $P = 2$  and  $P = 4$ . These drops are explained by memory contention issues. That is, we have used at most 32 compute nodes, limiting the processes per node to four (for streams benchmark reasons; see above), but the  $P = 1$  and  $P = 2$  jobs live



**Figure 8.5.** Plotting efficiency shows more detail, including drops at low concurrency and at  $P = 64$  (with different explanations; see text).

on a single node and thus they are a little faster. The efficiency relative to the  $P = 4$  case is close to the ideal up to  $P = 32$ .

But what speedup is even possible for a nontrivial solver? A classic theoretical answer is *Amdahl's law* [6, 48], a simple performance model as follows. Suppose the algorithm has serial run time  $t(N, 1)$  and that a certain fraction  $0 \leq f_s \leq 1$  of this time is spent in code that is not parallelizable. The remainder of the code is assumed to have perfect speedup, so Amdahl models the run time as

$$t(N, P) = f_s t(N, 1) + (1 - f_s) \frac{t(N, 1)}{P}. \quad (8.1)$$

In terms of the speedup  $S_N(P)$  there is then a limiting upper bound (Exercise 8.3),

$$S_N(P) = \frac{P}{f_s(P-1) + 1} \rightarrow \frac{1}{f_s} \quad \text{as } P \rightarrow \infty.$$

(Efficiency goes to zero in the same limit.) In other words, Amdahl's law is a model for “diminishing returns” at large concurrency.

**Example.** Consider an abstract computation governed by Amdahl's law. Supposing  $f_s = 0.1$ ,  $P = 10$  processors yields  $S_N(10) = 5.2$ . While this is useful speedup, higher concurrency is not worthwhile:  $P = 100$  and  $P = 1000$  yield speedups of 9.2, 9.9, respectively, and  $S_N(P) < 10$  for all  $P$ .

An appreciable amount of nonparallelizable code surely exists in any PETSc program. For example, operations such as opening files occur only on rank 0. More significantly, many operations, such as handling options or computing the Chebyshev polynomial coefficients in a smoother (Chapter 2), are both collective and identical across all ranks; they do not become faster with more processes, and thus they are serial in the sense of Amdahl.

The time used for communication between processes, such as in a global reduction to compute a norm, or the all-to-all communication in solving the coarse grid problem in multigrid, can

be regarded as another “nonparallelizable” part of the algorithm; the time does not decrease to zero as  $P \rightarrow \infty$ . Alternatively, the average waiting time for these synchronization events could be modeled by adding a (possibly)  $N$ - and  $P$ -independent communication time  $t_c$  [48, section 2.3.3.1] to Amdahl’s law (Exercise 8.5).

Identifying all nonparallelizable portions of a nontrivial code would be an unpleasant task. However, we may fit the Amdahl model to timing data. Doing so with the  $P \geq 16$  portion of the above data yields the dotted curve in Figure 8.4; this data supports a serial fraction  $f_s = 0.0089$  (Exercise 8.4). Small- $P$  values are excluded from the fit because such a 1% serial fraction will be undetectable given the noise in the run times. Note that 1% of  $N$ , in this computation, is about  $10^5$ . Thus Amdahl’s law substantially explains the folk wisdom that each process should be given an adequate amount of work.

**Fact 14.** Strong scaling requires that each process be kept busy on a problem of substantial size. *For a parallel PDE solver with  $N$  total degrees of freedom shared over  $P$  processes, something like  $N/P > 10^5$  is suggested.*

A precise fit to Amdahl’s law would require an algorithm that was invariant with  $P$ , but that would limit us to suboptimal PDE solvers (Exercises 8.2 and 8.4). In this context, *Gustafson’s law* [74, 48] attempts to address this shortcoming. However, instead of fiddling with strong-scaling models we prefer to reconsider directions in the  $(N, P)$  plane (Figure 8.2).

## Weak and static scaling

Problem size  $N$  is *not* fixed when numerically solving a PDE problem. Indeed, such solutions want  $N \rightarrow \infty$ , and a user’s perception of practical values for  $N$  will likely grow to fill the available HPC resources. While large problem sizes  $N$  are limited by available memory, overall resource and budget constraints tend to limit the number of compute nodes and  $P$ . User patience, budgets, and batch-system constraints all limit the allowed run time  $t(N, P)$ .

One might make a parable for this situation:

Pharaoh has  $10^4$  workers and an unbounded supply of limestone blocks, but only ten years to live. To glorify the dead and intimidate the living, Pharaoh asks the chief engineer, “How big a pyramid can my workers build in 10 years?”

Asking the chief engineer the strong-scaling question “how fast can  $P$  workers build this particular pyramid design using  $N$  blocks?” is less natural. Thus we define new scaling modes.

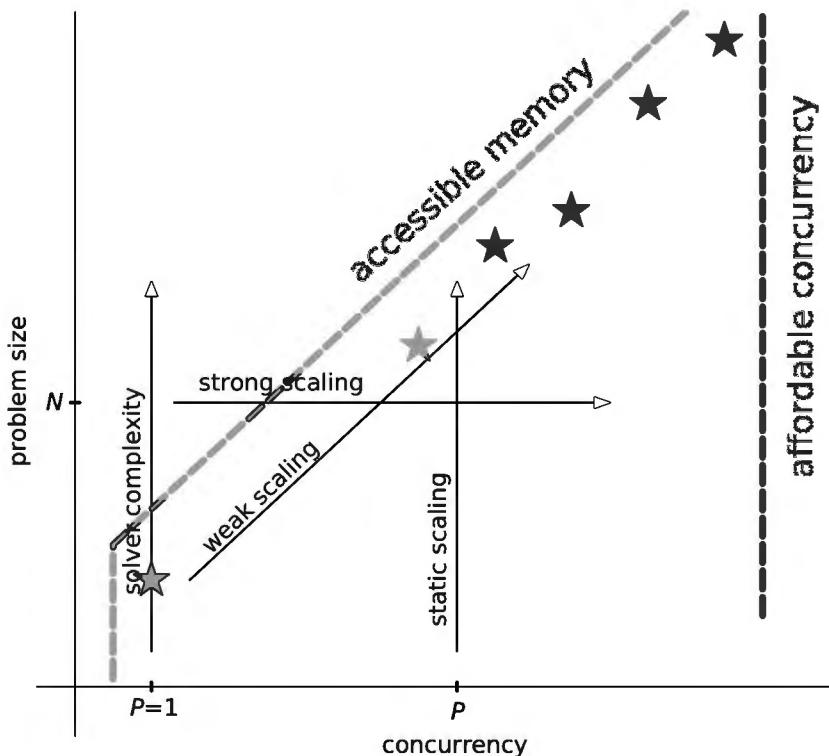
**Definition.** Consider a study of consumables over a sequence of runs  $\{(N_i, P_i)\}$ .

- A *static-scaling* [33] study fixes  $P_i = P$  and increases  $N_i$ .
- A *weak-scaling* (e.g., [48]) study fixes the degrees of freedom per process,

$$\alpha = \frac{N_i}{P_i} \quad \text{is fixed,}$$

and increases both  $N_i$  and  $P_i$ .

Static-scaling studies are especially relevant when the user is up against a hard concurrency limit with an algorithm which is not yet using all available memory. Then one has the same solver-complexity goal as in serial: optimality. That is, one wants  $t(N, P) = O(N)$  as  $N \rightarrow \infty$ . On the other hand, the coefficient in “ $O(N)$ ” definitely matters. For example, a solver could satisfy this relationship, notionally “good static scaling at  $P$ ,” with a coefficient which does not



**Figure 8.6.** Continuing from Figure 8.2, we observe that practical and achievable computations (stars) must respect cluster limitations.

decrease in  $P$ , a parallelization disaster. Such considerations motivate both weak scaling and the “ideal” scaling discussed in the next section.

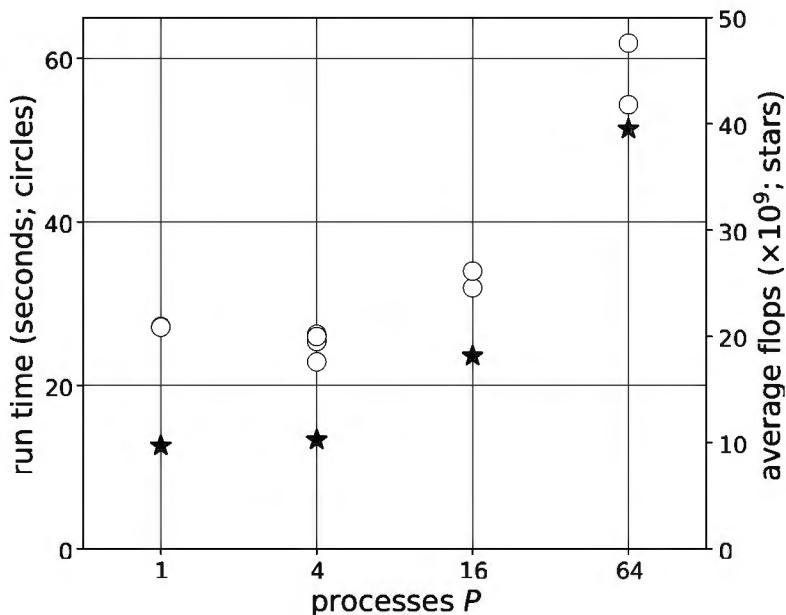
When solving a PDE on a domain  $\Omega \subset \mathbb{R}^d$ , a weak-scaling study increases the number of grid/mesh points simultaneously with the concurrency. As  $P$  increases, each process sees a smaller fraction of  $\Omega$  but an unchanging number of grid points. For structured grids, where it is convenient to refine grids by (linear) factors of two, it is common to have  $N_{i+1} = 2^d N_i$ .

One may give a practical argument for weak-scaling studies as follows: A user solving PDEs in parallel, not even conscious of scaling concerns, might run a sequence of computations like the stars in Figure 8.6, first gaining confidence in serial solver complexity and then debugging parallel runs at low concurrency. Then the desire for performance—to meet a predetermined accuracy goal or to impress colleagues—drives up both  $N$  and  $P$ . Concurrency might hit a limit first, either from the actual available number of processors or because the batch scheduler causes large- $P$  jobs to wait on a queue. However, as problem sizes get bigger the user also becomes aware of the limited memory on each node. Perhaps an irritating out-of-memory error leads to more-deliberate choices for  $N/P$ ; now runs might be designed around  $N/P \approx 10^6$ , for example. In other words, practical limits constrain the abstract  $(N, P)$  plane, leading to  $N/P \sim \alpha$  habits as more compute nodes become available (Figure 8.6).

**Example.** How good is the weak-scaling performance of our preferred ch7/minimal.c solver? Consider these runs, essentially the same as those for the strong-scaling study earlier:

```
$ mpicmd -n P ./minimal -da_grid_x 33 -da_grid_y 33 \
-snes_fd_color -pc_type mg -snes_grid_sequence LEV
```

We set  $P = 1, 4, 16, 64$  and  $LEV = 5, 6, 7, 8$ , respectively, so each process owns a  $1024 \times 1024$  grid and  $N/P \approx 10^6$ . At least two runs are done for each value of  $P$ . At  $P = 64$  we are solving on a  $8193 \times 8193$  grid with  $N = 6.7 \times 10^7$  degrees of freedom.



**Figure 8.7.** Run time  $t(N, P)$  and average flops  $f(N, P)/P$  in a weak-scaling study of `ch7/minimal.c`.

Figure 8.7 shows the result. The run time increases by a factor of only about 2 as  $N$  and  $P$  increase by a factor of 64, though we might like to see better weak scaling. It is not clear if  $t(N, P)$  will remain bounded as the job gets bigger along this  $N/P = \alpha$  path. A partial explanation of imperfect weak scaling is already clear in average flops  $f(N, P)/P$ , shown in the same figure. The amount of work per process is going up even though  $N/P$  is fixed. Namely, the number of linear iterations on the finest grid is increasing. To be more precise, each run ends on a finest grid on which the algorithm does a certain number of Newton iterations, each with inner KSP iterations. Under grid sequencing, only two Newton iterations occur for  $P = 1, 4, 16$  and just one for  $P = 64$ , so the nonlinear Newton iteration is not struggling. Instead, the total number of GMG-preconditioned GMRES iterations on the finest grid grows with  $N$ ; the values are 14, 18, 43, 104 for  $P = 1, 4, 16, 64$ .

Why are the iterations increasing? The reason must be that the preconditioned Jacobian  $M^{-1}J$  has a difficult spectrum, in the sense that polynomials which are small on  $\sigma(M^{-1}J)$  have high degree; see Chapters 2 and 6. (To confirm this understanding we can measure the condition numbers with `-ksp_view_singularvalues`.) Fundamentally, only spectral reasons cause increasing GMRES iterations, but different effects might spread the spectrum of  $M^{-1}J$ :

- As described in Chapter 7, the catenoid solution has large derivatives, especially near a corner of the domain. At high concurrency certain processes are seeing a harder problem, and (for instance) processor-block SSOR might struggle to resolve them. Determining the spectrum of the local (diagonal block) matrices would confirm this hypothesis.
- Perhaps the `-snes_fd_color` Jacobian is spreading the spectrum at high resolution. One could test this hypothesis by implementing an exact Jacobian or switching to a software framework where a symbolic Jacobian is easily available (Chapters 13 and 14).

One might instead try to transcend the Newton-Krylov paradigm. For example, recent work by PETSc developers [29] describes how to modify a PETSc code to use full-approximation-scheme multigrid with a nonlinear smoother and/or a nonlinear Richardson iteration, approaches which are beyond our scope. In any case our weak-scaling difficulties seem to originate in the mathematics of the PDE itself, and so we stop here.

Examples like this for nonlinear PDE problems generally suggest that weak scaling is more challenging than strong scaling because increasing problem size invokes increasing detail in the PDE itself.

## Toward ideal scaling

We inevitably ask parallel PDE solvers for the largest accessible problem sizes ( $N$ ), and thus we are interested in their  $N \rightarrow \infty$  behavior. Our demand for large  $N$  occurs whether the concurrency ( $P$ ) increases along with  $N$  (weak scaling) or is fixed at a value we can afford (static scaling). Such considerations motivate the following simple theory.

**Definition.** Suppose  $t(N, P)$  is the solver run time for  $N$  degrees of freedom on  $P$  processes. For fixed  $\alpha > 0$ , consider a sequence of computations with  $N/P = \alpha$  but  $N, P \rightarrow \infty$ .

- A solver is *weak bounded* for these computations, with bound  $\tau = \tau(\alpha)$ , if

$$t(N, P) \leq \tau.$$

- A solver has *weak efficiency* for these computations, with bound  $\epsilon = \epsilon(\alpha) > 0$ , if

$$E_N(P) = \frac{t(N, 1)}{Pt(N, P)} \geq \epsilon.$$

Claims in the literature that a code has good weak scaling should generally be interpreted as claims of weak boundedness (e.g., [28, 126]).

### Lemma.

- (i) A weak-bounded solver which is not magic in serial, i.e., for which there exists  $c > 0$  such that  $t(N, 1) \geq cN$ , has weak efficiency.
- (ii) A serially optimal solver with weak efficiency is weak bounded.
- (iii) A serially optimal solver with weak efficiency is optimal in static scaling, and the coefficient is  $O(P^{-1})$ . That is,  $t(N, P) = O(N/P)$ .

**Proof.** These are straightforward inequalities. For (i) let  $\epsilon = c\alpha/\tau > 0$  and note that

$$\frac{t(N, 1)}{Pt(N, P)} \geq \frac{cN}{P\tau} = \epsilon.$$

For (ii), by the definition of serial optimality (Chapter 7), namely  $t(N, 1) \leq CN$ ,

$$t(N, P) = \frac{Pt(N, P)}{t(N, 1)} \frac{t(N, 1)}{P} \leq \frac{1}{\epsilon} \frac{CN}{P} = \frac{C\alpha}{\epsilon},$$

so let  $\tau = C\alpha/\epsilon$ . Finally, for (iii) the proof of (ii) already shows  $t(N, P) = O(N/P)$ .

This theory allows one to move around the  $(N, P)$  plane, using serial optimality and weak scaling to connect different points in the plane. For example, a demonstration of both serial optimality and weak boundedness, for a significant range of  $N$  and  $P$  values, represents a high standard for scaling because weak efficiency and static scaling optimality both follow. Therefore, in later chapters we will demonstrate serial optimality and weak boundedness when possible.

However, one may look at scaling in a more holistic way. The best solver is highly parallelizable, e.g., with  $f_s \approx 0$  in the Amdahl's law sense, *and* weak scaling (weak bounded and/or efficient), *and* optimal at each  $P$ . That is, we want our solvers to have optimal complexity in static-scaling studies, but with a constant which depends on  $P$  in the right way.

**Definition.** A solver has *ideal scaling* if

$$t(N, P) \sim \frac{N}{P} \quad (8.2)$$

in a region of the  $(N, P)$  plane defined by bounded-below degrees of freedom per process.

This definition means that there exist constants  $C_2 \geq C_1 > 0$ , and  $c > 0$ , so that  $C_1 N/P \leq t(N, P) \leq C_2 N/P$  for all  $P \geq 1$  and  $N$  such that  $N/P \geq c$ . As with serial optimality (Chapter 7), this definition assumes infinite memory because  $N \rightarrow \infty$ .

An advantage of ideal scaling as a development and testing goal is that, instead of doing runs limited to a particular  $N/P$  value, as in a weak-scaling study, or limited to a fixed  $P$  in a strong-scaling study, one may simply do some computations for various points  $(N_i, P_i)$  and measure the degree to which they do not satisfy the ideal  $t(N, P) = \gamma N/P$ . To make such an analysis quantitative we use a new term which means “inability to scale.”

**Definition.** Choose a reference serial computation with  $N_0$  degrees of freedom. For a collection of computations  $(N_i, P_i)_{i=1}^k$  the (observed) *dyscalia* is the dimensionless number

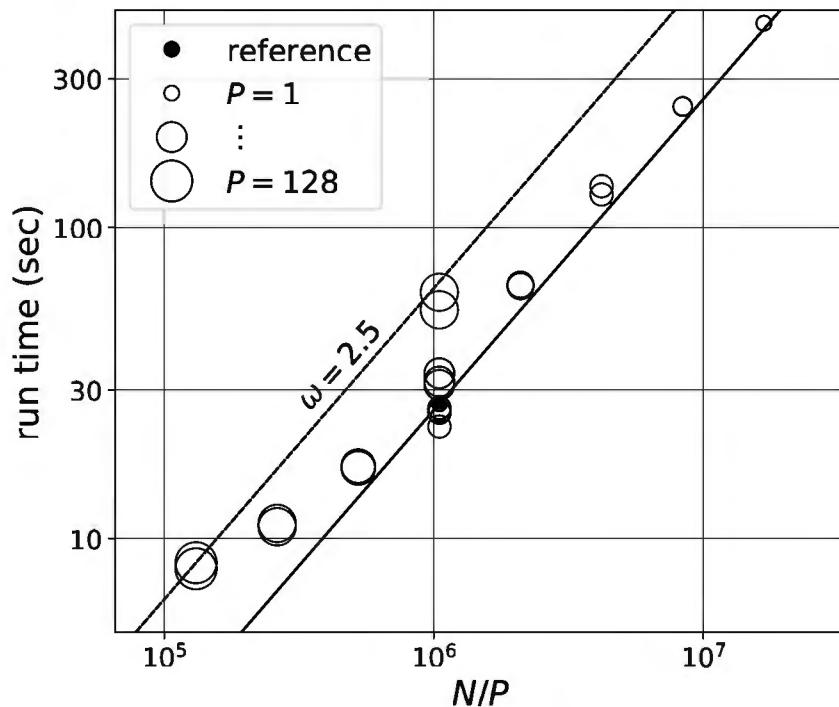
$$\omega = \max_i \left\{ \frac{t(N_i, P_i) P_i N_0}{t(N_0, 1) N_i} \right\}. \quad (8.3)$$

That is,  $\omega$  is the worst observed value of the obvious ratio when ideal scaling (8.2) is the goal, namely  $t(N_i, P_i)/(N_i/P_i)$ , relative to its serial reference value, namely  $t(N_0, 1)/N_0$ . Said a different way,  $\omega$  is the worst *total* processor time per degree of freedom ( $P_i t(N_i, P_i)/N_i$ ) relative to the processor time per degree of freedom in the reference case ( $t(N_0, 1)/N_0$ ). In any case note that  $\omega \geq 1$  and that adding more data generally increases  $\omega$ ; it is a measure of *observed* performance only.

The ideal solver satisfying  $t(N, P) = \gamma N/P$  exactly would have all run times on a line with slope  $\gamma$  in the time-versus- $N/P$  plane. Plotting time versus  $N/P$  on log-log axes would give a line with slope one for this ideal. On such a plot the points (computations) live in a band above the line of slope one which goes through the reference computation, and dyscalia  $\omega$  measures the size of this band.

For example, using our preferred solver of the minimal surface equation (Chapter 7), we collect the previous strong- and weak-scaling runs into Figure 8.8. The band shown in this figure has dyscalia  $\omega = 2.5$ . The figure also shows concurrency  $P$  by marker size. An advantage of this kind of plot is that it shows actual run times.

There are limits in how far the band in such a plot can extend to the left and right. Amdahl's law can be interpreted as a “ $N/P \geq 10^5$  for good performance” bound, for example, a limit on



**Figure 8.8.** A holistic scaling view of scaling for `ch7/minimal.c`:  $t(N, P)$  versus  $N/P$  on log-log axes. The observed dyscalia  $\omega = 2.5$ , from (8.3), measures the failure of ideal scaling (8.2).

the left side. On the right side the actual amount of memory per compute node will impose an upper bound on  $N/P$ . When showing results such as in Figure 8.8 it is reasonable to declare a range  $A \leq N/P \leq B$  in advance.

## Caveats

No matter how we display parallel scaling, whether in strong-scaling plots like Figures 8.4 and 8.5, a weak-scaling plot like Figure 8.7, or using the new style of Figure 8.8, note we have been measuring run time of the *whole* code. We might instead time the PCSetup and KSPSolve events separately, for example. Indeed, if we want good overall parallel scaling then both set-up and solve events need to scale. In particular, PCSetup needs to be parallel. If this event is part of the serial fraction then weak scaling is impossible. For example, preconditioned Krylov iterations generally involve assembling a sparse matrix, and this must be done in parallel.

Fact 15. Parallel efficiency requires assembling matrices using the same distribution as the solver [134]. *The vast majority of matrix entries should be generated on the process where they will be used most. Do not expect much benefit from setting up a big system elsewhere and then reading it into PETSc to “solve it in parallel.”*

The DMDA object makes PCSetup weak-scalable without user attention. In fact, the parallel examples in this book either use DMDA structured grids or call Firedrake (Chapter 13) for parallel assembly via DMFlex. DM objects are designed, to, among other things, facilitate scalable PCSetup stages [109]. (Compare the serial-only unstructured-grid code in Chapter 10.)

On the other hand, because weak scaling can be “gamed,” it is important to measure and analyze performance of serial runs and justify how flops are being used in serial cases. This is a prerequisite before considering parallel scaling.

**Fact 16.** The easiest way to make software scalable is to make it sequentially inefficient [73]. *Both deliberate and accidental attempts to “game” weak scaling come down to wasting time on each process, relative to the performance of the best solution method in serial.*

Finally, the  $N, P$  scaling considerations above, for evaluating the performance of PDE solvers by examining flops and run time, are subject to the criticism that the accuracy at which the PDE is solved does not enter into these performance measurements. Furthermore, changing to a different discretization could yield much smaller numerical error for the same problem size  $N$ . (The h/p finite element method demonstration in Chapter 13 is an example.) With such concerns in mind, Chang and others [32] propose a time-accuracy-size performance “spectrum,” modifying a static-scaling analysis by incorporating the numerical error.

## Parallel multigrid with PCTelescope

Parallel-distributed DMDA structured grids have a restriction: each process must own at least one degree of freedom. On the other hand, multigrid is essential for optimality (Chapters 6), so this restriction hurts weak and/or ideal scaling because spectral effectiveness in multigrid requires coarse grids with few points, potentially fewer than the concurrency  $P$ . For example, the coarsest grid is chosen to be large ( $33 \times 33$ ) in the runs in the current chapter to allow the coarse-grid solve to occur in parallel with  $P$  in the hundreds. As explained in Chapter 7, this solve is redundant across all processes and managed by the **redundant** PC type.

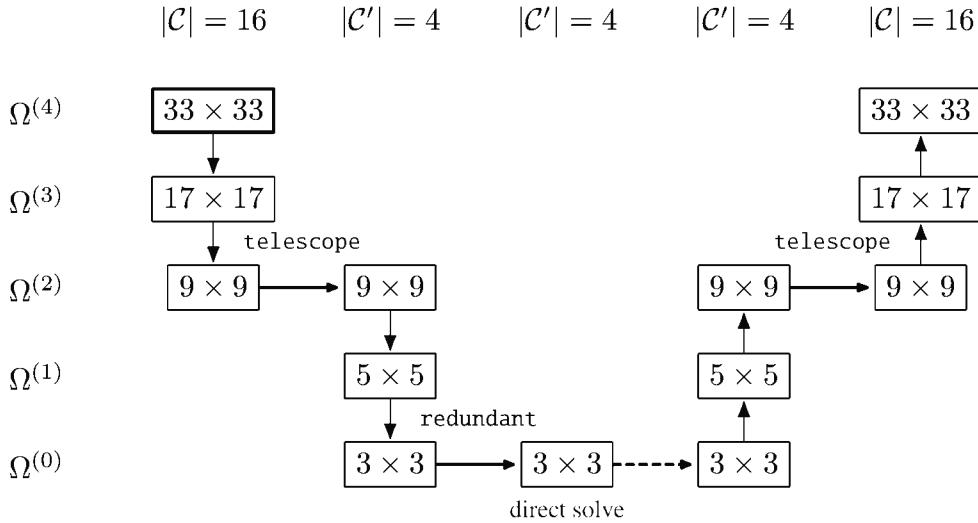
The **telescope** PC type [109] is designed to resolve this practical DMDA restriction and, more generally, to give better parallel GMG performance. It allows GMG to have deeper, more spectrally effective V-cycles while still using high concurrency (large  $P$ ) on the finer grids. It generalizes the **redundant** type by distributing and solving the coarser grid problems over a subset of processes, while in **redundant** the coarse-grid problem is solved on each process.

To describe **telescope** further we assume that our GMG solver has  $L$  levels and a fine grid  $\Omega^{(L)}$ —see notation in Chapter 6—distributed over an MPI communicator  $\mathcal{C}$  of large size  $P = |\mathcal{C}|$ . For solver efficiency we want a small coarsest grid with  $|\Omega^{(0)}| < P$  points. The **telescope** type uses “agglomeration” [144, subsection 6.3.2] to transfer the coarser-grid problems to fewer processors, i.e., to a smaller MPI communicator  $\mathcal{C}'$ .

Understanding command-line **telescope** usage requires slightly more detail. The user must choose a reduction factor  $r > 1$  and an intermediate level  $0 < L' < L$  at which the transfer will occur. At the setup stage **telescope** generates an MPI communicator with  $|\mathcal{C}'| = |\mathcal{C}|/r$  processes, presumably so that  $1 \leq |\mathcal{C}'| < P$ , and then, during each multigrid cycle, at the  $L'$  level **telescope** transfers (VecScatters) the problem from  $\mathcal{C}$  to  $\mathcal{C}'$  and back. In fact, starting from  $\Omega^{(L)}$ , a V-cycle descends  $k_1$  levels on  $\mathcal{C}$ , with (down-)smoothing and restriction, to an intermediate grid  $\Omega^{(L-k_1+1)}$ . On this intermediate grid, **telescope** scatters to  $\mathcal{C}'$ . Then we descend through  $k_2 = L - k_1 + 2$  additional levels to the coarsest grid  $\Omega^{(0)}$ . At  $\Omega^{(0)}$  the **redundant** PC, and usually a direct solver, solves the coarsest-grid problem over  $\mathcal{C}'$ . To complete the V-cycle we go up  $k_2$  levels on  $\mathcal{C}'$  to the intermediate grid, scatter back, and then go up  $k_1$  levels on  $\mathcal{C}$ . The user is in charge of arranging that  $\Omega^{(0)}$  can be partitioned on  $\mathcal{C}'$ , i.e.,  $|\Omega^{(0)}| \geq |\mathcal{C}'|$  is still required for DMDA. Because one counts the levels including the finest and coarsest,  $k_1 + k_2 = L + 2$ . Consider also adding these options for monitoring and cycle visualization:

```
-ksp_monitor_short -mg_{levels,coarse}_ksp_converged_reason \
-mg_coarse_telescope_mg_{levels,coarse}_ksp_converged_reason
```

The **-ksp\_view** of the solver is also worth seeing.



**Figure 8.9.** By scattering (horizontal arrows) to a smaller communicator at some intermediate level, telescope allows parallel V-cycles to use a coarse grid  $\Omega^{(0)}$ .

For example, as illustrated in Figure 8.9, the following run has  $L = 4$ ,  $P = |\mathcal{C}| = 16$ ,  $r = 4$ ,  $k_1 = 3$ , and  $k_2 = 3$ :

```
$ mpiexec -n 16 ./fish -da_refine 4 -pc_type mg -pc_mg_levels 3 \
    -mg_coarse_pc_type telescope -mg_coarse_pc_telescope_reduction_factor 4 \
    -mg_coarse_telescope_pc_type mg -mg_coarse_telescope_pc_mg_levels 3
```

See Exercise 8.7.

The native algebraic multigrid (AMG; Chapter 10) solver in PETSc, namely `-pc_type gamg`, is more flexible regarding parallel distribution of coarse grids than is GMG over a DMDA structured grid. (Likewise, `DMPlex` is more flexible; see Chapter 13.) The `gamg` type permits a process to own zero degrees of freedom, and therefore `telescope` may not be as necessary. Note that one may also use `telescope` to switch from GMG to AMG for the coarser grids. Furthermore, one may use `telescope` in other ways than a single reduction at a mid-level [109], and one may also switch to Galerkin coarse grid operators (6.21) using `-mg_coarse_telescope_pc_mg_galerkin`.

## Parallel nondeterminacy

A final, and occasionally important, observation about parallel computations arises from the fact that floating-point arithmetic is not exactly associative [63]. Furthermore, the total order in a parallel sum, such as in the inner product reduction operations in CG and other Krylov methods (Chapter 2), is not predetermined [121]. For example, the MPI standard [72] allows terms of a `MPI_Reduce()` sum to be accumulated on the rank 0 process as they arrive.

Consider a slowly converging sum (Exercise 8.8). Suppose there are many processes computing parts of the sum, and suppose that there is significant “noise” in the interconnect in the sense that the order in which processes report their partial sums is not predictable. Then the bits of the resulting sum are not exactly repeatable. Though *backward-stable* algorithms [143], including sums and inner products, will not suffer large output changes, it is a fact of life that sufficiently complicated parallel solvers are (bitwise) nondeterministic. This is one justification for using `-snes_monitor_short` and/or `-ksp_monitor_short` when doing regression testing on codes, for example.

Fact 17. Parallel reductions are nondeterministic at the bit level. *Because floating-point arithmetic is not exactly associative, different orders of arrival during reductions will affect results.*

---

## Exercises

- 8.1. (a) Modify the script `ch8/cluster.sh` to make it run on the batch system on your hardware. Consider starting with  $P \leq 20$ , for example, and look for interactive queues/partitions suitable for parallel development. See your system documentation or seek assistance from your system administrator.
- (b) What are the `streams` rates? Can you improve bandwidth by adjusting batch settings?
- (c) Choose one the several PDE solutions illustrated in the script. Increase the resolution and concurrency up to limits imposed by your hardware and/or account limits. Note that long run times are *undesirable*; solve the PDE near the highest achievable resolution while imposing a limit of  $t(N, P) \leq 15$  minutes or similar.
- (d) With this solver, redo the strong-, weak-, and ideal-scaling analyses shown in this chapter.

- 8.2. Consider this `ch6/fish.c` run using GMG:

```
| $ mpiexec -n P ./fish -fsh_dim 3 -da_refine 6 -pc_type mg
```

In what ways does the algorithm change when going from serial to parallel? By reviewing the discussion of GMG in Chapters 6 and 7, choose a smoother that will remove much (or all) of the  $P$  dependence. What is the coarse-grid solver like? What will change or not change with  $P$ , and what will be limited to a range of  $P$ ? Choose a coarse-grid solver which is nearly independent of  $P$  (for a modest range of  $P$ ) and measure strong-scaling performance.

- 8.3. Starting from Amdahl's law in (8.1), confirm that  $S_N(P) \leq 1/f_s$ ,  $S_N(P) \rightarrow 1/f_s$  as  $P \rightarrow \infty$ , and  $E_N(P) \rightarrow 0$  as  $P \rightarrow \infty$ .
- 8.4. (a) Redo the strong-scaling `ch7/minimal.c` runs which generated Figures 8.3–8.5. (Or use another PDE solver.) Then, using Amdahl's law (8.1), estimate the serial fraction  $f_s$  by linear regression to your timing data. For instance, (8.1) implies that  $y_* = t(N, P) - t(N, 1)/P$  should be proportional to  $x_* = t(N, 1)(1 - 1/P)$ :  $y_* = f_s x_*$ . Thus one might fit a line through the origin to the  $(x_*, y_*)$  data.
- (b) Precise use of Amdahl's law requires a method that is invariant with  $P$ , though this only occurs with suboptimal PDE solvers. For example,

```
| $ mpiexec -n P ./fish -fsh_dim 3 -da_refine 6 -pc_type jacobi
```

is independent of  $P$  and also converges in a reasonable time. Confirm for  $P = 1, 2, 4, 16, 64$  that the number of KSP iterations is  $P$  independent. Using such runs, redo the analysis in part (a). Is Amdahl's law a better fit here?

- 8.5. The law in section 2.3.3.1 of [48] says

$$t(N, P) = f_s t(N, 1) + (1 - f_s) \frac{t(N, 1)}{P} + t_c.$$

Continue Exercise 8.4 above by fitting the data, in an appropriate manner, to this model. Interpret your results.

- 8.6. In a figure like Figure 8.8, showing  $t(N, P)$  versus  $N/P$  on log-log axes, what will strong-scaling, static-scaling, and weak-scaling studies with good results look like? Find the strong-scaling (Figure 8.3) and weak-scaling (Figure 8.7) data points in Figure 8.8.
- 8.7. Suppose we want to solve the Poisson equation using GMG and large concurrency. For example, fix the number of MPI processes  $P > 9$  at some convenient level on your machine and do

```
| $ mpiexec -n P ./fish -da_refine L -pc_type mg
```

This run should give an error message for any value of  $L$  because the coarsest grid is too small; the one-degree-of-freedom-per-process restriction is active.

Now, for some attainable level  $L$ , compare the efficiency of redundant and telescope approaches by measuring the total flops and run time. First, use redundant and `-pc_mg_levels K` where  $K < L + 1$ . Vary  $K$  from less than  $L/2$  up to the highest value that runs without error (which depends on  $P$ ). Second, use the telescope options given in the text. Compare reduction factors  $r$  and intermediate levels  $k_1 \geq 2$ ; set  $k_2$  so that  $k_1 + k_2 = L + 2$ . (*It might be wise to reread the section on parallel multigrid in Chapter 7 before interpreting results.*)

- 8.8. To demonstrate that the order of summation can affect the result, first write a function which permutes the order of the elements stored in a one-dimensional C array, that is, implement a random shuffle algorithm. (*One might use a PETSc random number generator seeded by the wall clock time.*) Now modify `e.c` from Chapter 2 to create `lntwo.c` which does an  $N$ -term partial sum of the slowly converging infinite series

$$\sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} = \ln 2. \quad (8.4)$$

Do the finite sum by putting all the terms into an array, then randomly permuting it using your shuffle function, and then summing by any convenient loop. Show nondeterminacy of the least-significant digits using runs with  $N = 1000$  terms.

# **Part II**

# **Constructions**



## Chapter 9

# Finite element method I: Nonlinear optimization

The elliptic PDE problem in this chapter introduces a structured-grid *finite element* (FE) method, an easy transition from the FD schemes used so far. The problem is to minimize an objective functional in a function space, so we initially try to solve the discrete form by direct numerical minimization without gradient-evaluation code. This direct approach is, however, limited to very coarse grids, so we next add code to compute the gradient. The weak form of the PDE states that this gradient is zero, and this is the traditional starting point for FE presentations. Preconditioned Newton-Krylov methods can solve the resulting nonlinear algebraic equations, and, by using multigrid preconditioning, we achieve nearly optimal solver complexity in fine-grid cases.

### A $p$ -Helmholtz equation as minimization

Let  $\Omega$  be a domain in  $\mathbb{R}^2$  with well-behaved boundary, such as a polygon, and suppose  $f$  is a continuous function on  $\bar{\Omega}$ . For  $p > 1$  define

$$I[u] = \int_{\Omega} \frac{1}{p} |\nabla u|^p + \frac{1}{2} u^2 - fu. \quad (9.1)$$

This nonlinear functional is well defined and continuous on a space of functions with integrable gradient, namely the *Sobolev space*

$$W^{1,p}(\Omega) = \left\{ w : \int_{\Omega} |w|^p < \infty \text{ \& } \int_{\Omega} |\nabla w|^p < \infty \right\} \quad (9.2)$$

(see, e.g., [51, Chapter 5] or [60, Chapter 7]), a Banach space with norm

$$\|w\|_{W^{1,p}} = \left( \int_{\Omega} |w|^p + \int_{\Omega} |\nabla w|^p \right)^{1/p}.$$

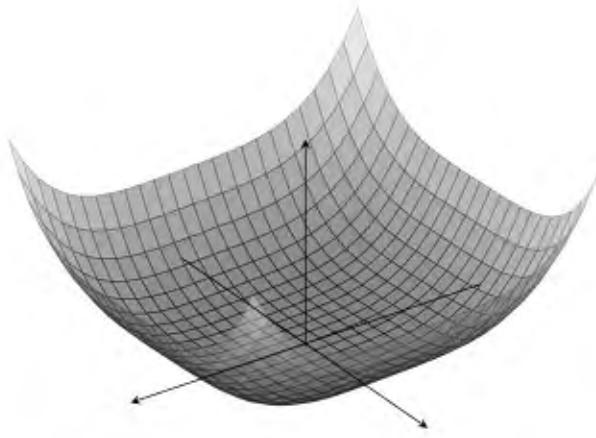
Figure 9.1 visualizes a function which is analogous to  $I[u]$  but defined on  $\mathbb{R}^2$  instead of the Banach space  $W^{1,p}(\Omega)$  (Exercise 9.1).

The functional (9.1) has two significant properties. First,  $I[u]$  is *coercive* in the sense that if the input is large in norm then the output is large:

$$\lim_{\|u\|_{W^{1,p}} \rightarrow +\infty} I[u] = +\infty. \quad (9.3)$$

Second it is *convex*, meaning that if  $u, v \in W^{1,p}(\Omega)$  and  $0 \leq \lambda \leq 1$  then

$$I[\lambda u + (1 - \lambda)v] \leq \lambda I[u] + (1 - \lambda)I[v] \quad (9.4)$$



**Figure 9.1.** If  $p = 4$  then the functional  $I[u]$  in (9.1) is analogous to a convex function like this surface.

(Exercise 9.2). In fact, because  $p > 1$ ,  $I[u]$  is *strictly convex*: if  $\|u - v\|_{W^{1,p}} > 0$  and  $0 < \lambda < 1$  then strict inequality applies in (9.4).

A standard theorem from the calculus of variations [51, Theorem 8.2.2] shows that coercivity and strict convexity of  $I[u]$  imply that our problem

$$\min_{w \in W^{1,p}(\Omega)} I[w] \quad (9.5)$$

has a unique solution  $u$ . The proof follows the familiar story that a continuous, real-valued function on a compact set achieves its extrema. Compactness arises from coercivity in the sense that sets of the form  $\{w : I[w] \leq L\}$ , which are bounded and closed, are compact in a certain topology on  $W^{1,p}(\Omega)$ , namely the weak topology. Convexity implies that  $I[u]$  is adequately continuous, i.e., in the weak topology (section 8.2 of [51]), and finally strict convexity is used to show uniqueness.

Being good calculus students, we solve a minimization problem like (9.5) by taking the derivative and setting it to zero. Because  $p > 1$  the functional  $I[u]$  does indeed have a gradient, and thus a minimizer also solves a nonlinear PDE. In fact, suppose  $\epsilon \in \mathbb{R}$  and  $u, v \in W^{1,p}(\Omega)$ . The binomial theorem implies

$$\begin{aligned} I[u + \epsilon v] - I[u] &= \int_{\Omega} \frac{1}{p} (|\nabla u + \epsilon \nabla v|^p - |\nabla u|^p) + \frac{1}{2} ((u + \epsilon v)^2 - u^2) - \epsilon f v \\ &= \epsilon \left( \int_{\Omega} |\nabla u|^{p-2} \nabla u \cdot \nabla v + uv - fv \right) + o(\epsilon), \end{aligned}$$

so the directional derivative is

$$\nabla I[u](v) = \lim_{\epsilon \rightarrow 0} \frac{I[u + \epsilon v] - I[u]}{\epsilon} = \int_{\Omega} |\nabla u|^{p-2} \nabla u \cdot \nabla v + uv - fv. \quad (9.6)$$

For each  $u \in W^{1,p}(\Omega)$ , (9.6) defines a linear and continuous map, the gradient  $\nabla I[u] : W^{1,p}(\Omega) \rightarrow \mathbb{R}$ . Thus if  $u \in W^{1,p}(\Omega)$  solves (9.5) then  $\nabla I[u](v) = 0$  or

$$\int_{\Omega} |\nabla u|^{p-2} \nabla u \cdot \nabla v + uv - fv = 0 \quad (9.7)$$

for all  $v \in W^{1,p}(\Omega)$ . The converse is also true (by convexity).

Equation (9.7) is the *weak form* of the *p-Helmholtz equation*, and it is also the *variational* or *Euler-Lagrange* equation of minimization problem (9.5). If the solution of (9.5) and/or (9.7) is

smooth enough, for instance if it has continuous second derivatives, then we can derive a *strong form* PDE as follows. (Sufficient smoothness actually does hold when the domain  $\Omega$  and data  $f$  are well behaved [51].) Assuming such smoothness, an integration by parts of (9.7) gives

$$\int_{\Omega} [-\nabla \cdot (|\nabla u|^{p-2} \nabla u) + u - f] v + \int_{\partial\Omega} v |\nabla u|^{p-2} \nabla u \cdot \mathbf{n} = 0. \quad (9.8)$$

Using functions  $v \in W^{1,p}(\Omega)$  which are zero along  $\partial\Omega$  shows that the term in square brackets is zero. Applying  $v$  which are only nonzero along and near  $\partial\Omega$ , so the first integral can be made arbitrarily small, it follows that  $\nabla u \cdot \mathbf{n} = \partial u / \partial n = 0$  on  $\partial\Omega$ . Thus if  $u$  solves (9.7) and is smooth then

$$-\nabla \cdot (|\nabla u|^{p-2} \nabla u) + u = f \text{ on } \Omega \quad \text{and} \quad \frac{\partial u}{\partial n} = 0 \text{ on } \partial\Omega, \quad (9.9)$$

that is, the problem has homogeneous Neumann boundary conditions. If  $p = 2$  then (9.9) reduces to a linear Helmholtz equation  $-\nabla^2 u + u = f$ .

In summary, problem (9.5) is equivalent to weak form (9.7), which can be converted to strong form, if the solution  $u$  is smooth, thus the  $p$ -Helmholtz PDE (9.9) arises from minimization. The differential operator in (9.9) is the  $p$ -Laplacian  $\Delta_p u = \nabla \cdot (|\nabla u|^{p-2} \nabla u)$ . This operator appears in applications including non-Newtonian fluids [30, 61] and differential games [50, 124]. Because the leading-order nonlinearity in  $p$ -Laplacian equations is more challenging than many zeroth-order nonlinearities, like those in Liouville-Bratu equation (Chapters 4 and 7), the  $p$ -Laplacian is also common in model problems for testing numerical methods for nonlinear elliptic PDEs [14, 18, 29, 36].

The coefficient  $D = |\nabla u|^{p-2}$  is regarded as a nonlinear *diffusivity*; compare Exercise 7.9. In *singular*  $p < 2$  cases the diffusivity becomes unbounded as  $|\nabla u| \rightarrow 0$ , while for  $p > 2$  it *degenerates* to zero as  $|\nabla u| \rightarrow 0$ . After we address the easy  $p = 2$  case, we will solve several  $p \neq 2$  cases.

Specific diffusion interpretations of the  $p$ -Laplacian apply in the extremes. If  $p = 1$  then  $\Delta_p$  computes the mean curvature of level surfaces (curves) of  $u$ , and thus it models isotropic diffusion within the level surfaces, with no diffusion between them [50]. The dual  $p = \infty$  case has diffusion along the gradient of  $u$ , but none in the level surfaces. Each extreme case thus allows nonsmooth solutions, even for bounded sources  $f$ , and we will not attempt numerical solutions. However, for  $1 < p < \infty$  the  $p$ -Laplacian can be written as an interpolation between  $p = 1, \infty$  [124] (Exercise 9.4), so these extremes may aid intuition.

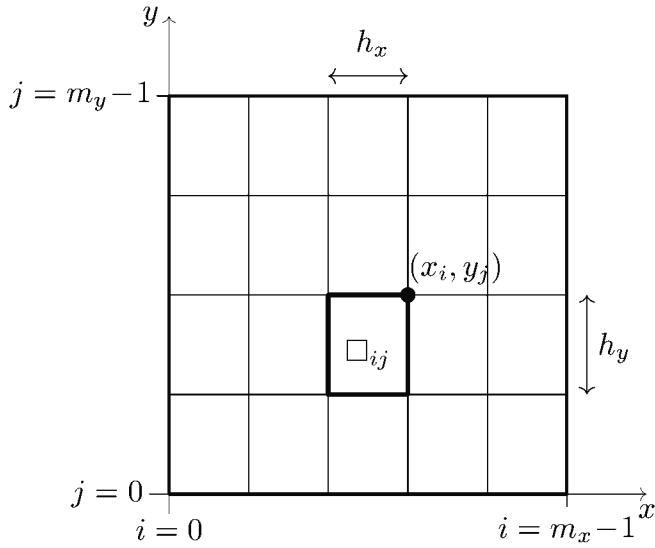
Before any numerical considerations we choose test problems. An exact solution is easy when  $f(x, y) = 1$ ; we observe that  $u(x, y) = 1$  trivially solves  $-\Delta_p u + u = 1$  and the Neumann boundary condition. This problem is called **constant** below. For a more interesting solution on the unit square  $\Omega = (0, 1)^2$  we manufacture a solution, called **cosines** below, which satisfies homogeneous Neumann boundary conditions:

$$u(x, y) = \cos(\pi x) \cos(\pi y). \quad (9.10)$$

The source function  $f(x, y)$  is computed by hand using (9.9), but note that  $|\nabla u| \rightarrow 0$  as  $(x, y)$  approaches a corner of  $\Omega$ . For  $p \neq 2$  the calculation is error prone, but at least by-hand errors are not always correlated with coding mistakes, so agreement may reflect correctness.

## Structured $Q_1$ finite elements

Our FE method approximates the solution  $u$  by a piecewise-polynomial function on a structured grid of rectangles. In contrast to FD methods in Chapters 3–7, the gridded unknowns determine a function from the same space  $W^{1,p}(\Omega)$  in which we seek the continuum solution.



**Figure 9.2.** The  $m_x \times m_y$  grid divides the unit square into elements  $\square_{ij}$  of area  $h_x h_y$ , each indexed by its upper-right corner.

Let  $\Omega = (0, 1)^2$  and consider the structured grid in Figure 9.2. For integers  $m_x \geq 2, m_y \geq 2$  let  $h_x = 1/(m_x - 1)$  and  $h_y = 1/(m_y - 1)$ . There are  $N = m_x m_y$  total nodes at locations  $x_i = ih_x, y_j = jh_y$ . The grid has  $K = (m_x - 1)(m_y - 1)$  rectangular *elements* indexed by their upper-right corners:

$$\square_{ij} = [x_{i-1}, x_i] \times [y_{j-1}, y_j].$$

These choices determine a DMDA structured-grid object (Chapter 3):

```
DMDACreate2d(COMM, DM_BOUNDARY_NONE, DM_BOUNDARY_NONE, DMDA_STENCIL_BOX,
2, 2, PETSC_DECIDE, PETSC_DECIDE, 1, 1, NULL, NULL, &da)
```

We use a box stencil because the contributions from the four elements incident to a given node  $(x_i, y_j)$  will depend on all nine nodal values. The default grid, with  $m_x = m_y = 2, N = 4$ , and  $K = 1$ , is as small as possible.

Functional (9.1) can be computed element by element,

$$I[u] = \sum_{i=1}^{m_x-1} \sum_{j=1}^{m_y-1} \int_{\square_{ij}} \frac{1}{p} |\nabla u|^p + \frac{1}{2} u^2 - fu, \quad (9.11)$$

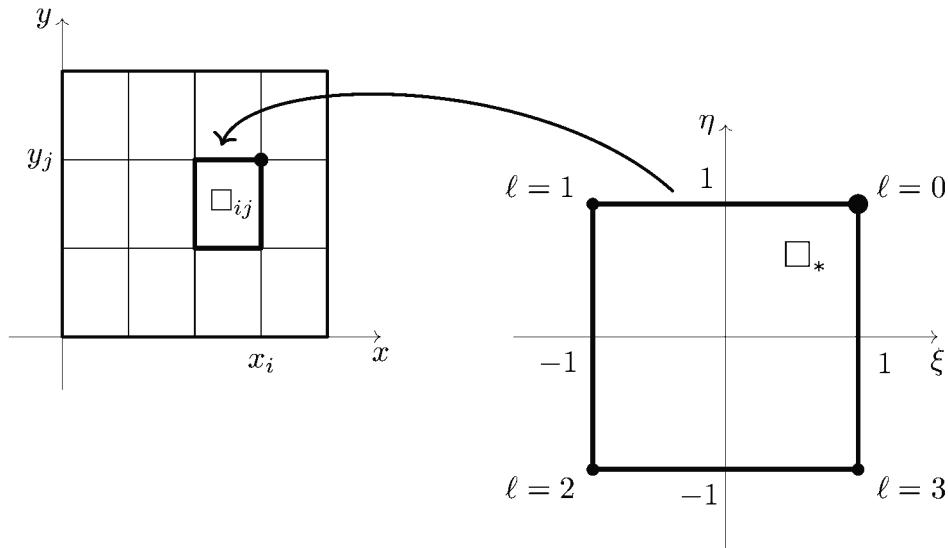
but how should we approximate  $u \in W^{1,p}(\Omega)$  on each element? A simple choice is the *bilinear* interpolant  $u^h|_{\square_{ij}} = a + bx + cy + dxy$ . Requiring  $u^h$  to be continuous on the whole domain  $\Omega$  then determines  $u^h$  from its nodal values  $u_{ij} = u^h(x_i, y_j)$  [49]. In fact, there is a linear isomorphism between nodal values in  $\mathbb{R}^N$  and the space

$$S^h = \left\{ v \in C(\Omega) \mid v|_{\square_{ij}} \text{ is bilinear} \right\} \subset W^{1,p}(\Omega). \quad (9.12)$$

Because the elements are quadrilaterals and the polynomial degree is one, i.e., in  $x$  and  $y$  separately,  $S^h$  is called the *Q<sub>1</sub> finite element space*.

The next step is to build a basis for  $S^h$ . We do this by constructing bilinear functions on a *reference element* (Figure 9.3)

$$\square_* = [-1, 1] \times [-1, 1].$$



**Figure 9.3.** Each element  $\square_{ij}$  is the image under map (9.16) of a reference element  $\square_*$  in  $\xi, \eta$  space.

In fact the use of a reference element is avoidable here, because our elements are congruent rectangles, but we are also thinking ahead toward unstructured meshes in Chapters 10, 13, and 14 where the strategy is advantageous. If  $v$  is bilinear on  $\square_*$  then  $v(\xi, \eta) = a + b\xi + c\eta + d\xi\eta$ , but the monomial basis  $\{1, \xi, \eta, \xi\eta\}$  is not convenient when manipulating nodal values. Instead we number the vertices of  $\square_*$  by  $\ell = 0, 1, 2, 3$  (Figure 9.3),

$$\begin{aligned} (\xi_0, \eta_0) &= (+1, +1), & (\xi_1, \eta_1) &= (-1, +1), \\ (\xi_2, \eta_2) &= (-1, -1), & (\xi_3, \eta_3) &= (+1, -1), \end{aligned} \quad (9.13)$$

and define four functions

$$\chi_\ell(\xi, \eta) = \frac{1}{4} (1 + \xi_\ell \xi) (1 + \eta_\ell \eta). \quad (9.14)$$

The  $\chi_\ell$  form a basis of bilinear functions on  $\square_*$  and, because  $\chi_\ell(\xi_{\ell'}, \eta_{\ell'}) = \delta_{\ell\ell'}$ , coefficients in this basis equal the nodal values on the reference element:

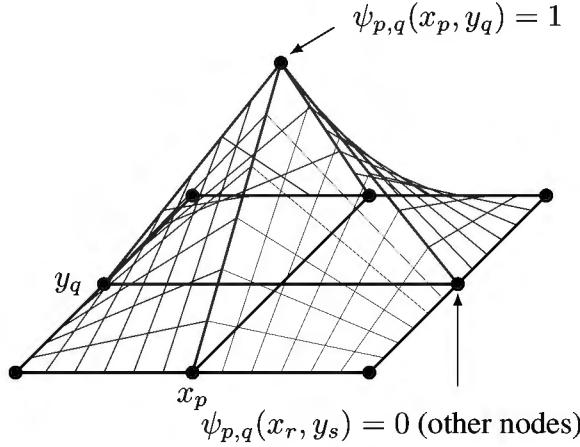
$$v(\xi, \eta) = \sum_{\ell=0}^3 v(\xi_\ell, \eta_\ell) \chi_\ell(\xi, \eta). \quad (9.15)$$

The element map  $\square_* \rightarrow \square_{ij}$  can then be written using basis functions:

$$\begin{aligned} x(\xi, \eta) &= \sum_{\ell=0}^3 x_\ell \chi_\ell(\xi, \eta) = x_i + \frac{h_x}{2}(\xi - 1), \\ y(\xi, \eta) &= \sum_{\ell=0}^3 y_\ell \chi_\ell(\xi, \eta) = y_j + \frac{h_y}{2}(\eta - 1). \end{aligned} \quad (9.16)$$

The Jacobian determinant of the map, used in integrals below, is a constant equal to the ratio of the areas of  $\square_{ij}$  and  $\square_*$ :

$$\left| \det \frac{\partial(x, y)}{\partial(\xi, \eta)} \right| = \det \begin{bmatrix} \frac{h_x}{2} & 0 \\ 0 & \frac{h_y}{2} \end{bmatrix} = \frac{h_x h_y}{4}. \quad (9.17)$$



**Figure 9.4.** A hat function  $\psi_{p,q} \in S^h$ .

For each node  $(x_p, y_q)$  in the grid there is a continuous, piecewise-bilinear *hat function*  $\psi_{p,q} \in S^h$ , defined on all of  $\overline{\Omega}$  and illustrated in Figure 9.4, which is equal to one on that node and zero at all others,

$$\psi_{p,q}(x_r, y_s) = \delta_{pr}\delta_{qs}. \quad (9.18)$$

Hat functions form a basis of  $S^h$ :

$$v(x, y) = \sum_{i=0}^{m_x-1} \sum_{j=0}^{m_y-1} v_{ij} \psi_{ij}(x, y). \quad (9.19)$$

Observe that  $\psi_{p,q}$  is identically zero on any element not incident to node  $(x_p, y_q)$ . Furthermore, when restricted to a particular element and then pulled back to the reference element  $\square_*$ , the hat function  $\psi_{p,q}$  is either identically zero or it is equal to one of the basis functions  $\chi_\ell$  in (9.14). Indeed, if corner  $(\xi_\ell, \eta_\ell) \in \square_*$  corresponds to node  $(x_p, y_q) \in \overline{\Omega}$  then

$$\psi_{p,q}(x(\xi, \eta), y(\xi, \eta)) = \chi_\ell(\xi, \eta). \quad (9.20)$$

Approximating  $I[u]$  in (9.11) also requires gradients (Exercise 9.5):

$$(\nabla_{x,y} \psi_{p,q})(x(\xi, \eta), y(\xi, \eta)) = \left\langle \frac{2}{h_x} \frac{\partial \chi_\ell}{\partial \xi}, \frac{2}{h_y} \frac{\partial \chi_\ell}{\partial \eta} \right\rangle; \quad (9.21)$$

note that derivatives  $\partial \chi_\ell / \partial \xi$  and  $\partial \chi_\ell / \partial \eta$  can be found from formula (9.14).

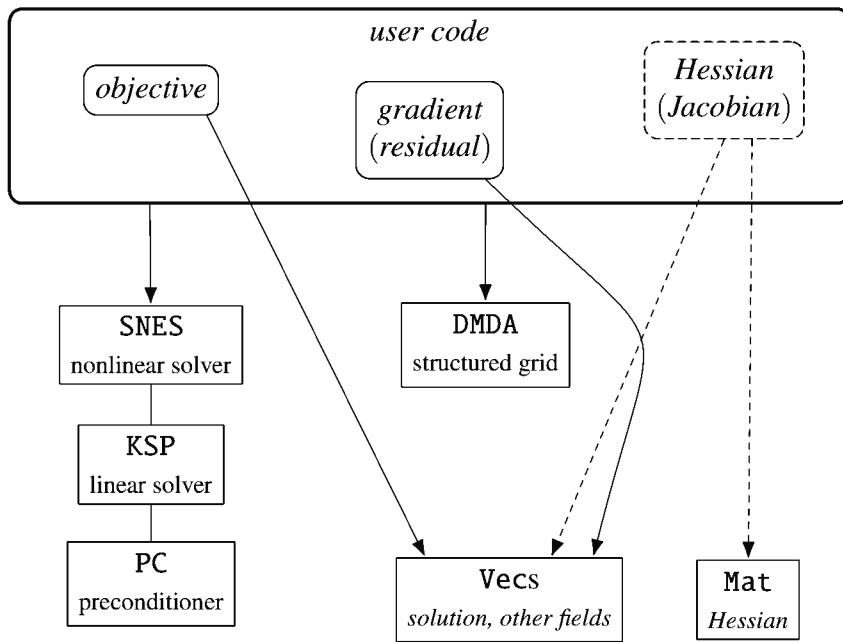
We do not plan to compute the integrals in (9.11) exactly. (For  $p \neq 2$  it would be challenging to exactly integrate  $|\nabla u|^p$ .) Instead, using tensor product quadrature with  $n$  points in each direction, formulas (I.2) and (I.3) from the Interlude (page 171) yield

$$\int_{\square_{ij}} v(x, y) dx dy \approx \frac{h_x h_y}{4} \sum_{r=0}^{n-1} \sum_{s=0}^{n-1} w_r w_s v(\xi_r, \xi_s). \quad (9.22)$$

(Note  $v(\xi, \eta) = v(x(\xi, \eta), y(\xi, \eta))$ .)

Finally, on the reference element we define

$$G_{ij}(u, \xi, \eta) = \left[ \frac{1}{p} |\nabla u|^p + \frac{1}{2} u^2 - fu \right]_{\square_*} \quad (9.23)$$



**Figure 9.5.** The PETSc stack for a structured-grid, minimization-type PDE problem, here without a Hessian (Jacobian) implementation; arrows mean “user code acts directly on.” Compare Figures 3.13 and 4.4.

using the nodal values of  $u$  and  $f$  on element  $\square_{ij}$  (Exercise 9.6). Now formulas (9.19), (9.21), (9.22), and (9.23) combine to approximate (9.11) by the following sum:

$$I^h[u] = \frac{h_x h_y}{4} \underbrace{\sum_{i=1}^{m_x-1} \sum_{j=1}^{m_y-1}}_{\text{elements}} \underbrace{\sum_{r=0}^{n-1} \sum_{s=0}^{n-1}}_{\text{quadrature points}} w_r w_s G_{ij}(u, \xi_r, \xi_s). \quad (9.24)$$

This is enough detail for an initial implementation.

## Objective only: Implementation

Figure 9.5 shows the PETSC components we will use. Initially we only implement the objective function  $I^h[u]$ , namely equation (9.24), because numerical minimization algorithms can use finite-difference approximations for the gradient. Though we delay its use until later, gradient-evaluation code will, in fact, be needed because our objective-only approach only works on very coarse grids. However, we never do implement an analytical Jacobian, i.e., we never write code to provide the *Hessian* of the objective.

Note that in an optimization context as here, “gradient” and “residual” refer to the same vector-valued function, and “Hessian” and “Jacobian” to the same matrix-valued function. They are the first and second derivatives of the objective, respectively. (See Exercises 9.11 and 9.12.)

Two snippets of our program `phelm.c` are shown, Codes 9.1 and 9.2. The first shows FE tools generic to any 2D  $Q_1$  method based on a reference element. These tools evaluate basis functions  $\chi_\ell$  at points  $(\xi, \eta) \in \square_*$ , implement formulas (9.14) and (9.15), and evaluate partial derivatives of  $\chi_\ell$ . Note that formula (9.21) generates a vector, the components of the gradient at a point, which is stored in a two-entry `struct`. Two additional functions compute pointwise inner products  $\nabla u \cdot \nabla v$  and powers  $|\nabla u|^q$  from nodal values.

```

static PetscReal xiL[4] = { 1.0, -1.0, -1.0, 1.0},
                           etaL[4] = { 1.0, 1.0, -1.0, -1.0};

static PetscReal chi(PetscInt L, PetscReal xi, PetscReal eta) {
    return 0.25 * (1.0 + xiL[L] * xi) * (1.0 + etaL[L] * eta);
}

// evaluate v(xi,eta) on reference element using local node numbering
static PetscReal eval(const PetscReal v[4], PetscReal xi, PetscReal eta) {
    return v[0] * chi(0,xi,eta) + v[1] * chi(1,xi,eta)
        + v[2] * chi(2,xi,eta) + v[3] * chi(3,xi,eta);
}

typedef struct {
    PetscReal xi, eta;
} gradRef;

static gradRef dchi(PetscInt L, PetscReal xi, PetscReal eta) {
    const gradRef result = {0.25 * xiL[L] * (1.0 + etaL[L] * eta),
                           0.25 * etaL[L] * (1.0 + xiL[L] * xi)};
    return result;
}

// evaluate partial derivs of v(xi,eta) on reference element
static gradRef deval(const PetscReal v[4], PetscReal xi, PetscReal eta) {
    gradRef sum = {0.0,0.0}, tmp;
    PetscInt L;
    for (L=0; L<4; L++) {
        tmp = dchi(L,xi,eta);
        sum.xi += v[L] * tmp.xi; sum.eta += v[L] * tmp.eta;
    }
    return sum;
}

static PetscReal GradInnerProd(PetscReal hx, PetscReal hy,
                               gradRef du, gradRef dv) {
    const PetscReal cx = 4.0 / (hx * hx), cy = 4.0 / (hy * hy);
    return cx * du.xi * dv.xi + cy * du.eta * dv.eta;
}

static PetscReal GradPow(PetscReal hx, PetscReal hy,
                        gradRef du, PetscReal P, PetscReal eps) {
    return PetscPowScalar(GradInnerProd(hx,hy,du,du) + eps*eps, P/2.0);
}

```

**Code 9.1.** *c/ch9/phelm.c, part I. Tools for 2D  $Q_1$  FE methods.*

Code 9.2 shows the parallel evaluation of  $I^h[u]$ . The first function implements pointwise integrand function  $G_{ij}(\xi, \eta)$  from (9.23). In the second function each MPI rank accumulates its portion of the sum over elements (9.24) into a variable `lobj` using the quadrature weights and nodes:

```
lobj += q.w[r] * q.w[s] * ObjIntegrandRef(info,f,u,q.xi[r],q.xi[s],user);
```

The local contributions `lobj` are then summed across the MPI communicator (i.e., reduced) into the global value `obj` =  $I^h[u]$ :

```
MPI_Allreduce(&lobj,obj,1,MPIU_REAL,MPIU_SUM,com);
```

```

static PetscReal ObjIntegrandRef(DMDALocalInfo *info ,
                                 const PetscReal ff[4], const PetscReal uu[4],
                                 PetscReal xi, PetscReal eta, PHelmCtx *user) {
    const gradRef du = eval(uu, xi, eta);
    const PetscReal hx = 1.0 / (info->mx-1), hy = 1.0 / (info->my-1),
                  u = eval(uu, xi, eta);
    return GradPow(hx, hy, du, user->p, 0.0) / user->p + 0.5 * u * u
           - eval(ff, xi, eta) * u;
}

PetscErrorCode FormObjectiveLocal(DMDALocalInfo *info, PetscReal **au,
                                  PetscReal *obj, PHelmCtx *user) {
    PetscErrorCode ierr;
    const PetscReal hx = 1.0 / (info->mx-1), hy = 1.0 / (info->my-1);
    const Quad1D q = gausslegendre[user->quadpts-1];
    PetscReal x, y, lobj = 0.0;
    PetscInt i, j, r, s;
    MPI_Comm com;

    // loop over all elements
    for (j = info->ys; j < info->ys + info->ym; j++) {
        if (j == 0)
            continue;
        y = j * hy;
        for (i = info->xs; i < info->xs + info->xm; i++) {
            if (i == 0)
                continue;
            x = i * hx;
            const PetscReal ff[4] = {user->f(x, y, user->p, user->eps),
                                    user->f(x-hx, y, user->p, user->eps),
                                    user->f(x-hx, y-hy, user->p, user->eps),
                                    user->f(x, y-hy, user->p, user->eps)};
            const PetscReal uu[4] = {au[j][i], au[j][i-1],
                                    au[j-1][i-1], au[j-1][i]};
            // loop over quadrature points on this element
            for (r = 0; r < q.n; r++) {
                for (s = 0; s < q.n; s++) {
                    lobj += q.w[r] * q.w[s]
                           * ObjIntegrandRef(info, ff, uu,
                                             q.xi[r], q.xi[s], user);
                }
            }
        }
    }
    lobj *= hx * hy / 4.0; // from change of variables formula
    PetscObjectGetComm((PetscObject)(info->da), &com);
    MPI_Allreduce(&lobj, obj, 1, MPIU_REAL, MPIU_SUM, com);
    PetscLogFlops(129*info->xm*info->ym);
    return 0;
}

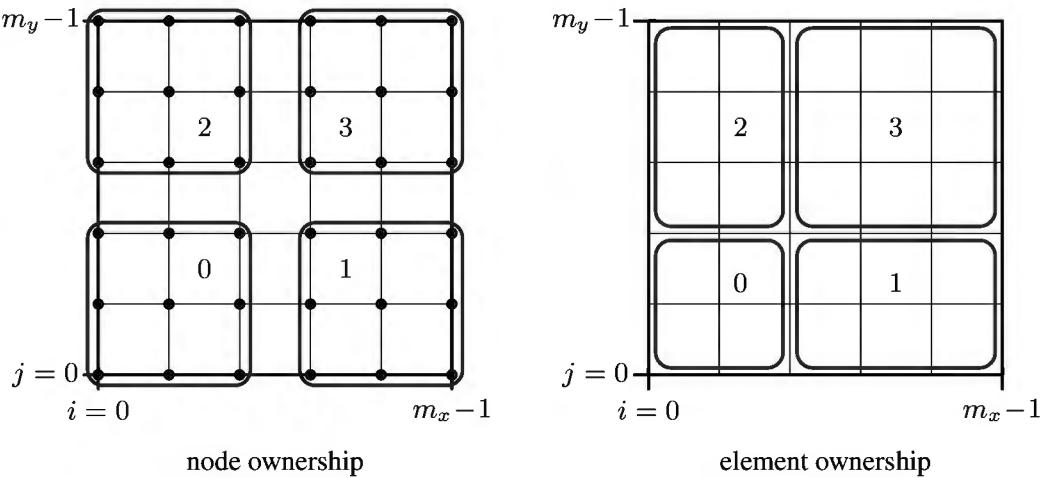
```

**Code 9.2.** *c/ch9/phelm.c, part II. Implementation of (9.24).*

Computing  $I^h[u]$  in parallel requires clarity on the distribution of nodes and elements. To do sum (9.24) we assign ownership of each element to a unique rank. Figure 9.6 shows the case of four MPI processes. The DMDA object distributes the nodes (Chapter 3), but here we make the additional decision that a rank owns an element if it is down or left from an owned node.

The `main()` function (not shown) sets options (prefix `ph_`) and then sets up a DMDA and a SNES. The SNES is told to use `FormObjectiveLocal()` by the call

```
DMDASNESSetObjectiveLocal(da, (DMDASNESObjective)FormObjectiveLocal, &user)
```



**Figure 9.6.** Parallel ownership of nodes (left) with four MPI processes. In evaluating the objective we must assign unique element ownership (right).

Then `main()` calls `SNESSolve()`, computes the numerical error, and destroys all the objects at the end.

There are two exact solution cases, `-ph_problem constant` with  $f(x, y) = 1$  and `-ph_problem cosines` with  $f(x, y)$  manufactured from (9.10). An initial iterate  $u(x, y) = 0.5$  is the default, but there is also an option to use the exact solution as the initial iterate. Other default values include  $p = 2$  for the exponent and  $n = 2$  for quadrature.

## Objective only: Solvers

To run `phelm.c` in objective-only mode requires option `-ph_no_gradient` to turn off the gradient code in Code 9.3 below. The following is in the default case where  $I^h[u]$  is quadratic and PDE (9.9) is linear:

```
$ cd c/ch9/ && make phelm
$ ./phelm -da_refine 2 -snes_converged_reason \
    -ph_no_gradient -snes_fd_function -snes_fd_color
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 9
done on 5 x 5 grid with p=2.000 ...
numerical error: |u-u_exact|_inf = 4.749e-02
```

Option `-snes_fd_function` asks SNES to do finite differencing on the objective functional  $I^h[u]$  to generate a function which evaluates the gradient/residual  $\mathbf{F}(\mathbf{u}) = \nabla I^h[\mathbf{u}]$ . Then one of `-snes_fd`, `-snes_fd_color`, `-snes_mf`, or `-snes_mf_operator` will compute the entries or action of the Hessian, i.e., the Jacobian of the residual, also by finite differencing, for each Newton step. On this structured grid we try `-snes_fd_color` for efficiency.

However, the disadvantages of this approach are likely clear. The computed Hessian must be a terrible approximation because it is *doubly* finite differenced. If  $\epsilon$  is machine precision then we expect  $O(\epsilon^{1/4})$  rounding error and thus only a few correct digits even with `double` precision, thus the computed search directions are far too “noisy.” For example, with `-da_refine 5` the above run reports `CONVERGED_SNORM_RELATIVE`, but the putative solution has low accuracy.

The approach is also very slow. First, finite differencing requires at least  $N + 1$  evaluations of the objective function to compute one gradient (residual), where  $N$  is the total number of degrees of freedom. (Because of the way `SNESObjectiveComputeFunctionDefaultFD()` is implemented, the true number is more than  $3N$  evaluations.) Then finite differencing using

coloring yields the Hessian in nine gradient evaluations, because of the box stencil. Additional objective evaluations are needed in the line search and in checking the convergence tolerance. In summary, about  $30N$  objective evaluations are needed per Newton step. Such an approach produces unsustainable  $O(N^2)$  scaling as  $N \rightarrow \infty$ , because each objective evaluation does  $O(N)$  flops.

One may count actual evaluations by adding `-log_view | grep Eval` to the run above. On the  $5 \times 5$  grid (`-da_refine 2`) the objective is evaluated more than 9000 times, rising to  $1.2 \times 10^5$  times for the  $17 \times 17$  grid (`-da_refine 4`).

However, there exist other algorithms which avoid such direct dependence on a noisy Hessian in determining the search direction. The following options can replace `-snes_fd_color` in the above run:

- (i) Preconditioned Jacobian-free Newton-Krylov with the noisy Hessian used only for preconditioning:

```
-snes_fd_color -snes_mf_operator
```

Like the run above, this approach uses `-snes_type newtonls`.

- (ii) The limited-memory BFGS method [118], a quasi-Newton (QN) method, which replaces the Hessian with increasing-rank approximations, as the iteration proceeds, computed from outer products of gradients:

```
-snes_type qn
```

- (iii) Nonlinear conjugate gradients (NCG) [118]. This extends the linear CG method (Chapter 2) to a minimization method suitable for not-necessarily quadratic functionals like our  $p$ -Helmholtz functional:

```
-snes_type ncg
```

The TAO library, distributed with PETSC, has additional nongradient methods such as Nelder-Mead [118], but none are expected to give improved scaling.

The above methods all use the same approximated gradient, namely the finite differenced result from `-snes_fd_function`, but they avoid generating Newton steps directly from the doubly-differenced Hessian. Note that in (ii) the default line-search type also changes from `bt` to `cp` (Chapter 4; compare [29]). Even with smooth gradients, however, quadratic convergence would not be expected in (ii) or (iii). In fact, while BFGS can exhibit superlinear convergence, NCG generally has linear convergence [118].

To compare these solvers note that the finite differenced residual has  $O(\epsilon^{1/2})$  accuracy so in double precision the default `-snes_rtol 1.0e-8` may be asking too much. Resetting this to  $10^{-6}$ , in Table 9.1 we compare the methods by number of SNES iterations. (The run times—not shown—are horrible, but solver convergence is the concern here.)

**Table 9.1.** Number of SNES iterations on various grids, DIVERGED errors, for objective-only solver methods. The asterisk indicates a low-accuracy solution.

	$2 \times 2$	$3 \times 3$	$5 \times 5$	$9 \times 9$	$17 \times 17$	$33 \times 33$
original	5	5	7	7	8	3*
(i) MF	4	5	5	8	DIVERGED	DIVERGED
(ii) QN	2	4	9	20	37	68
(iii) NCG	2	4	9	20	37	68

QN and NCG methods are preferred when running objective only, because the solver converges, including on finer grids (not shown), to discretization accuracy. The coincidence of their iterations is because the objective function is quadratic (Exercise 9.7; see [54]); for general objective functions QN and NCG do not generate the same search directions.

Given that we have an exact solution, we also want to see numerical convergence on these coarse grids. Using numerical errors from the QN/NCG runs, and excluding data from the two coarsest grids, we find a convergence rate  $O(h^{1.99})$ —see Figure 9.7 below—suggesting we have correctly implemented the objective functional.

In summary, our 2D Helmholtz equation arises from minimization in a function space and it is solvable using PETSc's ability to minimize an objective functional via a finite differenced gradient. Performance is poor using any of the above approaches; the large number of objective evaluations limits these methods to very coarse grids.

## Using a gradient function

Writing gradient code is now well motivated. Recall that the weak form (9.7) says  $\nabla I[u](v) = 0$  for all  $v \in W^{1,p}(\Omega)$ . The hat functions, which form a basis of  $S^h$ , then generate a finite nonlinear system  $\mathbf{F}(\mathbf{u}) = \mathbf{0}$  with one equation per node; we require  $u \in S^h$  to satisfy

$$\int_{\Omega} |\nabla u|^{p-2} \nabla u \cdot \nabla \psi_{pq} - f \psi_{pq} = 0 \quad (9.25)$$

for each node  $(x_p, y_q)$ . Expanding this integral over elements,

$$\sum_{i=0}^{m_x} \sum_{j=0}^{m_y} \int_{\square_{ij}} |\nabla u|^{p-2} \nabla u \cdot \nabla \psi_{pq} - f \psi_{pq} = 0, \quad (9.26)$$

observe that at most four distinct hat functions  $\psi_{pq}$  are nonzero on any element  $\square_{ij}$ . Therefore we define the integrand on the reference element,

$$H_{ij}^{\ell}(u, \xi, \eta) = [|\nabla u|^{p-2} \nabla u \cdot \nabla \psi_{pq} - f \psi_{pq}]_{\square_*}, \quad (9.27)$$

with the understanding that on  $\square_{ij}$  an index pair  $(p, q)$  corresponds to local index  $\ell$  under map (9.20). Note that the gradient in (9.27) is in the  $(x, y)$  variables, so chain rule (9.21) applies (Exercise 9.8).

To determine equation  $F_{pq}(u) = 0$  for node  $(x_p, y_q)$ , the four nonzero element integrals in (9.26) are each approximated using quadrature (9.22). These are assembled from the element  $\square_{ij}$  contributions of those  $\psi_{pq}$  which have value one at the  $\ell$ th corner of the reference element:

$$F_{pq}(u) += \frac{h_x h_y}{4} \sum_{r=0}^{n-1} \sum_{s=0}^{n-1} w_r w_s H_{ij}^{\ell}(u, \xi_r, \xi_s). \quad (9.28)$$

The resulting system  $\mathbf{F}(\mathbf{u}) = \mathbf{0}$  should determine all  $N$  nodal values  $u_{ij}$ .

Code 9.3 shows `FormFunctionLocal()` which implements (9.28) and `IntegrandRef()` which implements (9.27) on the reference element using tools from Code 9.1. As with FD examples in previous chapters, we access a ghosted local `Vec` via an array (`PetscReal **au`). In parallel, where ownership of nodes follows Figure 9.6, constructing certain equations requires nodal values owned by neighboring processes.

```

static PetscReal IntegrandRef(DMDALocalInfo *info, PetscInt L,
                             const PetscReal ff[4], const PetscReal uu[4],
                             PetscReal xi, PetscReal eta, PHelmCtx *user) {
    const gradRef du = deval(uu, xi, eta),
          dchiL = dchi(L, xi, eta);
    const PetscReal hx = 1.0 / (info->mx-1), hy = 1.0 / (info->my-1);
    return GradPow(hx, hy, du, user->p - 2.0, user->eps)
           * GradInnerProd(hx, hy, du, dchiL)
           + (eval(uu, xi, eta) - eval(ff, xi, eta)) * chi(L, xi, eta);
}

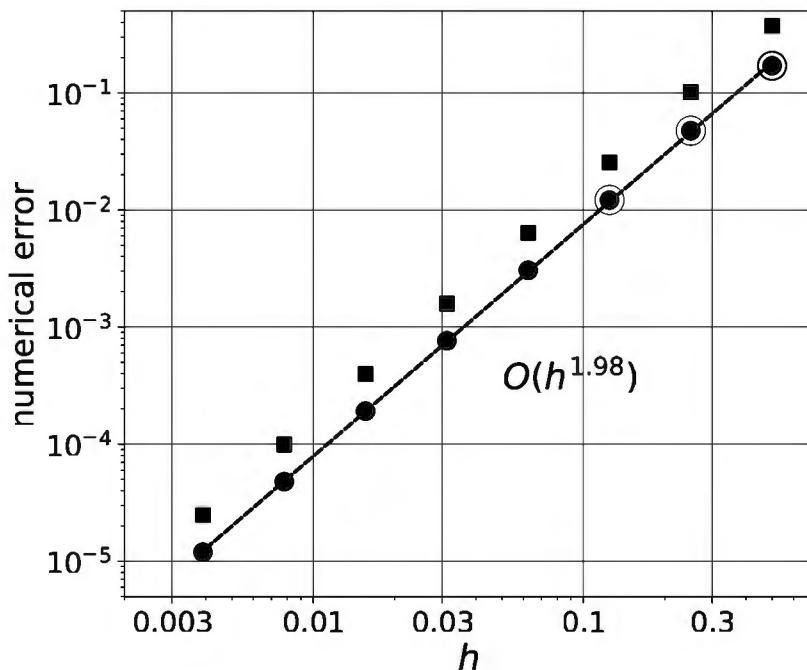
PetscErrorCode FormFunctionLocal(DMDALocalInfo *info, PetscReal **au,
                                PetscReal **FF, PHelmCtx *user) {
    const PetscReal hx = 1.0 / (info->mx-1), hy = 1.0 / (info->my-1);
    const Quad1D q = gausslegendre[user->quadpts-1];
    const PetscInt li[4] = {0,-1,-1,0}, lj[4] = {0,0,-1,-1};
    PetscReal x, y;
    PetscInt i, j, l, r, s, PP, QQ;

    // clear residuals
    for (j = info->ys; j < info->ys + info->ym; j++)
        for (i = info->xs; i < info->xs + info->xm; i++)
            FF[j][i] = 0.0;

    // loop over all elements
    for (j = info->ys; j <= info->ys + info->ym; j++) {
        if ((j == 0) || (j > info->my-1))
            continue;
        y = j * hy;
        for (i = info->xs; i <= info->xs + info->xm; i++) {
            if ((i == 0) || (i > info->mx-1))
                continue;
            x = i * hx;
            const PetscReal ff[4] = {user->f(x,y,user->p,user->eps),
                                     user->f(x-hx,y,user->p,user->eps),
                                     user->f(x-hx,y-hy,user->p,user->eps),
                                     user->f(x,y-hy,user->p,user->eps)};
            const PetscReal uu[4] = {au[j][i], au[j][i-1],
                                    au[j-1][i-1], au[j-1][i]};
            // loop over corners of element i,j
            for (l = 0; l < 4; l++) {
                PP = i + li[l];
                QQ = j + lj[l];
                // only update residual if we own node
                if (PP >= info->xs && PP < info->xs + info->xm
                    && QQ >= info->ys && QQ < info->ys + info->ym) {
                    // loop over quadrature points
                    for (r = 0; r < q.n; r++) {
                        for (s = 0; s < q.n; s++) {
                            FF[QQ][PP]
                                += 0.25 * hx * hy * q.w[r] * q.w[s]
                                * IntegrandRef(info, l, ff, uu,
                                               q.xi[r], q.xi[s], user);
                        }
                    }
                }
            }
        }
    }
    PetscLogFlops((5+q.n*q.n*149)*(info->xm+1)*(info->ym+1));
    return 0;
}

```

**Code 9.3.** *c/ch9/phelm.c, part III. Gradient  $\mathbf{F}(\mathbf{u}) = \nabla I^h[\mathbf{u}]$  from (9.26).*



**Figure 9.7.** Convergence for the  $p = 2$  Helmholtz problem. Objective-only results agree on coarse grids (circled). Midpoint quadrature with  $n = 1$  (squares) generates somewhat larger errors than default  $n = 2$  quadrature (dots).

Because we have replaced  $O(N)$  objective evaluations with a single gradient evaluation, our code will now run much faster. The difference is huge even on modest grids. For example, each SNES iteration in the run

```
| $ ./phelm -da_refine 4 -snes_rtol 1.0e-6 -snes_converged_reason
```

is 100 times faster than the objective-only version.<sup>29</sup> Just as important, the new code generates much higher-quality search directions in the Newton iteration; the objective-only version needs several times more SNES iterations on this grid.

Adding `-log_view | grep Eval` to count evaluations in the above run shows two objective evaluations per SNES iteration because the line search uses the supplied objective instead of the default merit function, namely  $\phi$  in equation (4.23). That is, though SNES constructs a merit function from the residual when no objective is provided, when the PDE arises from a minimization principle then providing the objective function should improve the line search.

We can now give convincing evidence of convergence (Figure 9.7) in the easy  $p = 2$  case:

```
for LEV in 1 2 3 4 5 6 7 8; do
    ./phelm -da_refine $LEV -snes_converged_reason
done
```

The numerical errors  $\|u - u_{\text{exact}}\|_\infty$  converge at the  $O(h^2)$  rate expected for uniformly elliptic PDEs on convex polygonal domains with smooth coefficient functions [49].

Recall that we have implemented 2D Gauss-Legendre quadrature for  $n = 1, 2, 3$  points in each direction. So far we have only used the default  $n = 2$  case, which suffices to exactly integrate cubic polynomials. Adding option `-ph_quadpts 1` to the above runs, for midpoint-rule quadrature, makes the numerical error larger by a factor of two (Figure 9.7). By contrast, switching to `-ph_quadpts 3` generates virtually the same errors as  $n = 2$ , but the run time increases.

<sup>29</sup>I.e., with `-ph_no_gradient -snes_fd_function`. Remember that `-snes_fd_color` is the default for DM<sup>A</sup> codes without Jacobian procedures; both runs under consideration use it.

In fact the run time is dominated by the time spent evaluating the integrand at quadrature points, with the  $n = 1$  time about 60% of the  $n = 2$  time, which in turn is about 60% of the  $n = 3$  time. This brief analysis justifies using  $n = 2$  quadrature points in each direction.

## Regularization for $p \neq 2$

For  $p = 2$  the functional (9.1) is quadratic, the PDE (9.9) is linear, and the problem is easy. Now we take an empirical approach to solving some  $p \neq 2$  cases, using the gradient evaluation just described.

Results from the following runs, with various exponents and grids, are shown in Table 9.2:

```
| $ ./phelm -snes_rtol 1.0e-5 -snes_converged_reason -ph_p P -da_refine LEV
```

The table shows either the number of SNES iterations if converged or, if not, the DIVERGED flavor. These results suggest that convergence of the Newton iteration requires modifying the gradient evaluation even on these coarse grids.

**Table 9.2.** SNES iterations or the DIVERGED flavor, e.g., DIVERGED\_FNORM\_NAN, for unregularized runs.

$\backslash$ $p$	Grid 9 × 9	Grid 33 × 33	Grid 129 × 129
1.1	FNORM_NAN	FNORM_NAN	FNORM_NAN
1.5	FNORM_NAN	FNORM_NAN	FNORM_NAN
2	2	2	2
4	10	13	LINEAR_SOLVE
10	21	LINEAR_SOLVE	LINEAR_SOLVE

Cases with  $p < 2$  fail immediately because NaN (not-a-number) values appear in the calculation. A moment of thought shows the NaNs must be generated in `FormFunctionLocal()`. The objective integrand uses a positive exponent, so it will not generate NaNs, and neither will the KSP and SNES internal methods. In fact, the exact solution  $u(x, y) = \cos(\pi x) \cos(\pi y)$ , used to generate the right-hand side  $f(x, y)$  of (9.9), has  $\nabla u \rightarrow 0$  in the corners of the unit square  $\Omega$ . At these points the coefficient  $|\nabla u|^{p-2}$ , and  $|f(x, y)|$  also, goes to infinity when  $p < 2$ . A quick check with option `-ph_view_f`, added for this diagnosis, confirms that gridded values of  $f(x, y)$  include NaNs.

We propose to use a parameter  $\epsilon > 0$  to regularize the diffusivity coefficient:

$$D_\epsilon(u) = (|\nabla u|^2 + \epsilon^2)^{(p-2)/2}. \quad (9.29)$$

This change avoids division by zero in  $p < 2$  cases. Notice that  $D_\epsilon$  is bounded above by  $\epsilon^{p-2}$  if  $p \leq 2$  and bounded below by the same quantity if  $p \geq 2$ . We use  $D_\epsilon$  in the weak form (9.7) and also in manufacturing the right-hand side from the exact solution (i.e.,  $f_\epsilon(x, y) = -\nabla \cdot (D_\epsilon(u) \nabla u) + u$ ). However, the original minimization goal remains because we do not regularize the functional in (9.24); the original objective is well defined for all  $p \geq 1$ . Line-search Newton minimization of  $I^h[u]$  now has better search directions, calculated from the regularized weak form, but we are solving the unmodified minimization problem.

Because we now have positive lower and upper bounds on the diffusivity coefficient, the problem is uniformly elliptic [51], and furthermore the condition number of the Hessian (i.e., Jacobian) relates to the bounds on the diffusivity coefficient. We expect that  $\kappa_2(A) \leq Ch^{-2}$  for a constant  $C$  constructed from these bounds and the geometry of the mesh [49, Theorem 1.33],

and indeed we see  $O(h^{-2})$  growth of the condition number in practice (below). On the other hand, loss of ellipticity for  $p > 2$  may be physically meaningful in models, as it may correspond to actual degeneration of diffusivity (e.g., [30]).

Given that the finite differenced Hessian is only accurate to one part in  $10^8$  anyway, the regularization in (9.29) defaults to  $\epsilon = 10^{-4}$ ; compare  $\epsilon = 10^{-5}$  for a similar problem in [29]. The new data in Table 9.3, generated using `-ph_eps 1.0e-4`, shows progress in  $p < 2$  cases but no essential change for  $p \geq 2$ . Evidently either a fine grid or  $p$  far from 2 makes the problem harder. All cases now seem to succeed on very coarse grids but if  $p \neq 2$  then they generate “bad” linear systems on fine grids.

**Table 9.3.** SNES iterations, or the DIVERGED flavor, using regularization (9.29) with  $\epsilon = 10^{-4}$ .

$p \backslash$ Grid	$9 \times 9$	$33 \times 33$	$129 \times 129$
1.1	19	28	LINEAR_SOLVE
1.5	6	6	LINEAR_SOLVE
2	2	2	2
4	10	13	LINEAR_SOLVE
10	20	LINEAR_SOLVE	LINEAR_SOLVE

Our DIVERGED\_LINEAR\_SOLVE errors suggest certain possibilities. Perhaps the Newton iteration is approaching a point where  $\|\nabla I(\mathbf{u}^{(k)})\| = \|\mathbf{F}(\mathbf{u}^{(k)})\| \rightarrow 0$ , but where the Hessian becomes singular. Alternatively, perhaps  $\|\mathbf{u}^{(k)}\|$  diverges to infinity in a singular-Hessian direction (in the refinement limit). In fact these are the only possibilities; iterative minimization methods using appropriate line searches will either converge to stationary points of  $I$  ( $\|\nabla I\| = \|\mathbf{F}\| = 0$ ) or the Hessian will become singular [118, section 3.2]. Thus a bound on Hessian condition numbers implies that  $\|\mathbf{F}(\mathbf{u}^{(k)})\| \rightarrow 0$ .

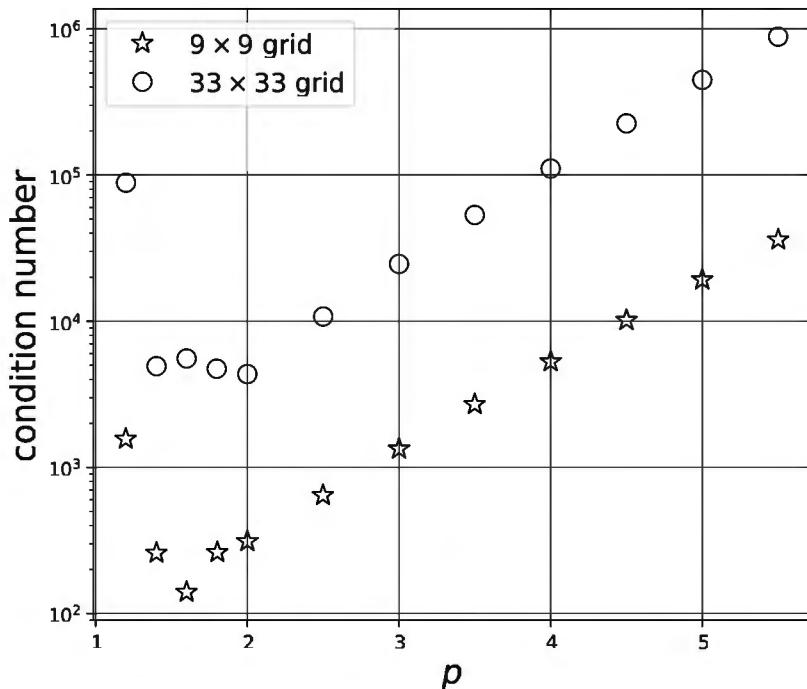
Several approaches can be used to investigate further:

- (i) Option `-ph_exact_init` uses the gridded values of the exact continuum solution  $u(x, y)$  as initial values, which puts the initial iterate inside the domain of quadratic convergence of the Newton iteration. In fact, all cases now exhibit SNES convergence. On the  $129 \times 129$  grid with  $p = 1.1$  the discrete solution does not have small numerical error, but for other  $p$  there is clear evidence of numerical convergence (not shown).
- (ii) One may estimate the condition number [143] of the Hessian at each Newton step using option `-ksp_view_singularvalues`. The output reports “max/min Z” values where Z is the estimated condition number (Chapter 3). Because the condition number estimate improves as the number of KSP iterations increases, and because our concern is with the unpreconditioned linear system, a useful option combination is

```
-pc_type none -ksp_rtol 1.0e-12 -ksp_view_singularvalues
```

(When using GMRES one also avoids restarts by using a large value for `-ksp_gmres_restart`.) Computing the condition number this way only works on relatively coarse grids because we are solving an unpreconditioned system with a slow method.

Figure 9.8 shows the results from regularized runs with  $\epsilon = 10^{-4}$ . The estimated condition numbers increase steadily with  $p > 2$ , but the numerical errors remain small (not shown). While the condition number trend may explain convergence failures with large  $p$ , perhaps including  $p = 10$ , this is not the obvious culprit for most of our fine-grid SNES failures.



**Figure 9.8.** Condition numbers at the converged solution, for coarse grids and a range of  $p$  values.

- (iii) An attempt to diagnose our difficulties by visualizing  $p < 2$  runs also makes sense. Two options are

```
-snes_monitor_solution draw
-snes_monitor_solution_update draw
```

(One typically adds `-draw_pause 1` or similar.) For  $p < 2$  runs on the  $33 \times 33$  grid from Figure 9.8, namely  $p = 1.8, 1.6, 1.4, 1.2$  in turn, the solution concentrates into the corners of  $\Omega$  more than it should (not shown). For  $p = 1.1$  the updates (Newton steps) show significant loss of smoothness near the corners. However, while visualization may help intuition, it does not suggest a solution strategy.

- (iv) An investigation into line-search types (`-snes_linesearch_type`) is also reasonable. Of the several types in Table 4.4, the critical point type `cp` is the most promising. However, testing shows that the same  $p < 2$  cases as in Figure 9.8 remain problematic, but now the  $p > 2$  cases also fail to converge (not shown); the reasons for this are not evident.

In fact our difficulties will largely be fixed by a better initial iterate method, namely by the application of grid sequencing, and only the  $p = 1.1$  case will remain unresolved. The conclusion we may draw from this situation, demonstrated next, is that the Hessian becomes singular if the Newton iteration is started too far from the solution.

## Convergence and (near) optimality

Recall that for DM-based codes applied to nonlinear PDEs, the `-snes_grid_sequence` technique introduced in Chapter 7 generates high-quality initial iterates by interpolating a converged solution from a coarser grid. When combined with a multigrid-preconditioned Newton-Krylov method, such grid sequencing is a nonlinear multigrid “full-cycle” solver; compare Figure 6.14. Such a full-cycle solver works upward from coarse grids to fine grids, with coarse-grid

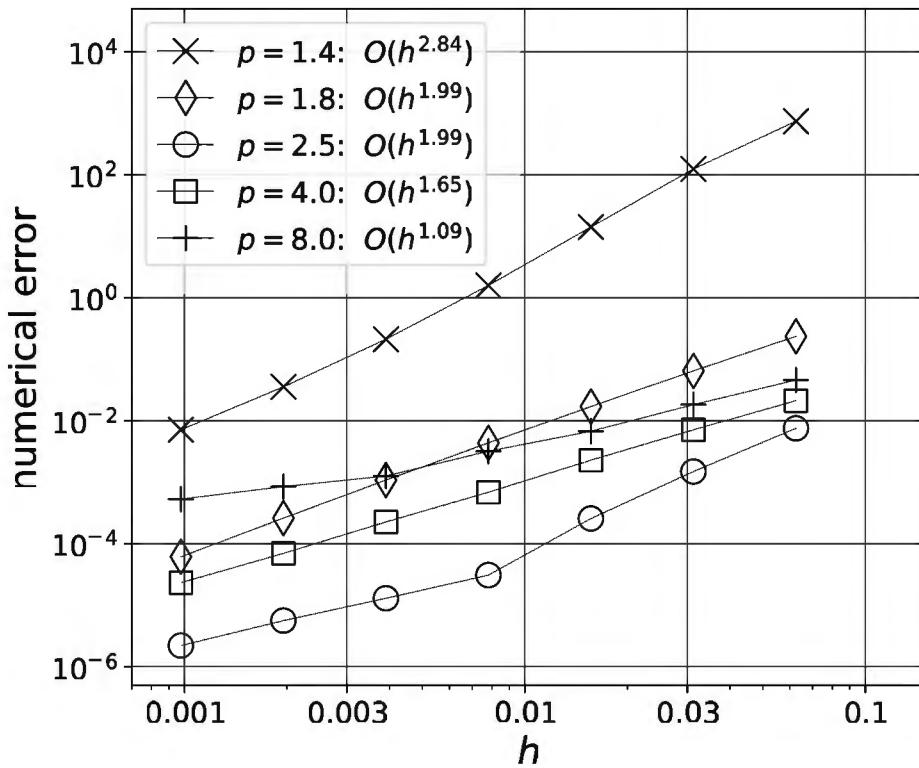


Figure 9.9. Numerical errors  $\|u - u_{\text{exact}}\|_\infty$  for some  $p \neq 2$  cases.

corrections occurring within the preconditioner during the solution of the linear Newton-step equations. As we will see, this solution strategy succeeds for  $p$  in a substantial range around  $p = 2$ . In fact, as with the minimal surface equation in Chapter 7, grid sequencing seems to be necessary for solving nonlinear  $p$ -Helmholtz problems, although cases near the extremes  $p = 1$  and  $p = \infty$  will remain problematical.

We can demonstrate convergence, and near-optimal solver complexity (Chapter 7), for non-extreme  $p$ . Consider these runs:

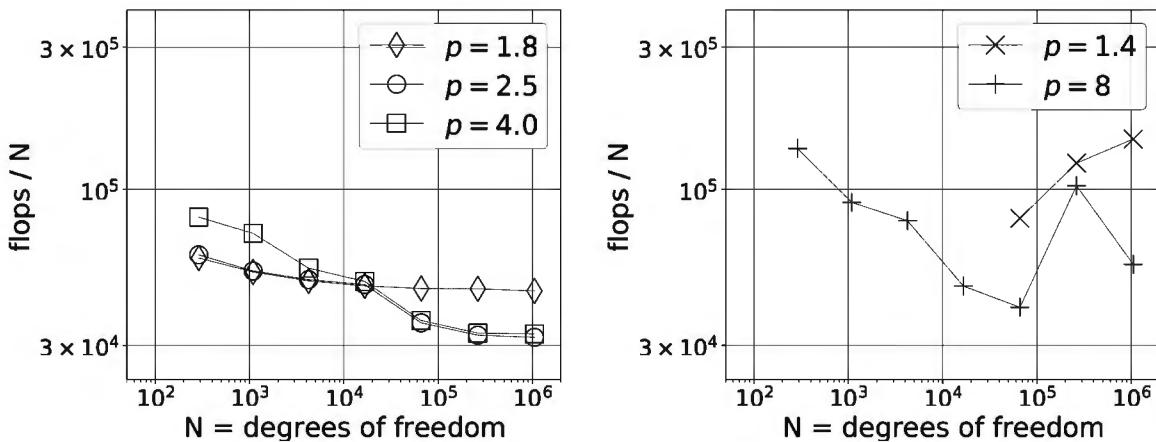
```
./phelm -snes_rtol 1.0e-5 -ph_eps 1.0e-4 -ksp_type cg -pc_type mg \
-ph_p P -snes_grid_sequence LEV
```

for  $p = 1.4, 1.8, 2.5, 4, 8$  and  $\text{LEV} = 4, 5, 6, 7, 8, 9, 10$ . The finest  $1025 \times 1025$  grid has  $N > 10^6$  degrees of freedom. We add `-snes_converged_reason` to count Newton iterations and `-log_view` to get the total flops.

The resulting numerical errors are shown in Figure 9.9. For exponents closest to the linear case ( $p = 1.8, 2.5$ ) we see the expected  $O(h^2)$  convergence rate. For the larger  $p > 2$  values ( $p = 4, 8$ ) the tendency of the diffusivity to degenerate reduces the rate to near  $O(h^1)$ ; as mentioned earlier, diffusion is reduced in the level-curve directions. For the  $p = 1.4$  case note that  $\|u - u_{\text{exact}}\|_\infty > 1$  on the coarse grids, i.e., there are no digits of accuracy.

We have chosen a GMG-based solver based on its performance on Poisson problems. As shown in Figure 9.10, the cost of the full-cycle nonlinear solver using `-snes_grid_sequence` appears to scale as  $O(N)$  for exponents in a range around  $p = 2$ . That is, the work per degree of freedom is roughly constant for  $p = 1.8, 2.5, 4$ . However, the  $p = 8$  case requires significantly more work and the trend is not clear on fine grids. Likewise for  $p = 1.4$ , noting we exclude grids where  $\|u - u_{\text{exact}}\|_\infty > 1$ , we see that the cost per degree of freedom may actually be increasing.

The solver scales well in parallel. For example, the following strong-scaling (Chapter 8) runs solve the 4-Helmholtz equation on a  $1025 \times 1025$  grid with  $P = 1, 4, 16, 64$  processes:



**Figure 9.10.** Flops per degree of freedom for  $p = 1.8, 2.5, 4$  (left) and  $p = 1.4, 8$  (right).

```
$ mpiexec -n P ./phelm -ph_p 4.0 -ph_eps 1.0e-4 -snes_rtol 1.0e-5 \
-snes_converged_reason -ksp_type cg -pc_type mg \
-da_grid_x 9 -da_grid_y 9 -snes_grid_sequence 7 -log_view
```

In this redundant form of parallel multigrid the coarsest grid must be sufficiently fine to give at least one degree of freedom per process (Chapter 7), thus the choice of a  $9 \times 9$  coarsest grid. The result is excellent, with the total number of flops varying by only 2% from  $P = 1$  to  $P = 64$  processes, and a maximum load imbalance of 4% on 64 processes.

In summary, we have used the  $p$ -Helmholtz problem to explore a number of ideas:

- the relationship between optimization and PDEs,
- implementation of a structured-grid  $Q_1$  finite element method,
- direct numerical optimization using only an objective function,
- quasi-Newton and nonlinear conjugate gradients methods,
- regularization of a nonlinear operator to improve Newton search directions,
- and nonlinear full-cycle multigrid methods, namely grid sequencing combined with a multigrid-preconditioned Newton-Krylov solver.

Our initial objective-only approach is actually limited to a prototyping role. However, note that implementing an objective function has the nice side effect that line search uses the correct objective, as opposed to a merit function (Chapter 4). In any case, a gradient (residual) evaluation function is certainly required for high-resolution solutions, and regularization is a helpful technique when facing strong nonlinearities.

## Exercises

- 9.1. Figure 9.1 shows the graph of  $\Phi(x, y) = \frac{1}{4}(x^4 + y^4) - 2x + 2y$  on  $[-5, 5]^2$ , a “cartoon” of the  $p$ -Helmholtz functional  $I[u]$  in the case  $p = 4$ . Find the unique minimum of the objective function  $\Phi$  and then write a code `cartoon.c` which only implements `FormObjective()` to solve  $\min_{x,y} \Phi(x, y)$ . Choose an initial iterate with attention to nonsingularity of the Hessian. After checking that the solver works using

`-snes_fd_function -snes_fd|mf`, add `FormFunction()` which computes the gradient; now `-snes_fd_function` is not needed. Compare numbers of evaluations and numerical error. (*Hints.* Your code will not use a DMDA; there is no grid. Consider `ch4/expcircle.c` as a starting point.)

- 9.2. Assume  $\Omega$  is a bounded domain and that  $f \in L^q(\Omega)$  for  $\frac{1}{q} + \frac{1}{p} = 1$ .
  - (a) For  $p = 2$ , prove coercivity (9.3) of functional  $I[u]$  on  $W^{1,2}(\Omega)$ . (*Hint.* The Cauchy-Schwarz inequality can be applied to the  $\int f u$  term.)
  - (b) For  $1 \leq p \leq 2$ , prove coercivity (9.3) of functional  $I[u]$  on  $W^{1,p}(\Omega)$ . (*Hints.* Jensen's inequality applied to the convex function  $g(x) = |x|^{2/p}$  shows that  $\|u\|_{L^2} \geq C\|u\|_{L^p}$  for  $C > 0$ . When  $f = 0$  one may now argue by contradiction: if  $\|u_n\|_{W^{1,p}} \rightarrow \infty$  and  $I[u_n] \not\rightarrow \infty$  then there is a subsequence where  $I[u_n]$  is bounded. And so on. Complete a square to extend to all  $f$ .)
  - (c) For  $1 < p < \infty$ , prove strict convexity (9.4) of functional  $I[u]$  on  $W^{1,p}(\Omega)$ . (*Hint.* See section 5.3 of [36].)
- 9.3. The weak formulation (9.7) of the  $p$ -Helmholtz problem implicitly includes homogeneous Neumann boundary conditions, but a boundary integral appears in the nonhomogeneous case, as follows. Suppose  $g \in L^q(\partial\Omega)$  and suppose  $u$  satisfies

$$\int_{\Omega} |\nabla u|^{p-2} \nabla u \cdot \nabla v + uv - fv = \int_{\partial\Omega} gv \quad (9.30)$$

for all  $v \in W^{1,p}(\Omega)$ , where  $p^{-1} + q^{-1} = 1$ . (A trace theorem [50] explains the assumed regularity for  $g$ .) By following the argument which derives strong form (9.9), show that if  $u$  is sufficiently smooth and satisfies (9.30) then

$$-\nabla \cdot (|\nabla u|^{p-2} \nabla u) + u = f \text{ on } \Omega \quad \text{and} \quad |\nabla u|^{p-2} \frac{\partial u}{\partial n} = g \text{ on } \partial\Omega.$$

- 9.4. Assume  $u$  is smooth and that the formulas below apply where  $|\nabla u| \neq 0$ . Recall  $\Delta_p u = \nabla \cdot (|\nabla u|^{p-2} \nabla u)$  denotes the  $p$ -Laplacian. Define  $\hat{\Delta}_{\infty} u = |\nabla u|^{-2} (u_x^2 u_{xx} + 2u_x u_y u_{xy} + u_y^2 u_{yy})$  and  $\hat{\Delta}_1 = \Delta_2 - \hat{\Delta}_{\infty}$ . For  $1 < p < \infty$  show that [124]

$$\Delta_p u = p |\nabla u|^{p-2} \left( \frac{1}{p} \hat{\Delta}_1 + \frac{1}{q} \hat{\Delta}_{\infty} \right) u$$

where  $q$  is the conjugate exponent to  $p$  satisfying  $p^{-1} + q^{-1} = 1$ . (*This justifies the name “ $\infty$ -Laplacian” for  $\hat{\Delta}_{\infty}$ , as a rescaled  $p \rightarrow \infty$  limit, and it gives  $\Delta_p$  as an interpolant between the extremes.*)

- 9.5. Use (9.13) and (9.14) to confirm (9.16). Then use (9.14), (9.16), and (9.20) to derive (9.21).
- 9.6. Show that (9.23) is, in detail,

$$\begin{aligned} G_{ij}(u, \xi, \eta) &= \frac{1}{p} \left[ \frac{4}{h_x^2} \left( \sum_{\ell=0}^3 u_{\ell} \frac{\partial \chi_{\ell}}{\partial \xi} \right)^2 + \frac{4}{h_y^2} \left( \sum_{\ell=0}^3 u_{\ell} \frac{\partial \chi_{\ell}}{\partial \eta} \right)^2 \right]^{p/2} \\ &\quad + \frac{1}{2} \left( \sum_{\ell=0}^3 u_{\ell} \chi_{\ell} \right)^2 - \left( \sum_{\ell=0}^3 f_{\ell} \chi_{\ell} \right) \left( \sum_{\ell=0}^3 u_{\ell} \chi_{\ell} \right), \end{aligned}$$

where  $u_{\ell}$  and  $f_{\ell}$  are local-node-indexed values on element  $\square_{ij}$ .

- 9.7. In the case where the objective is quadratic there are two ways to use classical conjugate gradient (CG) optimization [118] in PETSC. For illustration, generate two classical CG runs based on these common options:

```
| $ ./phelm -ph_p 2.0 -da_refine 2
```

One run should add `-snes_type ncg`. The other should use `-snes_type ksponly` and `-ksp_type cg`. Use monitoring to show that residual norms are the same up to rounding errors.

- 9.8. Show that for local node index  $L = 0, 1, 2, 3$ , (9.27) is, in detail,

$$\begin{aligned} H_{ij}^L(u, \xi, \eta) = & \left[ \frac{4}{h_x^2} \left( \sum_{\ell=0}^3 u_\ell \frac{\partial \chi_\ell}{\partial \xi} \right)^2 + \frac{4}{h_y^2} \left( \sum_{\ell=0}^3 u_\ell \frac{\partial \chi_\ell}{\partial \eta} \right)^2 \right]^{(p-2)/2} \\ & \cdot \left[ \frac{4}{h_x^2} \left( \sum_{\ell=0}^3 u_\ell \frac{\partial \chi_\ell}{\partial \xi} \right) \frac{\partial \chi_L}{\partial \xi} + \frac{4}{h_y^2} \left( \sum_{\ell=0}^3 u_\ell \frac{\partial \chi_\ell}{\partial \eta} \right) \frac{\partial \chi_L}{\partial \eta} \right] \\ & + \left( \sum_{\ell=0}^3 u_\ell \chi_\ell - \sum_{\ell=0}^3 f_\ell \chi_\ell \right) \chi_L. \end{aligned}$$

- 9.9. Our  $p$ -Helmholtz problem has a zeroth-order term and Neumann boundary conditions. A related  $p$ -Laplacian problem considers the functional

$$J[u] = \int_{\Omega} \frac{1}{p} |\nabla u|^p - fu$$

over the affine space  $W_g^{1,p}(\Omega)$  with Dirichlet boundary conditions  $g$  on  $\partial\Omega$ ; here the strong-form PDE  $-\Delta_p u = f$  generalizes the Poisson equation. However, in the discretize-and-minimize approach of this chapter, the implementation of Dirichlet boundary conditions is more subtle than Neumann conditions. The difficulty is that if the boundary nodes are treated as unknowns, as we have previously, then  $J[u]$  does not actually depend on these values. With a residual-evaluation approach we would solve trivial equations at boundary points, so a gradient function is easy to implement, but it is less obvious how to construct the functional  $J[u]$ . There are at least three approaches to constructing the objective function:

- (a) *Interior grid only*. Use the DMDA in an unusual way by having the grid extend only from  $h$  to  $1-h$  in each direction. Thus all degrees of freedom are interior points. This can be made to work but, because interpolation to a finer grid involves extrapolation using  $g$  values, it precludes GMG preconditioning without additional intervention in the interpolation/restriction operators.
- (b) *Penalize the boundary*. Add quadratic terms to the discrete objective function which correspond to  $\int_{\partial\Omega} \frac{\mu}{2} (u - g)^2$ . Care must be taken with the scaling parameter  $\mu > 0$ .
- (c) *Lagrange multipliers*. Use Lagrange multipliers to extend the objective to any  $u \in W^{1,p}(\Omega)$  by adding the constraint  $u = g$  at each boundary grid location. This generates a saddle-point objective  $J[u, \lambda]$  with an indefinite gradient operator; compare the Stokes problem in Chapter 14.

Write and test a code for one of these possibilities.

- 9.10. One may approximate PDE (9.9) by structured-grid FD methods as in previous chapters. Construct such an FD code `phelmd.c`, perhaps based on `ch7/minimal.c`, which implements the corresponding gradient-evaluation function. Handling the coefficient  $|\nabla u|^{p-2}$

in a consistent [115] manner requires thought, but note how the minimal surface equation in Chapter 7 is discretized. Use the same manufactured solution, namely (9.10). Compare convergence and performance. Comment on implementation differences, including programmer time.

- 9.11. In this exercise we address how the Jacobian of the weak form would be implemented. (*An actual implementation may not be justified.*)

- (a) Let  $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^d$  be defined as

$$\phi(X) = (X \cdot X)^{(p-2)/2} X. \quad (9.31)$$

Note that  $\phi(\nabla u)$  defines the expression “ $|\nabla u|^{p-2}\nabla u$ ” in (9.7). For each nonzero  $X \in \mathbb{R}^d$ , and any  $Z \in \mathbb{R}^d$ , show that

$$\lim_{\epsilon \rightarrow 0} \frac{\phi(X + \epsilon Z) - \phi(X)}{\epsilon} = |X|^{p-2}Z + (p-2)|X|^{p-4}X(X \cdot Z).$$

This defines a derivative map which, for  $X \neq 0$ , is linear in  $Z \in \mathbb{R}^d$ :

$$\phi'(X)[Z] = |X|^{p-2}Z + (p-2)|X|^{p-4}X(X \cdot Z).$$

- (b) (*Ever since Chapter 4, “ $\mathbf{F}(\mathbf{x})$ ” has denoted the finite-dimensional residual function, and “ $F_i(\mathbf{x})$ ” one of its components. It inspires the notation used here.*) For  $u, v \in W^{1,p}(\Omega)$ , the  $p$ -Laplacian weak form (9.7) is a residual function:

$$F_v(u) = \int_{\Omega} \phi(\nabla u) \cdot \nabla v + uv - fv,$$

a scalar, where the test function  $v$  plays the role of an index. For  $w \in W^{1,p}(\Omega)$  show that a scalar “entry” of the Jacobian is

$$J_{vw}(u) = \lim_{\epsilon \rightarrow 0} \frac{F_v(u + \epsilon w) - F_v(u)}{\epsilon} = \int_{\Omega} \phi'(\nabla u)[w] \cdot \nabla v + vw. \quad (9.32)$$

- (c) Verify that if  $d = 1$  then (9.32), with (9.31), simplifies to

$$J_{vw}(u) = \int_{\Omega} (p-1)|u'|^{p-2}v'w' + vw.$$

- (d) For  $p = 2$ , and any  $d$ , check that (9.32) includes the expected formula for an entry of the Laplacian stiffness matrix (see, e.g., [49, equation (1.22)]) in the case where  $v = \psi_{pq}$  and  $w = \psi_{rs}$ .

- (e) Starting from (9.32), show that  $J_{vw}(u) = J_{wv}(u)$ .

- (f) Write a pseudocode for `FormJacobianLocal()` in `phelm.c`.

- 9.12. One may linearize the  $p$ -Helmholtz equation (9.9) as a *Picard* iteration. This approach, discussed further in Chapter 10, reuses linear-case code while avoiding an actual Jacobian implementation.

- (a) If  $u^{(k)}$  is the  $k$ th iterate then, given  $u^{(k-1)}$ , suppose we solve

$$-\nabla \cdot (|\nabla u^{(k-1)}|^{p-2} \nabla u^{(k)}) + u^{(k)} = f \quad (9.33)$$

for  $u^{(k)}$ . Note (9.33) is a linear equation  $A(\mathbf{u}^{(k-1)})\mathbf{u}^{(k)} = \mathbf{b}$  in which the matrix changes at each iteration. Implement this method by adding `FormPicardLocal()`,

which computes and assembles a `Mat` for the left side, to `phelm.c`; this is the true Jacobian (Exercise 9.11) only when  $p = 2$ . Use `DMDASNESSetJacobianLocal()` to set the call-back and `MatSetValuesStencil()` to insert entries into the `Mat` (Chapter 3).

- (b) Confirm (`-log_view | grep Eval`) that the number of evaluations of `FormFunctionLocal()` per iteration is now smaller than with `-snes_fd_color`, but note quadratic convergence is lost. Recover quadratic convergence for  $p \neq 2$  using option `-snes_mf_operator`, which uses the Picard matrix for preconditioning (Chapter 4).
- (c) By also using `-snes_grid_sequence` and `-ksp_type cg -pc_type mg`, for which  $p > 2$  can you demonstrate near-optimal solver complexity?  $p < 2$ ?

## Chapter 10

# Finite element method II: Naive and unstructured

In this chapter we implement a finite element (FE) method for certain nonlinear Poisson equations on an unstructured triangulation of a 2D domain. Relative to the structured-grid  $Q_1$  method of the last chapter, we face new tasks including reading a mesh from a file and managing element and node indices in an unstructured manner. However, the construction of the nonlinear system  $\mathbf{F}(\mathbf{u}) = \mathbf{0}$ , from the PDE weak form, follows the same SNES-based approach as in Chapters 4–9. Our code simply sums over elements to evaluate the residual  $\mathbf{F}$ , so no user-written matrix-assembly procedure is initially needed. After verifying this implementation using a finite differenced Jacobian we write additional code to assemble a matrix, the exact Jacobian in the linear case. For nonlinear problems we either (Picard) iterate using this matrix, as an approximate Jacobian, or we use the matrix only in the preconditioner of a Jacobian-free Newton-Krylov method. In either case, preallocating this `Mat` is *essential* for performance.

However, a substantial programmer workload comes from the naive design of our mesh infrastructure. Furthermore, because the infrastructure is incomplete, solver choices are limited compared to DM-based discretizations. For instance, instead of solving the equations using geometric multigrid (GMG), using grid-based interpolation and restriction operators such as those from a DMDA (Chapters 6–9), we introduce and apply an algebraic multigrid (AMG) solver instead. Also, our code only works in serial. In fact, the benefits of using a PETSC DM mesh topology/geometry type become clear by their absence here. Therefore, in preparation for using the DMFlex type (Chapter 13), at the end of the current chapter we also discuss how FE mesh and assembly operations would be distributed across MPI processes.

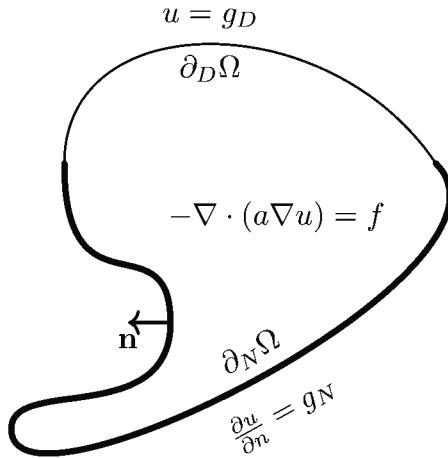
## A nonlinear Poisson problem

Let  $\Omega \subset \mathbb{R}^2$  be a bounded (open) region as in Figure 10.1. We suppose the boundary  $\partial\Omega$  is well-behaved, e.g., polygonal or Lipschitz continuous [36], and is decomposed into disjoint, measurable subsets  $\partial\Omega = \partial_D\Omega \cup \partial_N\Omega$ . Let  $a(u, x, y)$  and  $f(u, x, y)$  be given continuous functions and assume there is  $\epsilon$  so that

$$a(u, x, y) \geq \epsilon > 0, \quad (10.1)$$

that is, assume uniform ellipticity [51, 60]. Defining  $\partial u / \partial n = \mathbf{n} \cdot \nabla u$ , where  $\mathbf{n}$  is the outward unit normal on  $\partial\Omega$  (Figure 10.1), we solve the following nonlinear (quasilinear) Poisson problem:

$$\begin{aligned} -\nabla \cdot (a(u)\nabla u) &= f(u) && \text{on } \Omega, \\ u &= g_D && \text{on } \partial_D\Omega, \\ a(u)\frac{\partial u}{\partial n} &= g_N && \text{on } \partial_N\Omega. \end{aligned} \quad (10.2)$$



**Figure 10.1.** Problem (10.2) on domain  $\Omega$ ; Neumann boundary in bold.

The data of the problem include the *diffusion coefficient*  $a$ , *source term*  $f$ , *Dirichlet data*  $g_D$ , and *Neumann data*  $g_N$ . The last two functions, which we assume are continuous and independent of  $u$ , may be defined only on  $\partial_D\Omega$ ,  $\partial_N\Omega$ , respectively.

More general boundary conditions are possible, for instance *Robin* conditions of the form  $\alpha u + \beta \frac{\partial u}{\partial n} = \gamma$  [49] (Exercise 10.11). One might also allow  $a$  or  $f$  to depend on the gradient of  $u$ —e.g.  $a = |\nabla u|^{p-2}$  in the  $p$ -Laplacian equation of Chapter 9—but this is not done here.

Strong form problem (10.2) may have no solution wherein “ $\nabla \cdot (a\nabla u)$ ” makes sense as a continuous function, even for polygonal regions and continuous data. (There may be no  $u \in C^2(\Omega) \cap C(\bar{\Omega})$  satisfying (10.2) at all points.) However, once we convert (10.2) to weak form then, at least in the linear case, a solution exists (see below). On the other hand, while the linear Poisson problem comes from minimizing an objective function, the general nonlinear form (10.2) has no optimization formulation; compare the problem in Chapter 9 and see Exercise 10.1. Thus we derive the weak form from (10.2) simply by multiplying this equation by a test function and integrating by parts, as follows.

Consider two subsets of  $W^{1,2}(\Omega)$ ,<sup>30</sup> namely *trial functions*  $W_g^{1,2}(\Omega)$  and *test functions*  $W_0^{1,2}(\Omega)$ , with value  $g_D$  and zero, respectively, along  $\partial_D\Omega$ . We multiply (10.2) by a test function  $v$  and integrate by parts:

$$\int_{\Omega} a(u)\nabla u \cdot \nabla v - \int_{\partial\Omega} a(u)\frac{\partial u}{\partial n}v = \int_{\Omega} f(u)v.$$

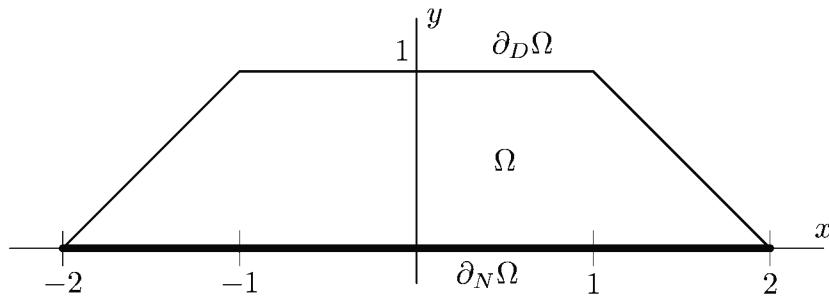
Now apply boundary conditions  $v = 0$  on  $\partial_D\Omega$  and  $a(u)\partial u/\partial n = g_N$  on  $\partial_N\Omega$ :

$$\int_{\Omega} a(u)\nabla u \cdot \nabla v = \int_{\Omega} f(u)v + \int_{\partial_N\Omega} g_N v. \quad (10.3)$$

Observe that the Dirichlet data  $g_D$  are used in defining  $W_g^{1,2}(\Omega)$  while the Neumann data  $g_N$  appear explicitly in (10.3).

We seek  $u \in W_g^{1,2}(\Omega)$  satisfying the weak formulation (10.3). The above derivation shows that a well behaved function  $u \in C^2(\Omega) \cap C(\bar{\Omega})$  which satisfies (10.2) also solves (10.3). On the other hand, if  $u \in W_g^{1,2}(\Omega)$  solves (10.3) then we accept it, by definition, as a solution. If  $u$  is well-behaved enough to reverse the derivation then it will also solve (10.2). For the linear case, where functions  $a$  and  $f$  are independent of  $u$ , and if  $\partial_D\Omega$  has positive measure, then a

<sup>30</sup>Recall that  $W^{1,2}(\Omega)$  is the Hilbert space of functions with square-integrable gradients; see definition (9.2).



**Figure 10.2.** Our exact solutions solve cases of (10.2) on this trapezoid. For cases 0 and 1 the Neumann boundary is as shown in bold.

unique solution to (10.3) exists [36, 51]. Furthermore there exist conditions on the domain (e.g., a convex polygon) and the boundary data so that  $u$  solving (10.3) is in  $C^2(\Omega) \cap C(\bar{\Omega})$  [51].

In terms of practical computation, the code `c/ch10/unfem.c`, described in this chapter, will solve certain nonlinear Poisson equations on arbitrary 2D domains. Interesting cases like the following are solvable:

- *Liouville-Bratu* equations (Exercise 7.12), where  $a \equiv 1$  and  $f = \lambda e^u$  (see Exercises 10.9 and 10.10), and
- *porous medium* equations [119], in which  $a = \epsilon + u^{m-1}$  for some  $m \geq 1$  and  $\epsilon > 0$ .

As usual, we will need simple problems with known exact solutions for testing the implementation. Ideally the measured numerical errors will converge at the theoretical rate  $O(h^2)$  [49], and only converge at that rate, if our implementation is correct. The three cases we propose, all implemented in `c/ch10/cases.h` (not shown), are based on the same “manufactured” solution:

$$u_{\text{exact}}(x, y) = 1 - xy^2 - \frac{1}{4}y^4. \quad (10.4)$$

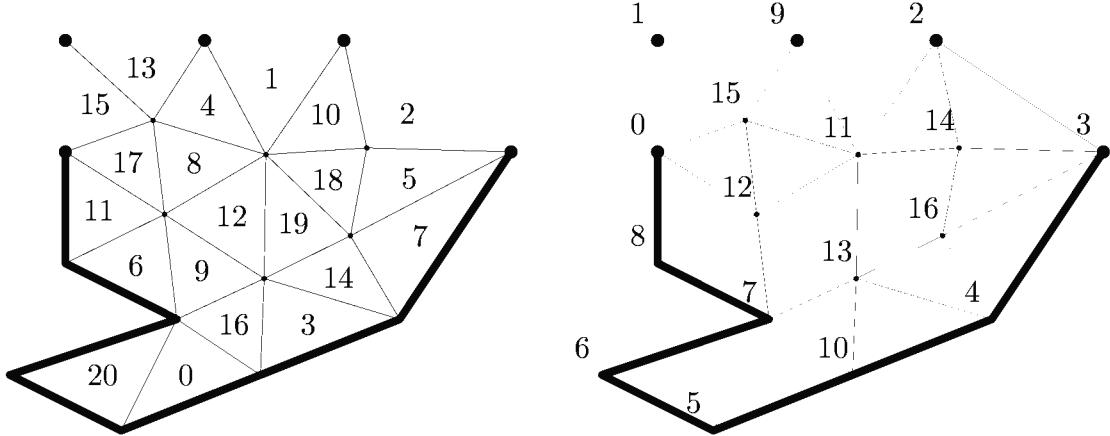
Case 0: *Linear*. Use the domain and boundary decomposition shown in Figure 10.2. Define  $g_D$  as the value of  $u_{\text{exact}}$  along  $\partial_D\Omega$ . Noting that  $u_{\text{exact}}$  has zero normal derivative on the Neumann boundary  $\partial_N\Omega = \{y = 0\}$ , let  $g_N = 0$ . Let  $a = 1$  and determine  $f$  by differentiating the exact solution (thus  $f(x, y) = 2x + 3y^2$ ).

Case 1: *Nonlinear (porous medium type)*. Use the same domain and boundary conditions as in case 0 but let  $a(u) = 1 + u^2$  and determine  $f(x, y)$  by differentiation.

Case 2: *Linear, with nonhomogeneous Neumann boundary conditions*. The domain is the same as in Figure 10.2 and  $a, f$  are the same as in case 0. However, the Neumann boundary  $\partial_N\Omega$  is the line segment from  $(2, 0)$  to  $(1, 1)$ , and the (nonzero) value  $g_N$  is found by differentiating  $u_{\text{exact}}$  along this segment.

## Unstructured $P_1$ finite elements

Our method will find an approximate solution  $u^h$ , from a finite-dimensional (affine) subspace of trial functions, by satisfying (10.3) for all test functions. These two subspaces are built from a triangulation of  $\Omega$  by using the same local functions, so this is a *Galerkin* method [49]. For simplicity we assume that  $\Omega$  is polygonal,  $\partial_D\Omega$  is closed and nonempty, and the segments forming the polygon  $\partial\Omega$  are each either in  $\partial_D\Omega$  or in  $\overline{\partial_N\Omega}$ .



**Figure 10.3.** A triangulation  $\mathcal{T}_h$  of a polygon with  $K = 21$  elements (left),  $N = 17$  nodes (right),  $n_D = 5$  nodes in the Dirichlet boundary (dots), and  $P = 7$  segments in the Neumann boundary (bold).

By definition, a *triangulation* is a finite set of nonoverlapping, nonempty open triangles  $\Delta_k$  whose closures tile  $\Omega$ :

$$\mathcal{T}_h = \left\{ \Delta_k \mid \cup_k \bar{\Delta}_k = \bar{\Omega} \text{ and } \Delta_k \cap \Delta_l = \emptyset \text{ if } k \neq l \right\}. \quad (10.5)$$

An example is shown in Figure 10.3. The subscript “ $h$ ” denotes the maximum diameter of the triangles. (It also serves as a reminder of our desired limit  $h \rightarrow 0$ .) In contrast to many standard references (e.g., [49]), numbering here is zero-based, appropriate for a C implementation, so triangles  $\Delta_k \in \mathcal{T}_h$  are indexed  $k = 0, \dots, K - 1$  while nodes are  $\mathbf{x}_i = (x_i, y_i)$  for  $i = 0, \dots, N - 1$ .

In the  $P_1$  FE method used here, functions are linear on each  $\Delta_k$ . To be in  $W^{1,2}(\Omega)$  such functions must also be continuous on  $\Omega$ , and then each function is determined by its values at the  $N$  nodes [19, 49].

A linear function  $a + bx + cy$  on a triangle has three degrees of freedom, but there are more convenient bases than  $\{1, x, y\}$ . Specifically, consider linear functions on  $\Delta_k$  which are nonzero at only one node. For each node  $j$  in  $\mathcal{T}_h$  let  $\psi_j(x, y)$  denote the “hat” function which is linear on each triangle, continuous on  $\bar{\Omega}$ , and satisfies  $\psi_j(\mathbf{x}_i) = \delta_{ij}$  (Figure 10.4; compare Figure 9.4). Note that the set  $\{\psi_j\}_{j=0}^{N-1}$  is linearly independent, and that the partial derivatives of  $\psi_j(x, y)$  are piecewise constant (and not continuous).

Hat functions allow us to extend the Dirichlet data  $g_D$  to all of  $\bar{\Omega}$ . First number the nodes  $\mathbf{x}_{j_l} \in \partial_D \Omega$  by  $l = 0, \dots, n_D - 1$ . Then define  $\hat{g}_D \in C(\bar{\Omega})$  as the piecewise-linear interpolant of  $g_D$  having value zero at all the nodes  $\mathbf{x}_j$  which are *not* in the Dirichlet boundary  $\partial_D \Omega$ ; expanding in hat functions yields

$$\hat{g}_D(x, y) = \sum_{l=0}^{n_D-1} g_D(\mathbf{x}_{j_l}) \psi_{j_l}(x, y). \quad (10.6)$$

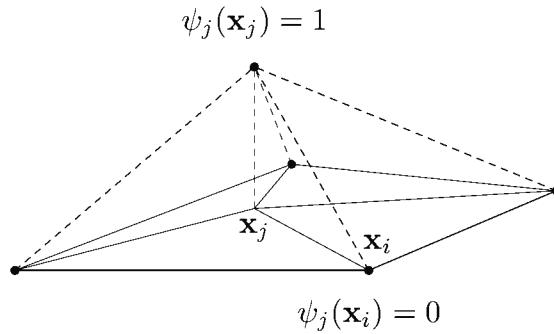
We can now define the FE subspaces of  $W^{1,2}(\Omega)$ . The *test functions* are in

$$S_0^h = \text{span} \{ \psi_j : \mathbf{x}_j \notin \partial_D \Omega \}. \quad (10.7)$$

If  $v \in S_0^h$  then  $v = 0$  on  $\partial_D \Omega$ . The *trial functions* have value  $g_D$  along  $\partial_D \Omega$ ,

$$S_g^h = \{ \hat{g}_D + w : w \in S_0^h \}. \quad (10.8)$$

Note that  $S_0^h$  is a linear subspace of  $W_0^{1,2}(\Omega)$ , while  $S_g^h$  is an affine subspace of  $W^{1,2}(\Omega)$ , but  $\dim(S_0^h) = \dim(S_g^h) = N - n_D$ .



**Figure 10.4.** A hat function  $\psi_j$ .

Our FE method seeks  $u^h \in S_g^h$  such that (10.3) holds for all  $v^h \in S_0^h$ . By linearity it suffices to test only a basis from  $S_0^h$ , so we require

$$\int_{\Omega} a(u^h) \nabla u^h \cdot \nabla \psi_i = \int_{\Omega} f(u^h) \psi_i + \int_{\partial_N \Omega} g_N \psi_i \quad (10.9)$$

for all  $i$  such that  $x_i \notin \partial_D \Omega$ . On the other hand we may expand the trial function  $u^h$  using  $N - n_D$  unknown coefficients  $u_j \in \mathbb{R}$ ,

$$u^h(x, y) = \hat{g}_D(x, y) + \sum_{x_j \notin \partial_D \Omega} u_j \psi_j(x, y). \quad (10.10)$$

The coefficients  $u_j$  in (10.10) are the unknowns in this FE method.

Given a triangulation  $\mathcal{T}_h$ , and the data  $a$ ,  $f$ ,  $g_D$ , and  $g_N$ , the FE solution  $u^h$  is completely specified by equations (10.6), (10.9), and (10.10). This finite-dimensional problem can be shown to be well posed by the same theory that applies to the continuum problem [49].

In the linear case we may write system (10.9) and (10.10) as  $A\mathbf{u} = \mathbf{b}$  where  $A$  is the *stiffness matrix* [19, 49]. While FE codes often assemble this matrix as their first goal, we will not write code to construct such a matrix until *after* we have a verified numerical solution. In fact, following the pattern since Chapter 4, we implement (10.9) by constructing a residual function  $\mathbf{F}(\mathbf{u})$  in a SNES call-back. Here the input  $\mathbf{u}$  is the representation of  $u^h$  as a vector of nodal values. We use (10.10) to get point values of  $u^h$  and  $\nabla u^h$ —namely at the quadrature points—so as to approximate the integrals in (10.9). In the linear case implementing such a residual  $\mathbf{F}(\mathbf{u})$  is nearly equivalent to assembling  $A$  and  $\mathbf{b}$ , but writing residual-evaluation code requires no direct contact with a `Mat` object, and our design is quite insensitive to whether the problem is linear or not. After our solution is tested for correctness, via a finite differenced Jacobian (Chapter 4), we will reconsider preallocating and constructing a Jacobian matrix for  $\mathbf{F}$ .

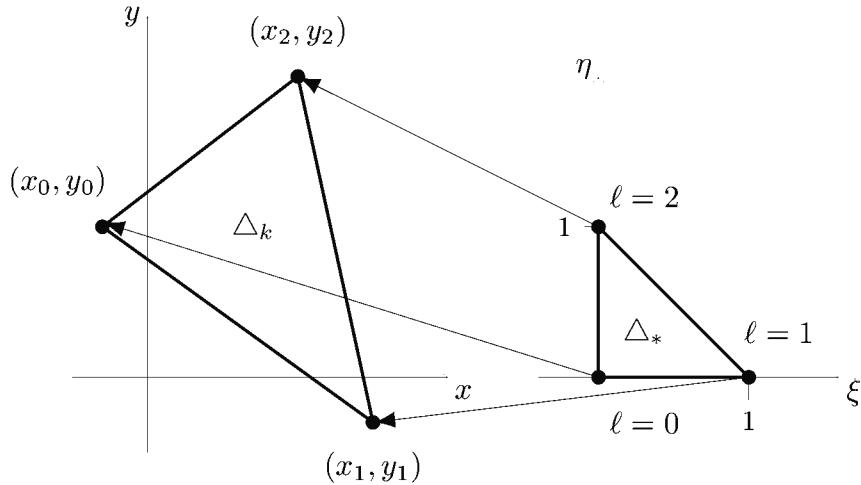
## Assembly of the residual equations

Integrals over  $\Omega$  may be written as sums of integrals over elements. Thus for each triangle  $\Delta_k$  and hat function  $\psi_i$  we define an *element residual*

$$F_i^k(\mathbf{u}) = \int_{\Delta_k} a(u^h) \nabla u^h \cdot \nabla \psi_i - f(u^h) \psi_i. \quad (10.11)$$

For each segment (edge)  $s_\nu$  in the Neumann boundary we likewise define

$$\varphi_i^\nu = \int_{s_\nu} g_N \psi_i. \quad (10.12)$$



**Figure 10.5.** Formula (10.17) maps from the reference triangle  $\Delta_*$  to  $\Delta_k$ .

(Note  $\varphi_i^\nu$  does not depend on the solution  $u^h$ .) Numbering the Neumann segments by  $\nu = 0, \dots, P - 1$ , weak form (10.9) then becomes the statement

$$F_i(\mathbf{u}) = \sum_{k=0}^{K-1} F_i^k(\mathbf{u}) - \sum_{\nu=0}^{P-1} \varphi_i^\nu = 0 \quad \text{if } \mathbf{x}_i \notin \partial_D \Omega. \quad (10.13)$$

The implementation is a bit easier if we also increase the size of the system by including the nodes in the Dirichlet boundary as unknowns, thus

$$\mathbf{u} = \{u_j\}_{j=0}^{N-1} \in \mathbb{R}^N. \quad (10.14)$$

To do so we define trivial residuals for the nodes in the Dirichlet boundary:

$$F_i(\mathbf{u}) = u_i - g_D(\mathbf{x}_i) \quad \text{if } \mathbf{x}_i \in \partial_D \Omega. \quad (10.15)$$

Observe that nodes in the Dirichlet boundary become degrees of freedom while segments in the Neumann boundary are used as domains of integration.

Together, equations (10.13) and (10.15) generate the nonlinear system

$$\mathbf{F}(\mathbf{u}) = \mathbf{0}, \quad (10.16)$$

with  $\mathbf{F} : \mathbb{R}^N \rightarrow \mathbb{R}^N$ . We will ask SNES to solve this problem. System (10.16) is sparse because the support of each hat function  $\psi_i$  only overlaps with a few triangles  $\Delta_k$  and boundary segments  $s_\nu$ , and, on the other hand, only a few nodal values  $\mathbf{u} = \{u_j\}$  enter into any given element residual  $F_i^k(\mathbf{u})$ . (Representative Jacobian sparsity patterns appear later in Figure 10.9.)

We compute the element residuals (10.11) by referring  $\Delta_k$  to a reference triangle  $\Delta_*$ , shown in Figure 10.5, namely  $\Delta_* = \{(\xi, \eta) : \xi \geq 0, \eta \geq 0, \xi + \eta \leq 1\}$ . If  $\Delta_k$  has vertices  $(x_0, y_0)$ ,  $(x_1, y_1)$ ,  $(x_2, y_2)$  then

$$\begin{aligned} x(\xi, \eta) &= x_0 + (x_1 - x_0)\xi + (x_2 - x_0)\eta, \\ y(\xi, \eta) &= y_0 + (y_1 - y_0)\xi + (y_2 - y_0)\eta \end{aligned} \quad (10.17)$$

is a linear map from  $\Delta_*$  to  $\Delta_k$ . The Jacobian determinant of this map is constant on each element, with magnitude equal to  $2|\Delta_k|$ , the ratio of triangle areas. On  $\Delta_*$  any linear function is

a linear combination of three (nodal) basis functions, namely

$$\chi_0(\xi, \eta) = 1 - \xi - \eta, \quad \chi_1(\xi, \eta) = \xi, \quad \chi_2(\xi, \eta) = \eta. \quad (10.18)$$

If vertex  $\ell$  of  $\Delta_*$  is mapped by (10.17) to node  $\mathbf{x}_i$  then the hat function  $\psi_i$  satisfies

$$\psi_i(x(\xi, \eta), y(\xi, \eta)) = \chi_\ell(\xi, \eta). \quad (10.19)$$

Now, recalling both the change-of-variables formula for integrals and the chain rule, we can write

$$F_i^k(\mathbf{u}) = 2|\Delta_k| \int_{\Delta_*} H_\ell^k(\mathbf{u}, \xi, \eta) d\xi d\eta, \quad (10.20)$$

where from (10.11) the integrand is

$$H_\ell^k(\mathbf{u}, \xi, \eta) = [a(u^h) \nabla u^h \cdot \nabla \psi_i - f(u^h) \psi_i]_{\Delta_*}. \quad (10.21)$$

Here vertex  $\ell$  of  $\Delta_*$  corresponds to node  $\mathbf{x}_i$  and the gradient “ $\nabla$ ” is in variables  $x, y$ . Exercises 10.2 and 10.3 address the remaining details needed to implement (10.21) as a C function.

Integrals (10.20) are now approximated using symmetric quadrature rules; see page 171 in the Interlude chapter. Thus, from quadrature nodes  $(\xi_r, \eta_r) \in \Delta_*$  and weights  $w_r$ , we actually compute element residuals as

$$F_i^k(\mathbf{u}) \approx 2|\Delta_k| \sum_{r=0}^{n_Q-1} w_r H_\ell^k(\mathbf{u}, \xi_r, \eta_r). \quad (10.22)$$

For each Neumann boundary segment we use midpoint rule quadrature; compare Exercise 10.4. If segment  $s_\nu$  is incident to node  $\mathbf{x}_i \in \partial\Omega$  then hat function  $\psi_i$  has value 1/2 at the segment midpoint  $(x_m, y_m)$ , thus

$$\varphi_i^\nu \approx g_N(x_m, y_m) \psi_i(x_m, y_m) |s_\nu| = \frac{1}{2} |s_\nu| g_N(x_m, y_m); \quad (10.23)$$

otherwise, if  $\mathbf{x}_i \notin \overline{s_\nu}$  then  $\varphi_i^\nu = 0$ .

## Meshes from Gmsh

The widely available Gmsh software [59] generates unstructured meshes, including triangulations, of plane regions. While Gmsh has a graphical user interface (GUI) for interactive meshing and visualization, we only need its command-line interface here. See the website [gmsh.info](http://gmsh.info) for documentation.

Gmsh takes as input an ASCII file with .geo extension to describe the polygonal boundary of the domain. For example, trap.geo in Code 10.1 describes the polygon  $\partial\Omega$  and boundary conditions shown in Figure 10.2.

```
// trapezoid domain geometry
// usage: gmsh -2 trap.geo

cl = 1.5; // characteristic length
Point(1) = {2.0,0.0,0,cl};
Point(2) = {1.0,1.0,0,cl};
Point(3) = {-1.0,1.0,0,cl};
Point(4) = {-2.0,0.0,0,cl};
```

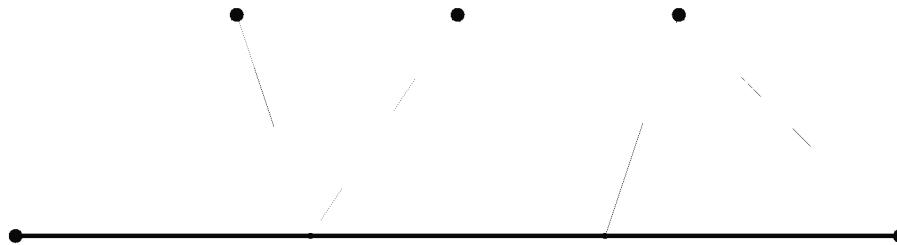
```

Line(5) = {1,2};
Line(6) = {2,3};
Line(7) = {3,4};
Line(8) = {4,1};
Line Loop(9) = {5,6,7,8};
Plane Surface(10) = {9};
Physical Line("dirichlet") = {5,6,7};
Physical Line("neumann") = {8};
Physical Surface("interior") = {10};

```

**Code 10.1.** *c/ch10/meshes/trap.geo*. Definition of the polygon  $\partial\Omega$  and boundary conditions shown in Figure 10.2.

In this file each polygon vertex (Point) and boundary segment (Line) gets a unique integer identifier. A characteristic length, included for each Point, gives the default target value of the typical side length  $h$  of the generated mesh (below). A single Line Loop and Plane Surface [59] define the topology of our simply-connected 2D domain. In order for the generated mesh to contain sufficient information to tell our solver where boundary conditions should be applied, all parts of the boundary must have added “Physical” labels. The parts of  $\partial\Omega$  on which `dirichlet` and `neumann` conditions apply are labeled, as is the `interior`.



**Figure 10.6.** The triangulation in Gmsh file *trap1.msh*.

The coarse triangulation in Figure 10.6 comes from applying Gmsh as follows:

```

$ cd c/ch10/meshes
$ gmsh -2 trap.geo -o trap1.msh

```

This generates `trap1.msh`,<sup>31</sup> shown in Code 10.2, with  $N = 7$  nodes,  $K = 5$  elements,  $P = 3$  Neumann boundary segments, and  $n_D = 5$  nodes on the Dirichlet boundary. This output file defines the labels (`$PhysicalNames`), node locations (`$Nodes`), and element topology (`$Elements`). The nodes are given by quadruples “`n x y 0`” where `n` is the node index, `x,y` are 2D coordinates, and the `z` coordinate is zero.

```

$MeshFormat
2.2 0 8
$EndMeshFormat
$PhysicalNames
3
1 1 "dirichlet"
1 2 "neumann"
2 3 "interior"
$EndPhysicalNames
$Nodes
7

```

<sup>31</sup>If your `.msh` file looks different try `gmsh -format msh22 -2 trap.geo -o trap1.msh` for legacy format.

```

1 2 0 0
2 1 1 0
3 -1 1 0
4 -2 0 0
5 2.755129457909788e-12 1 0
6 -0.6666666666703693 0 0
7 0.6666666666629641 0 0
$EndNodes
$Elements
12
1 1 2 1 5 1 2
2 1 2 1 6 2 5
3 1 2 1 6 5 3
4 1 2 1 7 3 4
5 1 2 2 8 4 6
6 1 2 2 8 6 7
7 1 2 2 8 7 1
8 2 2 3 10 7 5 6
9 2 2 3 10 7 1 2
10 2 2 3 10 4 6 3
11 2 2 3 10 6 5 3
12 2 2 3 10 7 2 5
$EndElements

```

**Code 10.2.** *c/ch10/meshes/trap1.msh*. Defines the mesh in Figure 10.6.

Both boundary segments and triangular elements are listed in the \$Elements section. The former correspond to lists “j 1 2 a b c d”, where j is the segment index and “1 2” says this a one-dimensional segment described by two labels. Then a,b are the labels while c,d are the node indices of the ends of the segment. Triangular elements are given by lines “k 2 2 a b c d e”, where k is the element index, “2 2” indicates a 2D triangle with two labels, a,b are the labels, and c,d,e are the indices of the vertices (nodes). The labels for boundary segments and elements include both the original values using in the .geo file plus new labels corresponding to the \$PhysicalNames.

We will test our FE code on a sequence of refined meshes. There are two approaches to refining supported in Gmsh, *remeshing* and *splitting*. The left mesh in Figure 10.7 comes from remeshing with a smaller characteristic length:

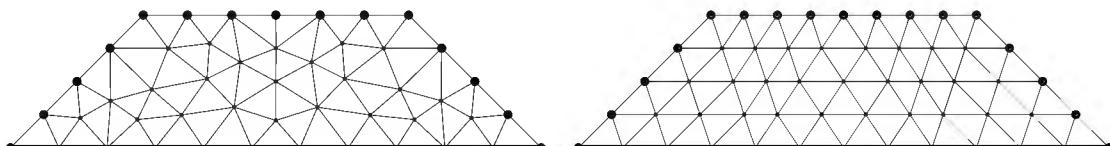
```
| $ gmsh -2 -clmax 0.375 trap.geo -o trapr.msh
```

The right mesh in Figure 10.7 comes from two stages of refinement of trap1.msh by splitting:

```
| $ gmsh -refine trap1.msh -o trap2.msh
| $ gmsh -refine trap2.msh -o traps.msh
```

The resulting meshes are comparable:

- trapr.msh has  $N = 48$  nodes and  $K = 69$  elements, while
- traps.msh has  $N = 55$  nodes and  $K = 80$  elements.



**Figure 10.7.** Finer meshes are generated by remeshing with smaller characteristic length (left; trapr.msh) or by splitting a coarser mesh (right; traps.msh).

However, refinement by splitting maintains the proportions of the elements. Because narrow elements reduce the quality of FE solutions [49], remeshing risks introducing bad elements. Thus, when testing convergence of the FE method later in this chapter, we will use splitting. In Chapter 14 we will again use Gmsh files to input unstructured meshes, but the splitting will be done through the Firedrake [126] interface, which also supports geometric multigrid (GMG) mesh hierarchies and solvers.

## Loading the mesh into PETSc data structures

Our plan is to compute (10.22) and (10.23) by traversing the elements  $\Delta_k$  and Neumann boundary segments  $s_\nu$ , respectively. First, however, we will convert the Gmsh-generated ASCII files into PETSC types and store them in binary format so that the initial stage of the solution method becomes reasonably fast.

As we have already seen there are two kinds of “data” to describe a triangulation, *geometrical* and *topological*. For the Gmsh format the geometry consists of the nodal coordinates, pairs of real numbers for each node (vertex). The topology includes which elements are incident to which nodes; in fact an element is simply a triple of node indices. Similarly, a boundary segment is a pair of indices for the endpoints.

For topology in PETSC we use IS “index set” types,<sup>32</sup> which the reader may regard as integer-valued Vecs. The element IS holds  $3K$  integers while the boundary segment IS holds  $2P$  integers. We use a third IS, of length  $N$ , to store nodal flags, namely a 0 for an interior node, a positive value for any boundary node, and a 2 for each Dirichlet boundary node.

The Python script `msh2petsc.py` (not shown) reads the Gmsh-generated ASCII<sup>33</sup> `.msh` file and then writes two files in PETSC binary format with extensions `.is` and `.vec`. The `.is` file holds the three ISs in a particular order: element triples, then nodal boundary flags, and then boundary segment pairs. (In the purely Dirichlet boundary conditions case where  $P = 0$ , the boundary segment IS has first entry negative.) The `.vec` file is simpler: it contains a single two degrees-of-freedom Vec for the node coordinates. An important detail is that Gmsh uses one-based indexing; the script lowers the node indices by one. (Also, the `.is` and `.vec` extensions here are not standard but they are used by `unfem.c` below.)

The following generates PETSC binary files `meshes/trap1.{is,vec}`:

```
$ cd c/ch10/
$ ln -s $PETSC_DIR/lib/petsc/bin/petsc_conf.py
$ ln -s $PETSC_DIR/lib/petsc/bin/PetscBinaryIO.py
$ ./msh2petsc.py meshes/trap1.msh
```

Note the script uses Python modules from the PETSC source directory, to which we make symbolic links.

Now that the mesh is stored in binary files, we turn to C codes which read the mesh and solve the problem. Mesh input/output functions are separated from the tasks of computing the residual and setting up the PETSC solver. See these source files in `c/ch10/`:

- `cases.h`: Exact solutions and boundary conditions.
- `um.h` and `um.c`: These define a naive unstructured-mesh “object” UM, actually just a C struct, and provide an interface for it. The functions read a mesh from binary files and view it.
- `unfem.c`: The FE method itself is here, as well as a `main()` function. It reads options, calls UM functions to read the mesh, calls functions from `cases.h` to set boundary conditions, provides a residual-evaluation function for call-back, sets up a SNES solver object, runs the solver, and reports the numerical error.

<sup>32</sup>IS types are also used to distribute indexing across processes, but not in this chapter.

<sup>33</sup>Gmsh can generate a binary format `.msh` file but we do not use it here.

Regarding this solver design, we have attempted modularity. However, our representation of an unstructured mesh remains simple, and even deliberately naive.

In the excerpts below the `UM` struct is declared in Code 10.3 and its methods are declared in Code 10.4. Regarding the methods, function `UMReadNodes()` should be called before `UMReadISs()`; these call PETSc functions `VecLoad()` and `ISLoad()`, respectively. The modestly tedious implementations of these functions, in `c/ch10/um.c`, including some input checking steps, are not shown.

```
// location of one node
typedef struct {
    PetscReal x,y;
} Node;

// data type for an Unstructured Mesh
typedef struct {
    PetscInt N,           // number of nodes
              K,           // number of elements
              P;           // number of Neumann boundary segments; may be 0
    Vec     loc;          // nodal locations; length N, dof=2 Vec
    IS      e,            // element triples; length 3K
            //      values e[3*k+0],e[3*k+1],e[3*k+2]
            //      are indices into node-based Vecs
            bf,          // flag for boundary nodes; length N
            //      if bf[i] > 0 then node i is on boundary
            //      if bf[i] == 2 then node i is Dirichlet
            ns;          // Neumann boundary segment pairs; length 2P;
            //      may be a null ptr; values s[2*p+0],s[2*p+1]
            //      are indices into node-based Vecs
} UM;
```

**Code 10.3.** `c/ch10/um.h`, part I. `UM` is an unstructured-mesh data type.

```
PetscErrorCode UMIInitialize(UM *mesh); // call first
PetscErrorCode UMDestroy(UM *mesh); // call last

// create Vec and then read node coordinates from file into it
PetscErrorCode UMReadNodes(UM *mesh, char *filename);

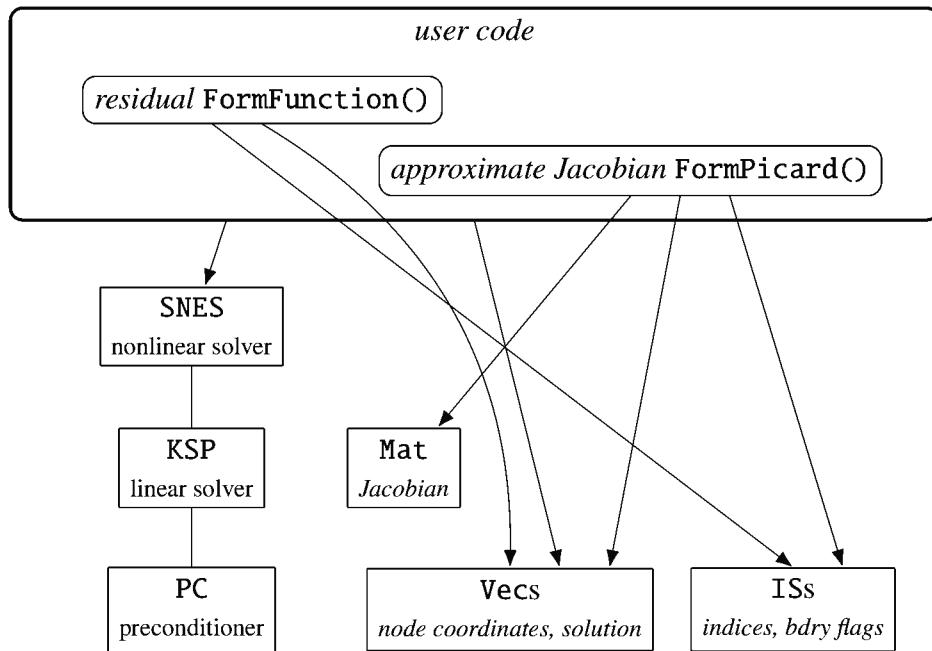
// create ISs and then read element triples, Neumann boundary segments,
// and boundary flags into them; call UMReadNodes() first
PetscErrorCode UMReadISs(UM *mesh, char *filename);

// view all fields in UM to the viewer
PetscErrorCode UMViewASCII(UM *mesh, PetscViewer viewer);
PetscErrorCode UMViewSolutionBinary(UM *mesh, char *filename, Vec u);

// compute statistics for mesh: maxh,meanh are for triangle side
// lengths; maxa,meana are for areas
PetscErrorCode UMStats(UM *mesh, PetscReal *maxh, PetscReal *meanh,
                      PetscReal *maxa, PetscReal *meana);

// access to a length-N array of structs for nodal coordinates
PetscErrorCode UMGetNodeCoordArrayRead(UM *mesh, const Node **xy);
PetscErrorCode UMRestoreNodeCoordArrayRead(UM *mesh, const Node **xy);
```

**Code 10.4.** `c/ch10/um.h`, part II. Methods of the `UM` data type.



**Figure 10.8.** The structure of `unfem.c`.

## Initial implementation and testing

With an unstructured mesh in hand we implement an FE method in the program `c/ch10/unfem.c`. Figure 10.8 shows its structure, and extracts are shown in Codes 10.5–10.8. So that call-backs have access to the functions  $a$ ,  $f$ ,  $g_D$ ,  $g_N$  in FE method (10.9), we first define a solution context `unfemCtx` (Code 10.5). This struct also includes the mesh, namely a pointer to a `UM` instance.

```

typedef struct {
    UM      *mesh;
    PetscInt solncase,
             quaddegree;
    PetscReal (*a_fcn)(PetscReal, PetscReal, PetscReal);
    PetscReal (*f_fcn)(PetscReal, PetscReal, PetscReal);
    PetscReal (*gD_fcn)(PetscReal, PetscReal);
    PetscReal (*gN_fcn)(PetscReal, PetscReal);
    PetscReal (*uexact_fcn)(PetscReal, PetscReal);
} unfemCtx;

```

**Code 10.5.** `c/ch10/unfem.c`, part I. Context for FE method (10.9), (10.10).

The first actions of the `main()` function are to read options, choose the problem case, and read a mesh from files via `UM` methods (not shown). Then, as shown in Code 10.6, it allocates `Vecs` and configures the `SNES` solver. The default `KSP` and `PC` types are reset to conjugate gradient (CG) and incomplete Cholesky (ICC). (See Chapter 3 regarding these choices, which we reconsider below.) Note we access the `KSP` inside the `SNES`, set its type, then access the `PC` inside the `KSP` and set its type as well. Next a `Mat` is allocated and configured, about which much more is said below.

```

// configure Vecs
VecCreate(PETSC_COMM_WORLD,&r);
VecSetSizes(r,PETSC_DECIDE,mesh.N);
VecSetFromOptions(r);
VecDuplicate(r,&u);
VecSet(u,0.0);

// configure SNES: reset default KSP and PC
SNESCreate(PETSC_COMM_WORLD,&snes);
SNESSetFunction(snes,r,FormFunction,&user);
SNESGetKSP(snes,&ksp);
KSPSetType(ksp,KSPCG);
KSPGetPC(ksp,&pc);
PCSetType(pc,PCICC);

// setup matrix for Picard iteration, including preallocation
MatCreate(PETSC_COMM_WORLD,&A);
MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,mesh.N,mesh.N);
MatSetFromOptions(A);
MatSetOption(A,MAT_SYMMETRIC,PETSC_TRUE);
// Preallocation and setting the nonzero (sparsity) pattern is
// recommended; setting the pattern allows finite difference
// approximation of the Jacobian using coloring. Option
// --un_noalloc reveals the poor performance otherwise.
if (noprealloc) {
    MatSetUp(A);
} else {
    PreallocateAndSetNonzeros(A,&user);
}
// The following call-back is ignored under option --snes_fd or
// --snes_fd_color.
SNESSetJacobian(snes,A,A,FormPicard,&user);
SNESSetFromOptions(snes);

// solve
SNESolve(snes,NULL,u);

```

**Code 10.6.** *c/ch10/unfem.c, part II. An extract of main().*

Code 10.7 shows the FE tools which evaluate local basis functions  $\chi_\ell(\eta, \xi)$ , their gradients  $\nabla \chi_\ell$ , and linear combinations  $\sum_\ell v_\ell \chi_\ell$ .

```

PetscReal chi(PetscInt L, PetscReal xi, PetscReal eta) {
    const PetscReal z[3] = {1.0 - xi - eta, xi, eta};
    return z[L];
}

const PetscReal dchi[3][2] = {{-1.0,-1.0},{ 1.0, 0.0},{ 0.0, 1.0}};

// evaluate v(xi,eta) on reference element using local node numbering
PetscReal eval(const PetscReal v[3], PetscReal xi, PetscReal eta) {
    PetscReal sum = 0.0;
    PetscInt L;
    for (L = 0; L < 3; L++)
        sum += v[L] * chi(L,xi,eta);
    return sum;
}

```

**Code 10.7.** *c/ch10/unfem.c, part III. FE tools including local basis functions.*

Then the residual  $\mathbf{F}(\mathbf{u})$  is evaluated by `FormFunction()` in Code 10.8. The input and output Vecs are accessed by `VecGetArrayRead()` and `VecGetArray()`, respectively; note that function `UMGetNodeCoordArrayRead()` (Code 10.4) returns a read-only array pointer of Nodes (Code 10.3). When accessing indices we use `ISGetIndices()`, and each Get function has a matching `Restore`. `FormFunction()` first zeros the residual  $\mathbf{F}$  so that elementwise contributions can be accumulated. It loops through the Neumann boundary segments computing integrals (10.12) by the midpoint rule, adds them to the corresponding entries of  $\mathbf{F}$ , and then loops through the element residuals (10.11) while also identifying Dirichlet boundary nodes and setting (10.15). Sums (10.13) are complete at the end of this loop.

```

PetscErrorCode FormFunction(SNES snes, Vec u, Vec F, void *ctx) {
    unfemCtx          *user = (unfemCtx*) ctx;
    const Quad2DTri q = symmgauss[user->quaddegree-1];
    const PetscInt  *ae, *ans, *abf, *en;
    const Node      *aloc;
    const PetscReal*au;
    PetscInt       p, na, nb, k, l, r;
    PetscReal      *aF, unode[3], gradu[2], gradpsi[3][2], uquad[4],
                      aquad[4], fquad[4], dx, dy, dx1, dx2, dy1, dy2,
                      detJ, ls, xmid, ymid, sint, xx, yy, psi, ip, sum;

    VecSet(F,0.0);
    VecGetArray(F,&aF);
    UMGetNodeCoordArrayRead(user->mesh,&aloc);
    ISGetIndices(user->mesh->bf,&abf);

    // Neumann boundary segment contributions (if any)
    if (user->mesh->P > 0) {
        ISGetIndices(user->mesh->ns,&ans);
        for (p = 0; p < user->mesh->P; p++) {
            na = ans[2*p+0]; nb = ans[2*p+1]; // end nodes of segment
            dx = aloc[na].x-aloc[nb].x; dy = aloc[na].y-aloc[nb].y;
            ls = sqrt(dx * dx + dy * dy); // length of segment
            // midpoint rule; psi_na=psi_nb=0.5 at midpoint of segment
            xmid = 0.5*(aloc[na].x+aloc[nb].x);
            ymid = 0.5*(aloc[na].y+aloc[nb].y);
            sint = 0.5 * ls * user->gN_fcn(xmid,ymid);
            // nodes could be Dirichlet
            if (abf[na] != 2)
                aF[na] -= sint;
            if (abf[nb] != 2)
                aF[nb] -= sint;
        }
        ISRestoreIndices(user->mesh->ns,&ans);
    }

    // element contributions and Dirichlet node residuals
    VecGetArrayRead(u,&au);
    ISGetIndices(user->mesh->e,&ae);
    for (k = 0; k < user->mesh->K; k++) {
        // element geometry and hat function gradients
        en = ae + 3*k; // en[0], en[1], en[2] are nodes of element k
        dx1 = aloc[en[1]].x - aloc[en[0]].x;
        dx2 = aloc[en[2]].x - aloc[en[0]].x;
        dy1 = aloc[en[1]].y - aloc[en[0]].y;
        dy2 = aloc[en[2]].y - aloc[en[0]].y;
        detJ = dx1 * dy2 - dx2 * dy1;
        for (l = 0; l < 3; l++) {
            gradpsi[l][0] = (dy2 * dchi[l][0] - dy1 * dchi[l][1]) / detJ;
            gradpsi[l][1] = (-dx2 * dchi[l][0] + dx1 * dchi[l][1]) / detJ;
        }
    }
}

```

```

    }
    // u and grad u on element
    gradu[0] = 0.0;
    gradu[1] = 0.0;
    for (l = 0; l < 3; l++) {
        if (abf[en[l]] == 2) // enforces symmetry
            unode[l] = user->gD_fcn(aloc[en[l]].x, aloc[en[l]].y);
        else
            unode[l] = au[en[l]];
        gradu[0] += unode[l] * gradpsi[l][0];
        gradu[1] += unode[l] * gradpsi[l][1];
    }
    // function values at quadrature points on element
    for (r = 0; r < q.n; r++) {
        uquad[r] = eval(unode, q.xi[r], q.eta[r]);
        xx = aloc[en[0]].x + dx1 * q.xi[r] + dx2 * q.eta[r];
        yy = aloc[en[0]].y + dy1 * q.xi[r] + dy2 * q.eta[r];
        aquad[r] = user->a_fcn(uquad[r], xx, yy);
        fquad[r] = user->f_fcn(uquad[r], xx, yy);
    }
    // residual contribution for each node of element
    for (l = 0; l < 3; l++) {
        if (abf[en[l]] == 2) { // set Dirichlet residual
            xx = aloc[en[l]].x; yy = aloc[en[l]].y;
            aF[en[l]] = au[en[l]] - user->gD_fcn(xx, yy);
        } else {
            sum = 0.0;
            for (r = 0; r < q.n; r++) {
                psi = chi(l, q.xi[r], q.eta[r]);
                ip = InnerProd(gradu, gradpsi[l]);
                sum += q.w[r] * ( aquad[r] * ip - fquad[r] * psi );
            }
            aF[en[l]] += PetscAbsReal(detJ) * sum;
        }
    }
}

ISRestoreIndices(user->mesh->e,&ae);
VecRestoreArrayRead(u,&au);
ISRestoreIndices(user->mesh->bf,&abf);
UMRestoreNodeCoordArrayRead(user->mesh,&aloc);
VecRestoreArray(F,&aF);
return 0;
}

```

**Code 10.8.** *c/ch10/unfem.c, part IV. FormFunction() traverses the elements and boundary segments to compute residuals (10.13).*

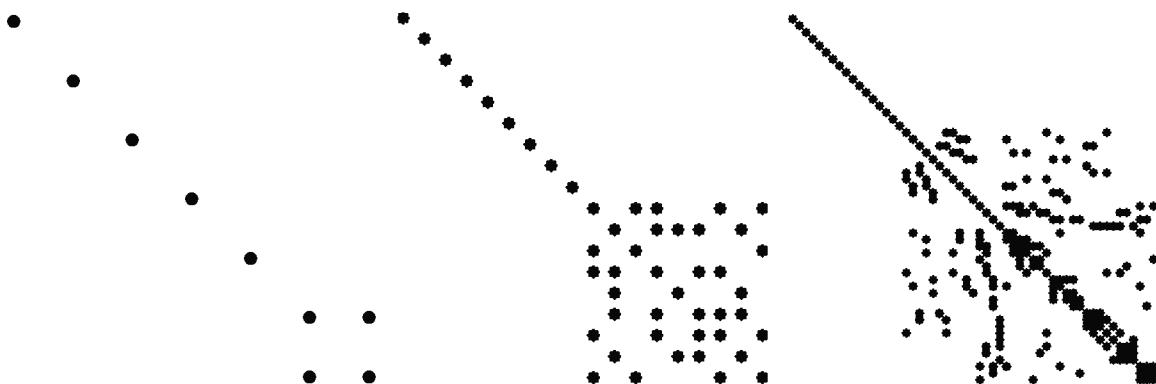
Now, the process of debugging is difficult to show, but of course bugs appeared and were resolved. Here is a first run using the coarse triangulation in Figure 10.6, option `-snes_fd`, the default “case 0” exact solution, and default quadrature degree 1:

```

$ make unfem
$ ./unfem -un_mesh meshes/trap1 -snes_fd
case 0 result for N=7 nodes with h = 1.414e+00: |u-u_ex|_inf = 7.59e-02

```

To determine whether we are in fact solving the problem, we start by refining the mesh and examining the finite differenced Jacobian matrix. The already generated mesh `traps.msh` (Figure 10.7) first needs to be converted to PETSC binary files:



**Figure 10.9.** Matrix sparsity patterns from `unfem -snes_fd` applied to meshes `trap1.msh`, `trap2.msh`, `traps.msh`.

```
$ ./msh2petsc.py meshes/traps.msh
$ ./unfem -un_mesh meshes/traps -snes_fd -mat_view draw -draw_pause 1
case 0 result for N=55 nodes with h = 3.536e-01: |u-u_ex|_inf = 5.52e-03
```

Note the error is an order of magnitude smaller than with mesh `trap1`. Matrix sparsity patterns for our three levels of refinement (Figure 10.9) show symmetry, and shows the diagonal entries associated to Dirichlet boundary, but no other structure is obvious.

If the implementation is correct then the error will decrease at the theoretically expected  $O(h^2)$  rate [49] as the mesh is refined. To test this we run the following Bash loop in directory `meshes/` to generate ten levels of refined (i.e., by splitting) meshes `trap1.msh-trap10.msh`:

```
gmsh -2 trap.geo -o trap1.msh
./msh2petsc.py trap1.msh
for (( Z=1; Z<10; Z++ )); do
    gmsh -refine trap$Z.msh -o trap$((Z+1)).msh
    ./msh2petsc.py trap$((Z+1)).msh
done
```

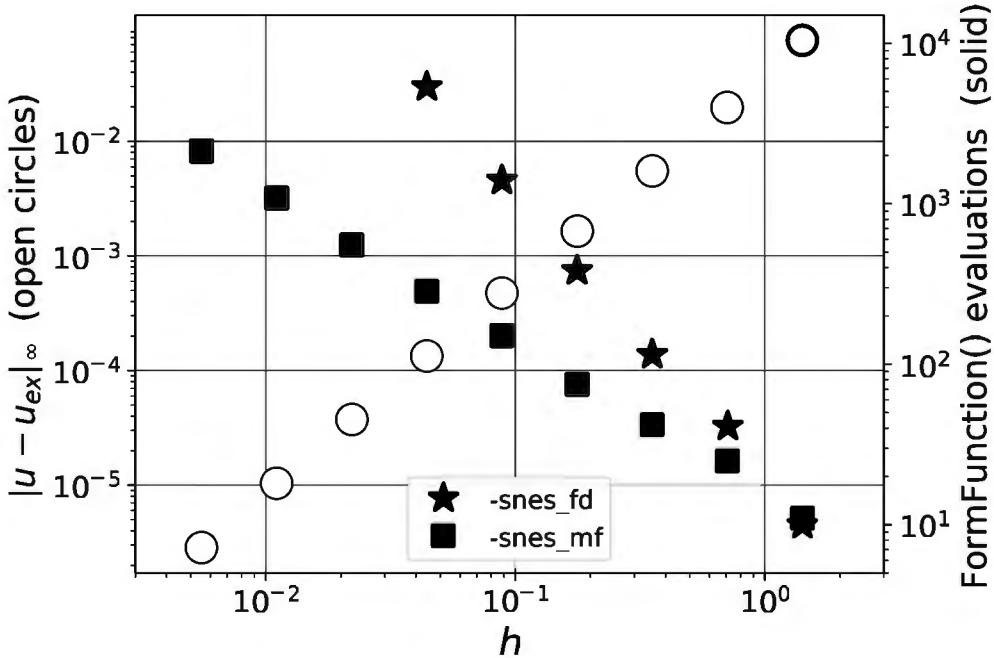
(A helper script is available: `./refinetraps.sh meshes/trap 10`.) Now consider the following runs to solve the linear case 0 problem:

```
| $ ./unfem METHOD -snes_converged_reason -un_mesh meshes/trapX
```

For `METHOD = -snes_fd` the solve succeeds on meshes  $X = 1, \dots, 6$ , but for  $X = 7$  we get a `DIVERGED_FUNCTION_COUNT` error as the evaluations exceeds the default `-snes_max_funcs 10000`. However, Figure 10.10 shows both convergence and, as expected, that the number of function evaluations grows rapidly.<sup>34</sup>

Jacobian-free Newton-Krylov (Chapter 4), i.e., `METHOD = -snes_mf`, is better here; we reach mesh level  $X = 9$  with no error messages and with times less than a minute. Numerical errors from `-snes_fd` and `-snes_mf` runs coincide for all grids where both methods complete. While with `-snes_fd` the number of evaluations of `FormFunction()` is proportional to the number of unknowns  $N$ , for `-snes_mf` it scales with the number of Krylov iterations (Chapter 4). However, the Krylov iterations grow rapidly with refinement. Indeed, without an assembled matrix for preconditioning, we are applying the Krylov method to an unpreconditioned operator which has growing condition number under refinement.

<sup>34</sup>Add `-log_view` and `grep` for `FunctionEval`.



**Figure 10.10.** Numerical error (open circles; left axis) and number of residual evaluations (solid symbols; right axis) using `-snes_fd` and `-snes_mf`.

## Picard iteration as a Newton-like step

To do better we need to assemble a Jacobian and use it for preconditioning. In fact, we will build a matrix which, in nonlinear cases of PDE (10.2), is only an approximate linearization, namely a “Picard” matrix. This common technique, somewhat easier than building the full Jacobian, works reasonably well here.

Because (10.2) is quasilinear [51], the FE discretization (10.9), (10.10) generates an algebraic system of form

$$A(\mathbf{u})\mathbf{u} = \mathbf{b}(\mathbf{u}) \quad (10.24)$$

for  $A(\mathbf{u}) \in \mathbb{R}^{N \times N}$  and  $\mathbf{b}(\mathbf{u}) \in \mathbb{R}^N$ . In *Picard iteration* the matrix  $A(\mathbf{u})$  and right-hand side  $\mathbf{b}(\mathbf{u})$  are “frozen” at the current iterate and a linear system is solved for a new iterate:

$$A(\mathbf{u}^k)\mathbf{u}^{k+1} = \mathbf{b}(\mathbf{u}^k). \quad (10.25)$$

This can also be stated for the continuum problem as a linear PDE:

$$-\nabla \cdot (a(\mathbf{u}^k)\nabla \mathbf{u}^{k+1}) = \mathbf{f}(\mathbf{u}^k). \quad (10.26)$$

Under strong hypotheses on the functions  $A(\mathbf{u})$  and  $\mathbf{b}(\mathbf{u})$  one can prove convergence (Exercise 10.5), but often one has no more *a priori* knowledge about the convergence of (10.25) than for the Newton iteration.

Our approach to (10.2) could have been to write code for the linear case with  $a = a(x, y)$ ,  $f = f(x, y)$  and then to “hard-code” the Picard iteration for the nonlinear cases. However, a more powerful approach is to regard the Picard matrix  $A(\mathbf{u})$  as an approximation to the Jacobian and then use Newton’s method and related tools from SNES. For example, this allows use of a line search to globalize convergence, and the use of  $A(\mathbf{u})$  as a preconditioner for JFNK (Chapter 4).

In fact, we subtract  $A(\mathbf{u}^k)\mathbf{u}^k$  from each side of (10.25) to get

$$A(\mathbf{u}^k)(\mathbf{u}^{k+1} - \mathbf{u}^k) = - (A(\mathbf{u}^k)\mathbf{u}^k - \mathbf{b}(\mathbf{u}^k)),$$

so the step  $\mathbf{s} = \mathbf{u}^{k+1} - \mathbf{u}^k$  and the residual  $\mathbf{F}(\mathbf{u}) = A(\mathbf{u})\mathbf{u} - b(\mathbf{u})$  appear. That is, we have a linear system for the step,

$$A(\mathbf{u}^k)\mathbf{s} = -\mathbf{F}(\mathbf{u}^k), \quad (10.27)$$

which has identical form to the Newton method itself:

$$J(\mathbf{u}^k)\mathbf{s} = -\mathbf{F}(\mathbf{u}^k). \quad (10.28)$$

(See equation (4.4).)

For the linear case where  $\partial a/\partial u = 0$  and  $\partial f/\partial u = 0$  the matrices  $A(\mathbf{u})$  and  $J(\mathbf{u})$  are the same, and neither depends on  $\mathbf{u}$ . On the other hand, for nonlinear cases we expect  $A(\mathbf{u})$  to be a good spectral approximation of  $J(\mathbf{u})$  because (10.26) captures the highest-order derivative terms in (10.2), which control the large eigenvalues and thus the convergence of Krylov methods (Chapter 2).

## Preallocating and assembling the matrix

The function `FormPicard()`, which computes SPD matrix  $A(\mathbf{u})$  from the current estimate  $\mathbf{u}$  of the solution, is shown below in Code 10.9. The entries of  $A$  are sums of element contributions computed as in equations (10.11), (10.13), and (10.15). That is, if

$$A_{ij}^k(\mathbf{u}) = \int_{\Delta_k} a(u^h) \nabla \psi_j \cdot \nabla \psi_i \quad (10.29)$$

then the entries of  $A$  are sums over elements,

$$A_{ij}(\mathbf{u}) = \begin{cases} \delta_{ij}, & i \in \partial_D \Omega \text{ or } j \in \partial_D \Omega, \\ \sum_{k=0}^{K-1} A_{ij}^k(\mathbf{u}) & \text{otherwise.} \end{cases} \quad (10.30)$$

Most values  $A_{ij}^k(\mathbf{u})$  are zero because the support of  $\psi_i$  or  $\psi_j$  often does not overlap  $\Delta_k$ ; the matrix is sparse.

```

PetscErrorCode FormPicard(SNES snes, Vec u, Mat A, Mat P, void *ctx) {
    unfemCtx      *user = (unfemCtx*) ctx;
    const Quad2DTri q = symmgauss[user->quaddegree-1];
    const PetscInt *ae, *abf, *en;
    const Node     *aloc;
    const PetscReal *au;
    PetscReal      unode[3], gradpsi[3][2], uquad[4], aquad[4], v[9],
                    dx1, dx2, dy1, dy2, detJ, xx, yy, sum;
    PetscInt       n, k, l, m, r, cr, cv, row[3];

    MatZeroEntries(P);
    ISGetIndices(user->mesh->bf,&abf);
    for (n = 0; n < user->mesh->N; n++) {
        if (abf[n] == 2) {
            v[0] = 1.0;
            MatSetValues(P,1,&n,1,&n,v,ADD_VALUES);
        }
    }
    ISGetIndices(user->mesh->e,&ae);
    VecGetArrayRead(u,&au);
    UMGetNodeCoordArrayRead(user->mesh,&aloc);
    for (k = 0; k < user->mesh->K; k++) {
        en = ae + 3*k; // en[0], en[1], en[2] are nodes of element k
        MatSetValues(A,1,&n,1,&k,au,ROW_VALUES);
    }
}

```

```

// geometry of element
dx1 = aloc[en[1]].x - aloc[en[0]].x;
dx2 = aloc[en[2]].x - aloc[en[0]].x;
dy1 = aloc[en[1]].y - aloc[en[0]].y;
dy2 = aloc[en[2]].y - aloc[en[0]].y;
detJ = dx1 * dy2 - dx2 * dy1;
// gradients of hat functions and u on element
for (l = 0; l < 3; l++) {
    gradpsi[l][0] = (dy2 * dchi[l][0] - dy1 * dchi[l][1]) / detJ;
    gradpsi[l][1] = (-dx2 * dchi[l][0] + dx1 * dchi[l][1]) / detJ;
    if (abf[en[l]] == 2)
        unode[l] = user->gD_fcn(aloc[en[l]].x, aloc[en[l]].y);
    else
        unode[l] = au[en[l]];
}
// function values at quadrature points on element
for (r = 0; r < q.n; r++) {
    uquad[r] = eval(unode, q.xi[r], q.eta[r]);
    xx = aloc[en[0]].x + dx1 * q.xi[r] + dx2 * q.eta[r];
    yy = aloc[en[0]].y + dy1 * q.xi[r] + dy2 * q.eta[r];
    aquad[r] = user->a_fcn(uquad[r], xx, yy);
}
// generate 3x3 element stiffness matrix (may be smaller)
cr = 0; cv = 0; // cr = count rows; cv = entry counter
for (l = 0; l < 3; l++) {
    if (abf[en[l]] != 2) {
        row[cr++] = en[l];
        for (m = 0; m < 3; m++) {
            if (abf[en[m]] != 2) {
                sum = 0.0;
                for (r = 0; r < q.n; r++)
                    sum += q.w[r] * aquad[r]
                           * InnerProd(gradpsi[l], gradpsi[m]);
            }
            v[cv++] = PetscAbsReal(detJ) * sum;
        }
    }
}
MatSetValues(P, cr, row, cr, row, v, ADD_VALUES);
}
ISRestoreIndices(user->mesh->e, &ae);
ISRestoreIndices(user->mesh->bf, &abf);
VecRestoreArrayRead(u, &au);
UMRestoreNodeCoordArrayRead(user->mesh, &aloc);

MatAssemblyBegin(P, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(P, MAT_FINAL_ASSEMBLY);
if (A != P) {
    MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
}
return 0;
}

```

**Code 10.9.** *c/ch10/unfem.c, part V. Generate Mat A, the Picard matrix.*

Note that in `main()` we created a `Mat` for  $A(\mathbf{u})$ , called `MatSetSizes()` and `MatSetFromOptions()` on it, and set a call-back to `FormPicard()` using `SNESSetJacobian()`. (A method called `SNESSetPicard()` exists in the PETSc API, but it is not recommended. The method used here is Picard iteration treated as a Newton-like step.) Now, inside `FormPicard()`,

`MatSetValues()` is called with `ADD_VALUES` to do sum (10.30). Note that we have already computed integral (10.29) when evaluating the residual, namely in equation (10.20), so `FormFunction()` could have been rewritten to generate  $A(\mathbf{u})$  as a side effect, but we write a separate function so that already-tested `FormFunction()` is not touched.

However, preallocating memory for the `Mat` is critical for performance. Without this step, memory would be allocated incrementally as nonzero entries are generated; the resulting inefficiency is easy to demonstrate (below). In addition to preallocation we provide the precise sparsity pattern of the matrix. Actually, in previous chapters we have benefited, perhaps without being aware of it, from automatic preallocation and sparsity-pattern generation when using a DM`DA`. The following function `PreallocateAndSetNonzeros()`, shown in Code 10.10, accomplishes essentially the same actions as `DMCreateMatrix()`. It first preallocates storage for the sparse matrix by providing a count of nonzero entries in each row. Then it actually sets the sparsity pattern by inserting zeros (ironically) where there will be nonzero entries. Because the nonzero pattern of the `Mat` is now known to the SNES, option `-snes_fd_color` can be used, again with easily demonstrated performance benefit.

```
PetscErrorCode PreallocateAndSetNonzeros(Mat J, unfemCtx *user) {
    const PetscInt *ae, *abf, *en;
    PetscInt       *nnz, n, k, l, cr, row[3];
    PetscReal      zero = 0.0,
                    v[9] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};

    // preallocate: set number of nonzeros per row
    ISGetIndices(user->mesh->bf,&abf);
    ISGetIndices(user->mesh->e,&ae);
    PetscMalloc1(user->mesh->N,&nnz);
    for (n = 0; n < user->mesh->N; n++)
        nnz[n] = (abf[n] == 1) ? 2 : 1;
    for (k = 0; k < user->mesh->K; k++) {
        en = ae + 3*k; // en[0], en[1], en[2] are nodes of element k
        for (l = 0; l < 3; l++)
            if (abf[en[l]] != 2)
                nnz[en[l]] += 1;
    }
    MatSeqAIJSetPreallocation(J,-1,nnz);
    PetscFree(nnz);

    // set nonzeros: put values (=zeros) in allocated locations
    for (n = 0; n < user->mesh->N; n++) {
        if (abf[n] == 2) {
            MatSetValues(J,1,&n,1,&n,&zero,INSERT_VALUES);
        }
    }
    for (k = 0; k < user->mesh->K; k++) {
        en = ae + 3*k; // en[0], en[1], en[2] are nodes of element k
        // a 3x3 element stiffness matrix (at most) for each element
        cr = 0; // cr = count rows
        for (l = 0; l < 3; l++) {
            if (abf[en[l]] != 2) {
                row[cr++] = en[l];
            }
        }
        MatSetValues(J,cr,row,cr,row,v,INSERT_VALUES);
    }
    MatAssemblyBegin(J,MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(J,MAT_FINAL_ASSEMBLY);
    // the assembly routine FormPicard() will generate an error if
    // it tries to put a matrix entry in the wrong place
}
```

```

    MatSetOption(J,MAT_NEW_NONZERO_LOCATION_ERR,PETSC_TRUE);
    ISRestoreIndices(user->mesh->e,&ae);
    ISRestoreIndices(user->mesh->bf,&abf);
    return 0;
}

```

**Code 10.10.** *c/ch10/unfem.c, part VI. Preallocation and nonzero pattern for the Picard Mat A.*

Function `PreallocateAndSetNonzeros()` calls `MatSeqAIJSetPreallocation()` with an integer array `nnz`. To compute `nnz[n]`, the number of nonzeros in row  $n$ , where  $n$  is also the node index, we traverse the elements just as in `FormFunction()`. For Dirichlet boundary nodes `nnz[n]` equals one, but otherwise it is one more than the number of triangles incident to an interior node, or two more than that number for Neumann boundary nodes. (In terms of the vertex degree  $d$ ,  $d$  triangles are incident to each interior node while  $d - 1$  are incident to each boundary node.) This scheme is an overestimate for rows near a Dirichlet node, but otherwise exact.

After preallocating, our function sets nonzero locations by traversing the elements and computing  $3 \times 3$  element stiffness matrices just as in `FormPicard()`. In fact, a reasonable workflow is to write the matrix assembly procedure first and then strip it down to simply indicating nonzeros in a separate procedure (like `PreallocateAndSetNonzeros()` here).

Preallocation makes a huge difference. To demonstrate it, first observe that the default path in `main()` is to call `PreallocateAndSetNonzeros()` and then to set the Jacobian call-back to `FormPicard()`. Using a `-with-debugging=0` PETSc configuration we can show over 100 times faster runs with preallocation, on a medium-resolution mesh:

```

$ time ./unfem -un_mesh meshes/trap8 -un_noprealloc
case 0 result for N=41409 nodes with h = 1.105e-02: |u-u_ex|_inf = 1.04e-05
real 103.04
$ time ./unfem -un_mesh meshes/trap8
case 0 result for N=41409 nodes with h = 1.105e-02: |u-u_ex|_inf = 1.04e-05
real 0.56

```

Adding `-info | grep malloc` to these runs shows that former run issues about  $4 \times 10^4$  mallocs for memory [90] during matrix assembly while the latter issues none.

## Convergence and profiling

Next we measure convergence for the verification problems described at the beginning of the Chapter, by using the already-generated meshes. For the linear case 0 and 2 verification problems, runs with mesh levels  $\text{LEV} = 1, 2, \dots, 10$  generated the results in Figure 10.11:

```

$ ./unfem -un_case $CASE -un_mesh meshes/trap$LEV \
-snes_type ksponly -ksp_rtol 1.0e-10

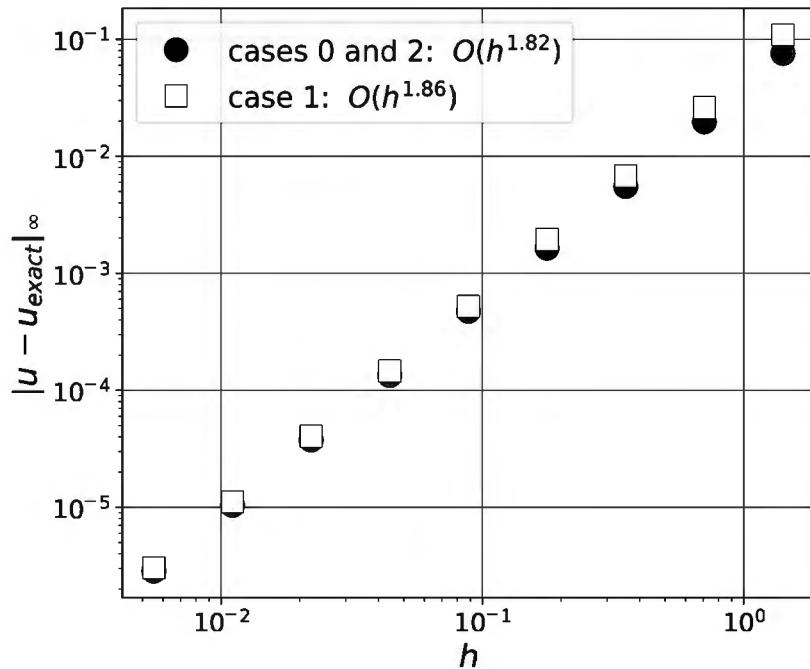
```

The finest mesh has  $N \approx 6.6 \times 10^5$  nodes, with run times less than 60 seconds on the author’s laptop. The solutions to the case 0 and 2 problems are the same—see equation (10.4)—so the numerical errors are the same, thus case 2 serves only as a check on the implementation of Neumann boundary conditions. For the nonlinear case 1 problem, the runs were

```

$ ./unfem -un_case 1 -un_mesh meshes/trap$LEV \
-snes_rtol 1.0e-10 -ksp_rtol 1.0e-10

```



**Figure 10.11.** Numerical errors and convergence rates as a function of maximum mesh edge length  $h$ .

for the same mesh levels, with results also in Figure 10.11. These rates, reasonably close to the theoretical rate  $O(h^2)$  for well-behaved refinements [19], suggest that our numerical method converges.

Now, how does the solver scale with the size of the problem, say the number of mesh nodes  $N$ ? Our goal is optimality (Chapter 7), so we want solver flops and run time to be  $O(N)$  for large  $N$ . A Krylov method can achieve this if we can find a preconditioner such that the (preconditioned) iteration counts are independent of the mesh spacing  $h$ , or equivalently independent of  $N$ .

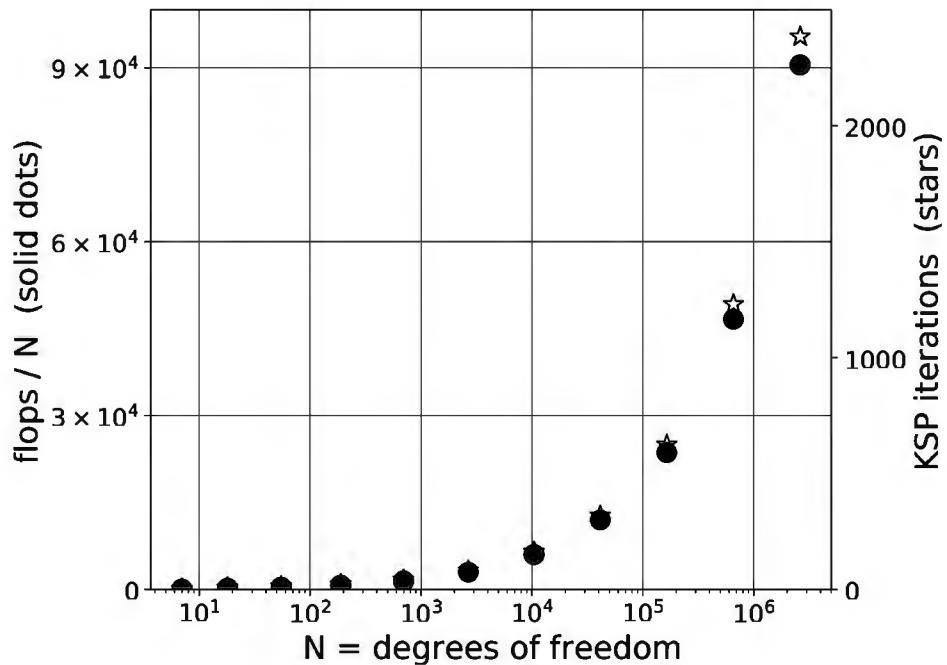
Our solver is constructed from KSP and PC choices made in `main()`—see Code 10.6—namely CG+ICC. For linear case 0, as shown in Figure 10.12, the scaling is poor. The flops grow as  $O(N^{1.58})$  and the run time, for the four finest grids where the timing is not too noisy, at the same rate (not shown). The KSP iterations grow proportionally to  $N$ , roughly doubling each time we reduce the grid spacing by half. (Mesh refinement by splitting gives  $h \rightarrow h/2$  and  $N \rightarrow 4N$ .) Thus the flops per  $N$  are nowhere near constant, and the solver is far from optimal.

For the Poisson equation on a structured grid, and generally in the structured-grid solvers of Chapters 6–9, the CG+GMG combination (geometric multigrid) was optimal because its KSP iterations grew slowly, or not at all, as we decreased  $h$ . However, the run time here includes stages not present in those solvers, specifically our mesh-reading and preallocation schemes.

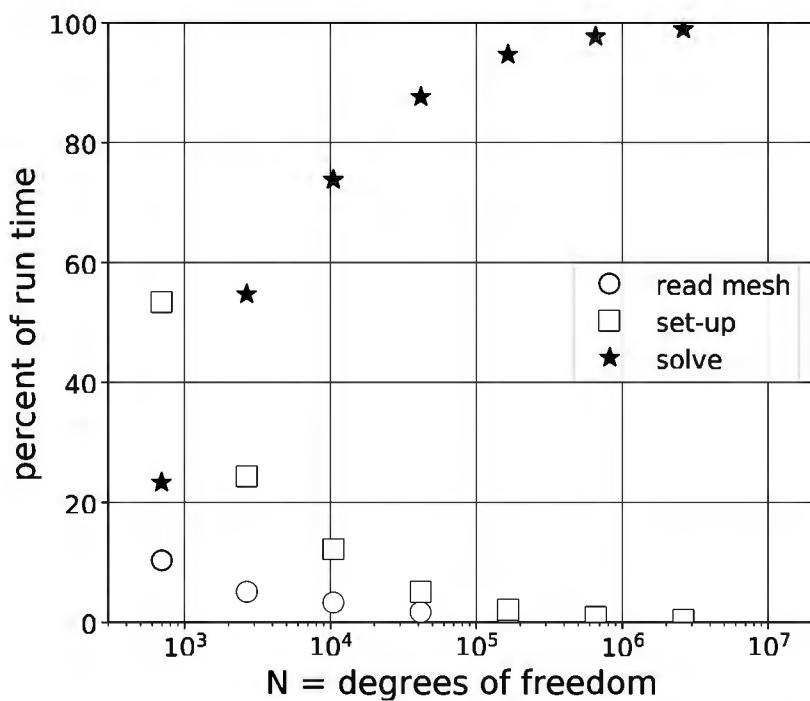
Does profiling support a focus on the preconditioner? Yes, it does. In fact, `unfem.c` logs three computational stages, namely reading the mesh, setup of the solver including preallocation of the linear system, and solving the system. (The third stage is the call to `SNESSolve()`.) That is, the code has lines looking like the following, though they are stripped out of the code extracts above:

```
PetscLogStagePush(stage);
...
code ...
PetscLogStagePop();
```

By adding `-log_view` to the above runs, now using the next-level mesh with  $N = 2.6 \times 10^6$  nodes, we get the timing results in Figure 10.13, showing the percentage of total run time spent



**Figure 10.12.** Our CG+ICC solver is nowhere near optimal. The KSP iterations and flops/ $N$  grow as  $O(N)$ , whereas we want them to be independent of  $N$ .



**Figure 10.13.** Profiling the major code stages of our CG+ICC runs shows that improving the preconditioner is all important. By contrast, our mesh reader, based on PETSC binary files, has tiny run-time fraction even on fine meshes.

in the three stages.<sup>35</sup> For the finer meshes the lesson is very clear: the cost of the solver, ICC-preconditioned CG iteration, dominates over everything else. On the finest two grids the solver takes more than 97% of the time.

## Algebraic multigrid

In recent chapters, geometric multigrid (GMG) on structured grids has been the key to optimal  $O(N)$  solver complexity. The main benefit of GMG preconditioning, namely an  $h$ -independent Krylov iteration count, requires the construction of a hierarchy of grids and associated interpolation/restriction operators. While the DMDA object constructs these internally, `unfem.c` reads only one unstructured mesh from a file. Substantial additional work would be required to construct a mesh hierarchy geometrically. Such a hierarchy will be available using Firedrake—see the example in Chapter 14—but for now we need a different multigrid approach.

*Algebraic multigrid* (AMG) is intended to recover much of the benefit of GMG in situations like this. The idea is that one may act on an assembled matrix  $A$  by recursively identifying and extracting coarse “subgrids,” actually subsets of the variables, to generate interpolation (prolongation) and restriction operations for use in a V-cycle (for example). More or less heuristic rules, based on the expected structure of a Laplacian, or more general elliptic operators, identify these coarse grids. A classical iteration such as SOR or Chebyshev (Chapter 6) is used as a smoother on each subgrid. AMG therefore has a nontrivial setup stage, but at least for linear problems this effort can be reused at (and amortized over) each preconditioner application.

High-level motivation for AMG might start by observing that GMG solutions of elliptic PDEs are “geometric” in two particular senses:

- (i) Interpolation (prolongation)  $P$  and restriction  $R$  operators are defined using the geometry of grids.
- (ii) Coarse-grid operators  $A^C$  created by rediscritization (Chapter 6) use geometric information.

AMG must remove these geometric dependencies. One quickly dispenses with (ii) by using the Galerkin formulation  $A^C = P^\top AP$ , equation (6.21). In fact, `-pc_mg_galerkin` may be regarded as partly-algebraic multigrid. Also, from now on we will assume  $R = P^\top$  so item (i) only concerns prolongation.

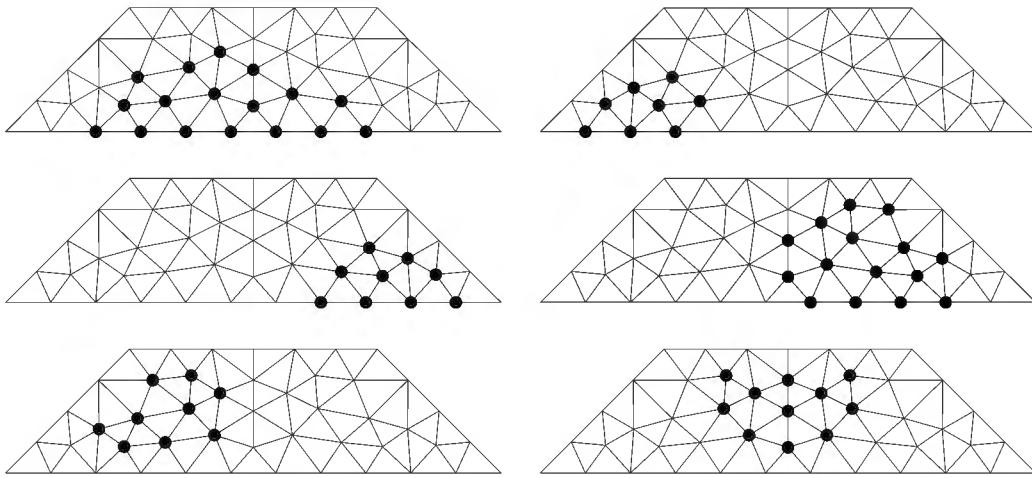
The AMG setup phase must construct a prolongation matrix  $P$  using only the entries of  $A$ . The algorithm which does this will fully determine the “coarse grid” as the range (column space) of the prolongation. To illustrate an AMG-generated coarse grid we use PETSC’s native implementation `-pc_type gamg` [10]. (An alternative is the BoomerAMG preconditioner [77] from the Hypre library, which may be linked to PETSC during configuration; see [10].) Using the  $N = 48$  mesh shown at left in Figure 10.7, the run

```
| $ ./unfem -un_mesh meshes/trapr -pc_type gamg
```

applies AMG-preconditioned CG using only one coarse level—thus a two-level method—with six nodes. The prolongation matrix  $P$  is a full-rank, sparse  $48 \times 6$  matrix whose six columns have nonzero entries as shown by dots in Figure 10.14. The nonzeros in a column of  $P$  identify those fine-grid (original mesh) points which linearly combine to give the interpolant for that column index. (The figure indicates the sparsity of  $P$  but not the magnitude of the entries.) Note that geometric locations for coarse grid points are never defined; the nonzero entries in a column of  $P$  “are” the coarse grid.

---

<sup>35</sup>Time in residual and Jacobian evaluation is also logged (not shown), and included in the solver stage. These evaluations take a small,  $O(N)$  time both for linear and nonlinear cases.



**Figure 10.14.** In AMG, the nonzero entries in columns of  $P$  define overlapping aggregates which are the “nodes” of the “coarse grid.”

However, we will give no details on how AMG generates prolongation matrices, and we proceed to treat AMG as a black-box preconditioner. For more information, introductions [53, 144] are highly recommended.

The default `gamg` preconditioner type is *smoothed aggregation* [144, 150] (`-pc_gamg_type agg`). Exercise 10.8 compares this default type to “classical” AMG (`-pc_gamg_type classical`). At run time the AMG preconditioner can be viewed either by `-ksp_view` or by `-info | grep GAMG` for greater detail. Though tuning of AMG is outside our scope, see the advice in [10].

The untuned, default AMG preconditioner generates a nearly optimal solver for the case 0 linear problem. Consider these runs using trapezoidal meshes at levels  $\text{LEV} = 1, \dots, 12$  (page 258):

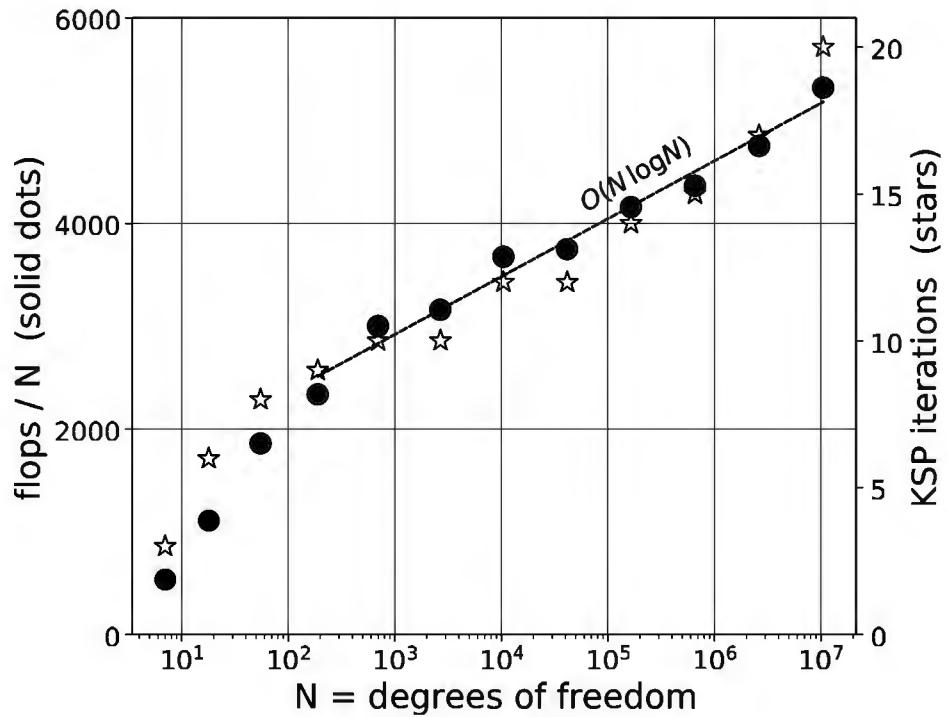
```
$ ./unfem -snes_type ksponly -ksp_rtol 1.0e-10 -pc_type gamg \
-un_mesh meshes/trap$LEV
```

Running on the finest level-12 mesh, with  $N \approx 10^7$ , took about 100 seconds on the author’s workstation.

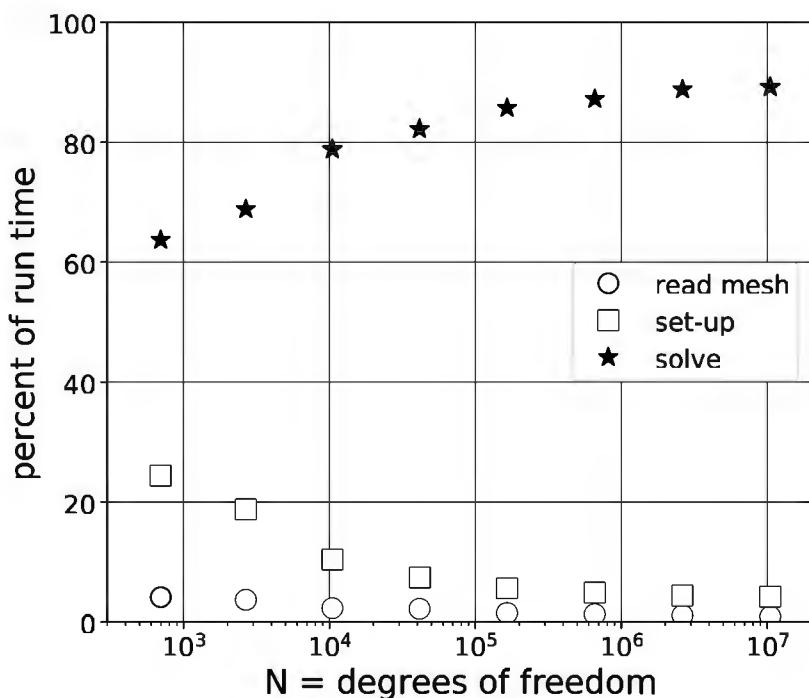
Figure 10.15 shows the result of recording KSP iterations, flops, and time using `-ksp_converged_reason -log_view`. The solver is not quite optimal, and in fact straightforward power regression, over the finest five meshes gives  $\text{flops} = O(N^{1.12})$ , and total run time  $O(N^{1.09})$  on fine meshes. Observe that the flops per degree of freedom roughly doubles as  $N$  increases by four orders of magnitude from  $10^3$  to  $10^7$ . The slow increase in KSP iterations suggests that  $\text{flops} = O(N \log N)$  might be a more appropriate fit, and regression to most of the data (dashed line) gives

$$\frac{\text{flops}}{N} \approx 1232 + 244 \log N.$$

Recall that when we profiled the earlier CG+ICC solver we found that nearly 100% of the run time was taken in the solver, that is, in the `SNESSolve()` event. With AMG the picture is significantly different (Figure 10.16), but the solver still takes a majority of the time. On fine meshes it seems to steadily use about 90% of the time, with AMG PC setup requiring most of the remainder. Again we see that reading the mesh from PETSc binary files takes negligible time. In any case the AMG solver is much faster than the ICC solver. On the level 11 mesh, the finest



**Figure 10.15.** AMG-preconditioned CG iterations solve the Poisson equation, on refinements of a trapezoidal domain, with slowly growing iterations and  $O(N \log N)$  near-optimality of flops.



**Figure 10.16.** Profiling the CG+GAMG solver shows the solver still dominates the cost, with setup costs noticeable and mesh-reader costs very small.

shown in Figure 10.13, the AMG solution takes 25 seconds while ICC takes 250 seconds (on the author’s workstation).

## Nonlinear performance

Recall that the Case 1 problem is a nonlinear porous medium equation:

$$-\nabla \cdot ((1 + u^2) \nabla u) = f(x, y). \quad (10.31)$$

As implemented in `c/ch10/cases.h`, the source  $f$  is computed from the exact solution (10.4) so as to solve (10.31), and the boundary data are the same as in the linear case 0 problem.

We have already implemented and verified a Picard iteration solution to (10.31); see Figure 10.11. While this method is not expected to be quadratically convergent, we may recover that property using either a finite differenced Jacobian or the JFNK method (Chapter 4). In fact, given that AMG gives near-optimal solutions to the linear problems, three sensible strategies are available for (10.31). Namely, consider the run

```
| $ ./unfem -un_mesh meshes/trap$LEV -un_case 1 -snes_rtol 1.0e-10 \
|   -snes_converged_reason -pc_type gamg
```

This is Picard iteration using the analytically-constructed sparse matrix from `FormPicard()`. We may also add either of the following options:

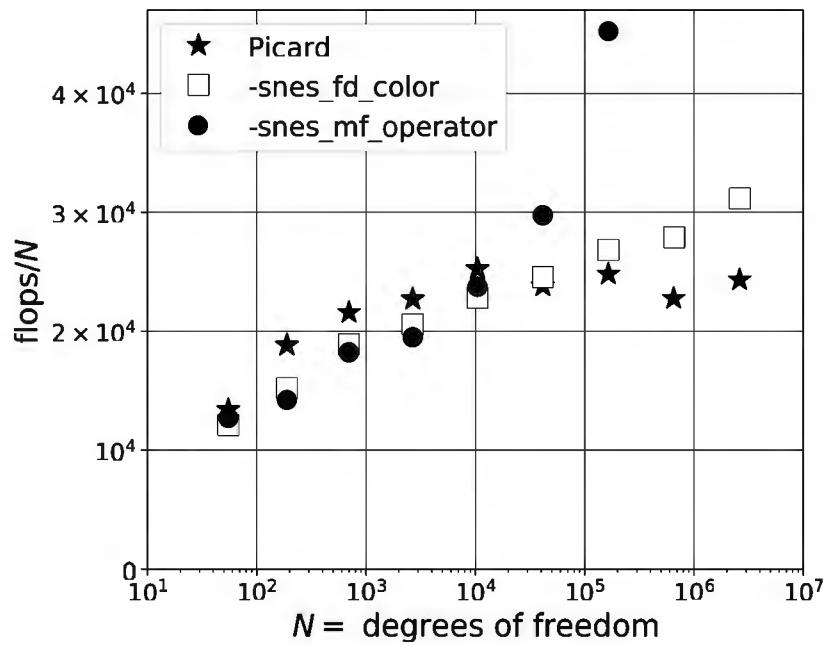
- `-snes_fd_color`: Newton iteration using a finite differenced Jacobian, exploiting the fact that our preallocation function also sets the sparsity of the `Mat`, permitting coloring. `FormPicard()` is not called.
- `-snes_mf_operator`: Newton iteration using a finite differenced Jacobian action—see (4.19) and related discussion—with the `FormPicard()` matrix used only by the AMG preconditioner.

The results, shown in Figure 10.17 for  $\text{LEV} = 3, \dots, 11$  meshes—note  $N = 2.6 \times 10^6$  on the finest mesh—show surprisingly good performance from the plain Picard iteration. It also shows the poor performance of the `-snes_mf_operator` method, which does not converge on the finest two levels. The existence of such examples reflects the details of the nonlinearity in a given PDE, but it surely justifies including Picard iteration in one’s bag of tricks. Both the Picard iteration and `-snes_fd_color` methods show near-optimality, with  $O(N^{1.04})$  and  $O(N^{1.08})$  flops, respectively, from logarithmic regression to all the data.

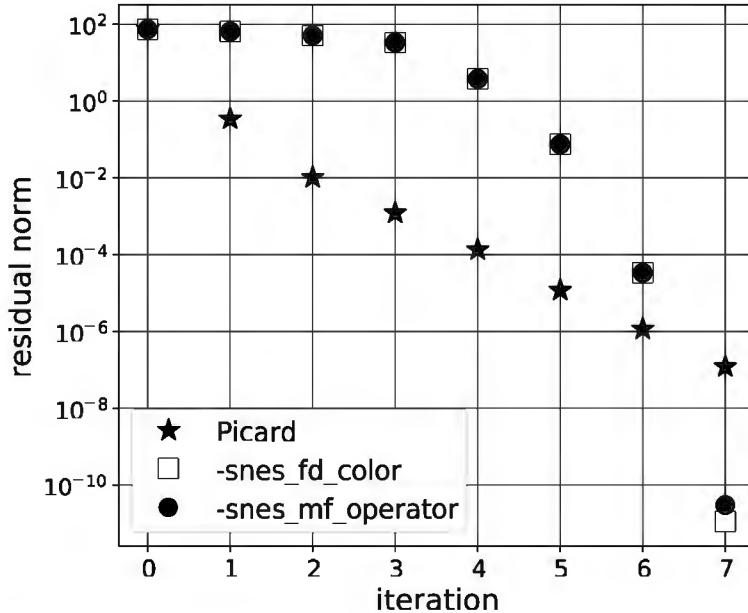
A look at SNES residual norms shows that the Picard iteration, in this case, reduces the residual even when the iterate is far from the solution. Consider the result shown in Figure 10.18 for the  $\text{LEV} = 9$  mesh with  $N = 1.6 \times 10^5$  nodes (on which all methods converge). The Picard iteration steadily decreases the residual, by about an order of magnitude per iteration, while `-snes_fd_color` and `-snes_mf_operator` show classic, delayed quadratic convergence. Which of the Picard and `-snes_fd_color` methods is better on a given grid may therefore depend on the convergence tolerance.

## Performance relative to a DMDA structured grid

One is permitted, of course, to ask whether a given code is “efficient.” Though a good answer is often lacking, for `unfem.c` a reasonable approach is to assume that a DMDA-based structured-grid solution of the Poisson problem, e.g., `fish.c` from Chapter 6, is efficient. We can compare the



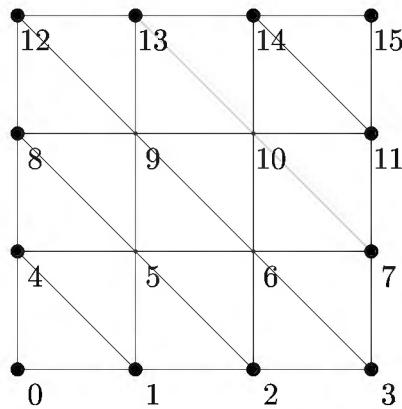
**Figure 10.17.** For the nonlinear porous medium equation (10.31) the Picard and `-snes_fd_color` methods using AMG preconditioning show near-optimality, but `-snes_mf_operator` has convergence difficulties on fine grids.



**Figure 10.18.** For the nonlinear Case 1 problem the the Picard iteration generates immediate and steady decreases in residual norm. Newton methods show classic quadratic convergence.

two codes on the same structured grid, but with `unfem.c` managing it as an unstructured triangulation. Our unstructured solution is efficient if it can, despite the unstructured-mesh overhead, solve the problem nearly as fast as structured FD.

To do this we first write a Python script `c/ch10/genstructured.py` (not shown) which stores a triangulation of the unit square domain  $\mathcal{S} = (0, 1)^2$  directly into PETSc binary format.



For example,

```
| $ ./genstructured.py small 4
```

generates, and stores in `small.vec`, the small grid shown in Figure 10.19.

Now we generate a  $N = 1025^2 \approx 10^6$  grid by

```
| $ ./genstructured.py square 1025
```

Table 10.1 compares timing results on the author’s workstation from `unfem.c`, on this grid, to `fish.c` runs at the same resolution. Note we have added a “Case 3” in `cases.h`, specifically for square domains, so as to solve the same problem as `fish.c`. The runs in the table all have the options

```
-snes_type ksponly -ksp_type cg -ksp_rtol 1.0e-11 -pc_type X
```

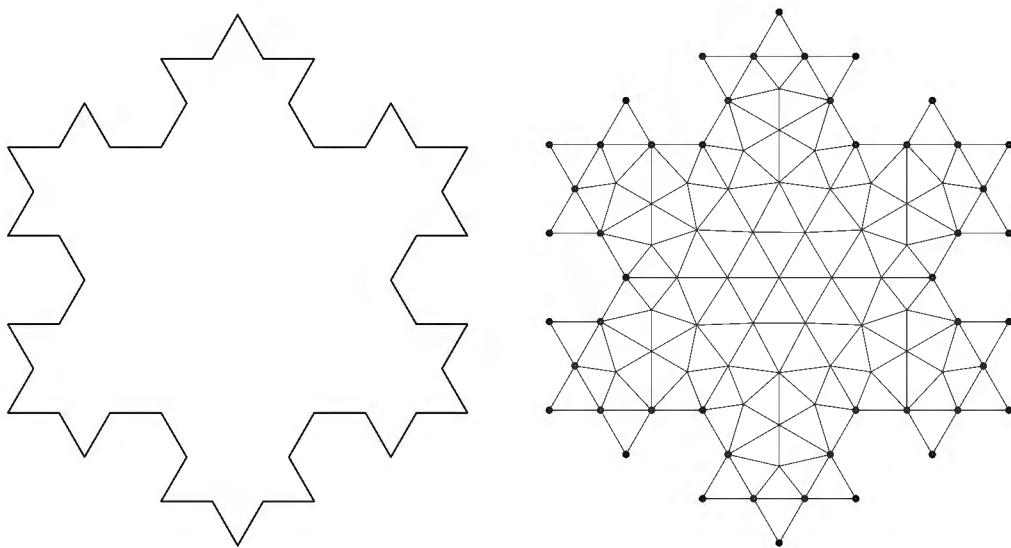
with preconditioning either by V-cycles of `X=gamg` (untuned AMG) or `X=mg` (GMG). For the latter we also compare Galerkin or rediscretized coarse-grid matrices (Chapter 6). All of these runs produced a numerical error norm of less than  $10^{-7}$ .

**Table 10.1.** Measured performance of multigrid solutions of the Poisson equation on a grid of  $N \approx 10^6$  nodes. The `c/ch6/fish.c` code uses DMDA.

Code	Preconditioner	KSP iterations	Time (s)
<code>unfem -un_mesh square -un_case 3</code>	<code>gamg</code>	16	8.8
<code>fish -da_refine 9 (same) (same)</code>	<code>gamg</code> <code>mg -pc_mg_galerkin</code> <code>mg</code>	16 8 8	7.3 3.7 3.3

This comparison shows `unfem.c` is quite efficient. The cost of unstructured indexing is reflected by the difference in run times when using the same preconditioner (i.e., `gamg`). By that measure, `unfem.c` is only 20% slower than the DMDA-based solution. Note that the `unfem.c` time includes reading the mesh from a binary file, something which `fish.c` is not doing at all. The same performance ratios remain at the next-coarser and next-finer levels of refinement as well, namely  $513^2$  and  $2049^2$  grids (not shown).

The additional performance improvement from using GMG, roughly a factor of two faster, is not available to our naive unstructured solver. However, see Chapter 14 for Firedrake/DMplex solutions using GMG on unstructured meshes.



**Figure 10.20.** A Koch snowflake polygon (left) and triangulation thereof (right).

In summary, the following approach solves the Poisson equation efficiently in serial on unstructured meshes:

- (i) reading an mesh from PETSC binary files,
- (ii) evaluating the residual and the (Picard) matrix by traversing the elements using unstructured indexing,
- (iii) preallocating the matrix by traversal of the elements, and
- (iv) AMG solution of the discrete equations.

The performance is, remarkably, comparable to using an FD method on a DMDA structured grid. Items (i) and (iii) are particularly important to efficiency on unstructured meshes. For nonlinear problems, an additional step of setting the sparsity pattern during preallocation, which permits use of `-snes_fd_color`, is also worthwhile.

## Poisson equation on the Koch snowflake

Just for fun we now solve a Poisson problem on polygonal domains like that shown in Figure 10.20. This domain is a stage in the construction of the *Koch snowflake*, one of the earliest fractals [151]. The polygon was generated by a Python script `domain.py` in `c/ch10/koch/` (not shown), and then the mesh was generated by Gmsh:

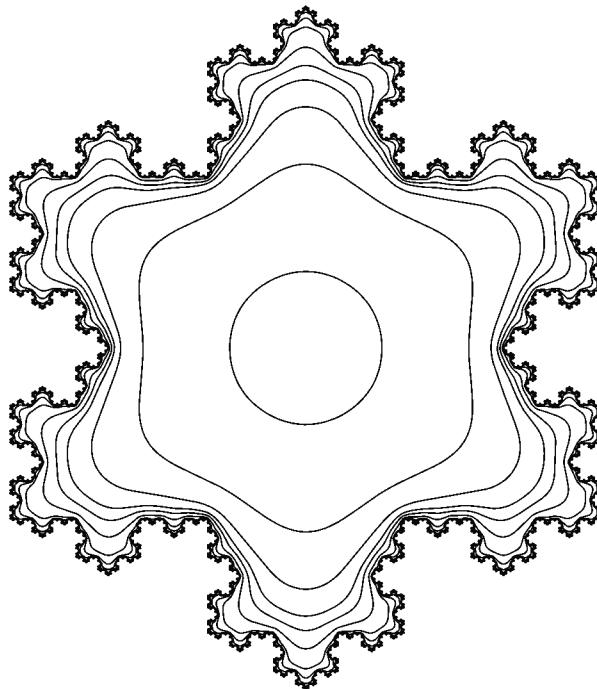
```
$ cd koch/
$ ./domain.py -l 2 -o koch2.geo
$ gmsh -2 koch2.geo
```

As a test problem on these Koch domains, a new case 4 in `cases.h` sets  $a(u) = 1$ ,  $f(u) = 2$ , and  $g_D = 0$  in (10.2), giving

$$-\nabla^2 u = 2, \quad (10.32)$$

subject to homogeneous Dirichlet boundary conditions on all of  $\partial\Omega$ .

The solution  $u(x, y)$  to (10.32) is the expected time at which Brownian motion started at  $(x, y) \in \Omega$  first hits the boundary  $\partial\Omega$ . (See [52, section 6.2] or [120].) In particular, suppose



**Figure 10.21.** A Poisson solution on a Koch snowflake.

$B_t$  is a (random) Brownian path started at the center  $(0, 0)$  of a Koch domain  $\Omega$ . Note that  $B_t$  will, with probability one, cross  $\partial\Omega$ . If  $\tau$  is the (random) first time at which  $B_t$  exits  $\Omega$  then by symmetry,

$$\mathbb{E}(\tau) = u(0, 0) = \max_{(x, y) \in \Omega} u(x, y).$$

We will compute this value using highly refined Koch polygonal domains, to estimate its value for the limiting fractal.

The following commands construct a refined Koch polygon with  $P \approx 1.2 \times 10^4$  boundary segments and a mesh with  $N \approx 4.8 \times 10^5$  nodes (not shown):

```
$ ./domain.py -l 6 -o koch6.geo
$ gmsh -2 koch6.geo
$ ./msh2petsc.py koch6.msh
```

Then we solve the Poisson equation in a few seconds:

```
$ ./unfem -un_mesh koch6 -un_case 4 -un_view_solution \
-snes_type ksponly -ksp_converged_reason -pc_type gamg
```

Option `-un_view_solution` saves a PETSc binary file `koch6.soln` containing the solution  $u$ . A script `petsc2contour.py` (not shown), which can be provided with specific contours to show details near the boundary, then generates Figure 10.21:

```
$ export CT="1e-6 1e-5 1e-4 3e-4 1e-3 3e-3 0.01 0.02 0.03 0.05 0.1 0.2"
$ ./vis/petsc2contour.py -i koch6 --contours $CT
```

The contouring script also reports the solution maximum, from which we estimate  $u(0, 0) \approx 0.22484$ . On a finer polygon from `./domain.py -l 7` we find  $u(0, 0) \approx 0.22499$ ; not shown.

A simple timing study with polygon levels 5, 6, 7 gives  $O(N^{1.29})$  run time, evidence of good scaling of the AMG-preconditioned CG solver. Note we do *not* expect  $O(N^1)$  scaling even from a perfect solver because the problem becomes harder as the boundary becomes more detailed and

longer. (Indeed, the boundary length grows exponentially in the Koch-fractal construction.) One could, however, use the same tools to create a faster solver either by exploiting symmetry, or by grading the mesh to be more refined near the boundary, or both; see Exercise 10.12. Nonetheless it is fair to say that we have mastered the Poisson equation on arbitrary 2D domains.

## Toward mature unstructured-mesh FE method capabilities

This chapter's naive approach to the unstructured-mesh FE method is useful as an introduction, and it might even be adequate for modest-scale problems. However, significant capabilities are missing, including, among others,

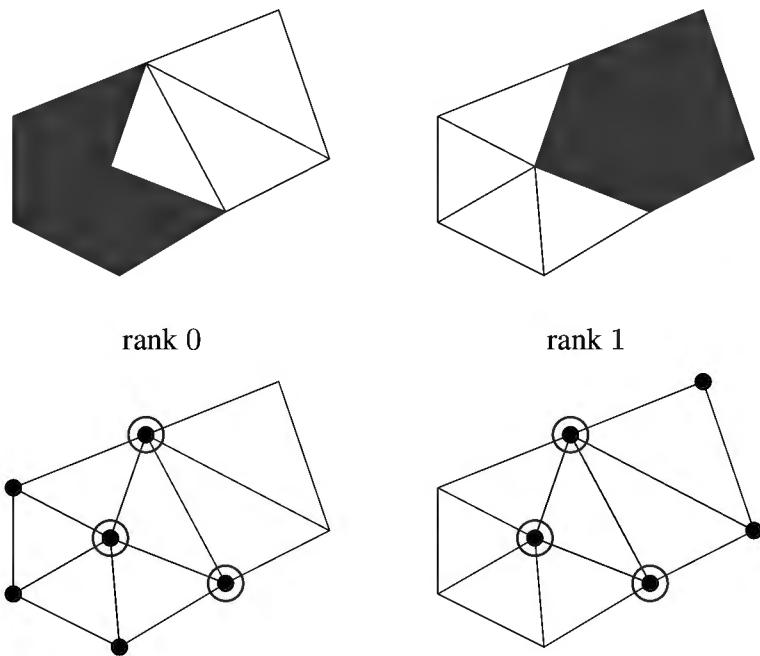
- (i) 3D domains,
- (ii) more element types (e.g., quadrilaterals in 2D, various 3D elements),
- (iii) higher-degree polynomial FE spaces (e.g.,  $P_k, Q_k$  spaces),
- (iv) geometric multigrid (GMG),
- (v) parallel distribution of meshes, and
- (vi) parallel solvers.

A common way to add (i)–(iii) is to use a FE discretization library such as deal.II [12], libMesh [93], FEniCS [107], or Firedrake [126]. These libraries make application of the FE method much easier by hiding the details of the discretization stage. These libraries include domain-specific languages, like the Unified Form Language [5] in FEniCS and Firedrake, for specifying PDE weak forms.

Regarding item (iv), though AMG preconditioning applies without awareness of the topology and geometry of the FE mesh, a desire to use GMG is motivated by our performance measurements on DMDA structured grids. Furthermore, for problems other than the basic Poisson equation the heuristics used in AMG sometimes yield performance significantly worse than GMG. However, adding the ability to use GMG solvers to our code `unfem.c` would require implementations of interpolation and restriction operators, which we have not yet done. (However, this is a capability of the DM`Plex` type.)

A strategy for using unstructured FE in parallel, items (v) and (vi), can be sketched as follows. (Observe that these two goals are not separate; efficient parallel solvers always require distributed data structures.) To integrate weak form (10.3), which is summed element by element, each element should be owned by exactly one process. Thus the elements should be partitioned across the processors (Figure 10.22); this can be done in parallel by ParMETIS [88]. The goal of such a partitioner is to reduce interprocess communication in the solver by minimizing the number of vertices, edges, or faces which are incident to elements on different processors, while also balancing the number of elements. In fact the element partitioning induces a list of vertices, edges, or faces which must be accessible on each process. As each process loops over its owned elements, computing the integral contribution to the residual requires the topological closure of the element, including those lower-dimensional objects which are incident to the element. For example, to use quadrature one must interpolate to the quadrature points from function values at the nodes. These shared nodes must therefore be “ghosted” onto neighboring processors; compare Figures 10.22 and 3.4. Note that in PETSC the management of such indexing is by a `VecScatter` object, and one uses both a “global” indexing for all nodes and a “local” indexing for the process-owned nodes.

The PETSC `DMPlex` type abstracts and implements all of the topological and geometrical information needed for parallel mesh management, as described above. As illustrated in



**Figure 10.22.** In a parallel partition of an unstructured mesh, elements (top) are uniquely owned by processes, but certain lower-dimensional entities (e.g., nodes; bottom) are shared, i.e., ghosted (circled).

Chapter 13, the Firedrake Python library uses DMPlex under the hood. In Chapters 13 and 14 we will achieve all six objectives (i)–(vi), while avoiding tedious user involvement in the discretization and mesh-management aspects of the FE method.

## Exercises

- 10.1. A PDE like (10.2) does not generally arise from a minimization principle. To show this, suppose for simplicity that  $f$  is independent of  $u$  and set homogeneous Dirichlet boundary conditions on  $\partial\Omega$ . Show that if  $\partial a/\partial u \neq 0$  then the Euler-Lagrange equation of

$$\hat{I}[u] = \int_{\Omega} \frac{1}{2} a(u) |\nabla u|^2 - fu \quad (10.33)$$

is *not* the weak form (10.3). (Compare with the derivation of (9.7).) However, knowing that reasonable guess (10.33) is wrong does not exclude that another minimization problem leads to (10.3). Show next that if  $\partial a/\partial u \neq 0$  then for some value of  $u$  the residuals (10.13) satisfy

$$\frac{\partial F_i(u)}{\partial u_j} \neq \frac{\partial F_j(u)}{\partial u_i}. \quad (10.34)$$

This fact *does* exclude a minimization formation when the coefficient  $a(u)$  depends on  $u$ ; explain why.

- 10.2. For the map (10.17) from  $\Delta_*$  to  $\Delta_k$ , the Jacobian is

$$J_k = \frac{\partial(x, y)}{\partial(\xi, \eta)} = \begin{pmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{pmatrix} = \begin{pmatrix} \Delta x_1 & \Delta x_2 \\ \Delta y_1 & \Delta y_2 \end{pmatrix}.$$

Use the chain rule and (10.19) to show that

$$\nabla_{x,y} \psi_i = \frac{1}{\det J_k} \left\langle \Delta y_2 \frac{\partial \chi_\ell}{\partial \xi} - \Delta y_1 \frac{\partial \chi_\ell}{\partial \eta}, -\Delta x_2 \frac{\partial \chi_\ell}{\partial \xi} + \Delta x_1 \frac{\partial \chi_\ell}{\partial \eta} \right\rangle, \quad (10.35)$$

where indices  $i$  and  $\ell$  have the same relationship as in (10.19). Compare formula (9.21) for the structured case.

- 10.3. Formula (10.21) requires some interpretation before the implementation becomes clear. Confirm that formulas (10.17) and (10.19) lead to the following expressions:

$$u^h = \sum_{j=0}^{N-1} \left\{ \begin{array}{c} g_D(\mathbf{x}_j) \\ u_j \end{array} \right\} \chi_{\ell'}(\xi, \eta), \quad (10.36)$$

$$\nabla u^h = \nabla_{x,y} u^h = \sum_{j=0}^{N-1} \left\{ \begin{array}{c} g_D(\mathbf{x}_j) \\ u_j \end{array} \right\} \nabla_{x,y} \psi_j. \quad (10.37)$$

In (10.36) and (10.37), the two cases for computing the coefficient are when  $\mathbf{x}_j \in \partial_D \Omega$  and  $\mathbf{x}_j \notin \partial_D \Omega$ , respectively, and node  $\mathbf{x}_j$  corresponds to vertex  $\ell'$  on  $\triangle_*$ . Note that (10.35) expands  $\nabla_{x,y} \psi_j$  in (10.37).

- 10.4. We use the midpoint rule (10.23) when computing the Neumann boundary segment contributions  $\varphi_\nu^i$  in (10.12), but this is not the only choice. Modify `unfem.c` to optionally use two-point Gaussian quadrature instead. Confirm that accuracy improves in some cases, for coarse grids, but this effect disappears under grid refinement.
- 10.5. Let  $\|\cdot\|$  denote the 2-norm:  $\|w\| = \sqrt{\mathbf{w}^\top \mathbf{w}}$  for  $\mathbf{w} \in \mathbb{R}^N$ . Assume that the right-hand side of Picard iteration (10.25) is bounded,  $\|\mathbf{b}(\mathbf{u})\| \leq B$ , for all  $\mathbf{u}$ , and that the functions are Lipschitz so that  $\|A(\mathbf{u}) - A(\mathbf{v})\| \leq L_A \|\mathbf{u} - \mathbf{v}\|$  and  $\|\mathbf{b}(\mathbf{u}) - \mathbf{b}(\mathbf{v})\| \leq L_b \|\mathbf{u} - \mathbf{v}\|$ . Assume also that  $A(\mathbf{u})$  is uniformly positive-definite, so there is  $\delta > 0$  for which  $\mathbf{w}^\top A(\mathbf{u}) \mathbf{w} \geq \|\mathbf{w}\|^2 / \delta$  for all  $\mathbf{u}, \mathbf{w} \in \mathbb{R}^N$ ; equivalently this says  $\|A^{-1}(\mathbf{u})\| \leq \delta$ .
- (a) Show that the sequence generated by (10.25) is then well defined and bounded,  $\|\mathbf{u}_k\| \leq \delta B$ .
  - (b) Now let  $\alpha = \delta(L_A \delta B + L_b)$ . Show that (10.25) satisfies  $\|\mathbf{u}^{k+1} - \mathbf{u}^k\| \leq \alpha \|\mathbf{u}^k - \mathbf{u}^{k-1}\|$ .
  - (c) Under the (strong) assumption that  $\alpha < 1$ , it follows from a geometric series argument that the Picard iteration converges; show this.

*For the FE matrix in this chapter, the existence of  $\delta$  follows from uniform-ellipticity assumption (10.1). Note that these norm techniques are often applied to ODE IVPs [82].*

- 10.6. Explain why the first of the following two runs, which solve the linear “Case 0” problem, requires fewer SNES iterations:

```
| $ ./unfem -un_mesh meshes/trap3 -snes_monitor -ksp_rtol 1.0e-14
| $ ./unfem -un_mesh meshes/trap3 -snes_monitor -ksp_rtol 1.0e-14 -snes_fd
```

Now explain why the *second* of the following two runs, of a nonlinear problem, requires fewer SNES iterations:

```
| $ ./unfem -un_case 1 -un_mesh meshes/trap3 -snes_monitor
| $ ./unfem -un_case 1 -un_mesh meshes/trap3 -snes_monitor -snes_fd
```

- 10.7. We have assumed that the Dirichlet boundary  $\partial_D\Omega$  contains at least one node so that the linear problem has a unique solution, but one can also modify the solver to handle  $\partial_D\Omega = \emptyset$  cases.
- By modifying the Case 2 exact solution to use only Neumann boundary conditions, confirm experimentally that if  $\partial_D\Omega = \emptyset$  then the equations no longer have a unique solution. Even a direct solve `-ksp_type preonly -pc_type lu` will fail; explain the error message.
  - Now use `SNESGetKSP()` and `KSPSetNullSpace()` to tell the linear solver that the linearization of the Neumann-only problem has constant functions as its null space. Confirm that direct and iterative solvers now succeed.
- 10.8. Option `-un_gamg_save_pint_matlab` saves the top-level GAMG prolongation operator in MATLAB format if the PC is of type `gamg`. Generate `meshes/trapr.vec|is` as shown in the text, and compare the interpolation operators  $P$  from
- ```
$ ./unfem -snes_type ksponly -pc_type gamg -pc_gamg_type X \
    -un_mesh meshes/trapr -un_gamg_save_pint_matlab PX.m
```
- for  $X = \text{agg}, \text{classical}$ . The entries of  $P$  from classical AMG are all in  $[0, 1]$ , and there are entries with exact value one. These can be regarded as an actual coarse subgrid; visualize it and compare Figure 10.14. Now redo the scaling study which generated Figure 10.15 using type `classical`.
- 10.9. Implement the true Jacobian in the  $a = a(x, y)$  case, that is, with the correct linearization when  $\partial f / \partial u \neq 0$ . This will add a diagonal term to the matrix. A test case is appropriate, and it might be built by manufacturing a solution to the equation  $-\nabla^2 u = f(u, x, y)$  where  $f(u, x, y) = e^u + F(x, y)$  and where  $F(x, y)$  is generated by differentiation from a chosen exact solution. Compare results from Picard and Newton iterations.
- 10.10. Use the code from the previous exercise to solve the Liouville-Bratu equation  $-\nabla^2 u = \lambda e^u$  with homogeneous Dirichlet boundary conditions on the unit square  $\mathcal{S} = (0, 1)^2$ . Compare results from Picard and Newton iterations. Compare the critical  $\lambda$  value from Exercise 7.12.
- 10.11. Implement and test a version of `unfem.c` which can handle Robin boundary conditions.
- 10.12. The solution to the Poisson problem on the Koch fractal shown in Figure 10.21 has sixfold symmetry, and it is not difficult to exploit this symmetry to reduce the work. Furthermore one may grade the mesh so it is coarse in the interior and fine near the boundary. Rewrite `c/ch9/koch/domain.py` so that it only generates one-sixth of the boundary, adds rays to the origin to make a closed polygon, and grade the mesh by setting the characteristic length at the origin to a larger value. Apply homogeneous Neumann boundary conditions along the rays. Demonstrate the resulting efficiencies by measuring run-time performance.

## Chapter 11

# Advection without, and then with, diffusion

Previous chapters have solved nonlinear elliptic PDEs by finite difference (FD) and finite element (FE) methods. Now we switch to advection equations, both time dependent and steady state, solving them by freely mixing finite volume (FV) and FD concepts. This chapter serves as a very minimal introduction to FV methods.

The first code solves a linear advection equation in a time-dependent, two-spatial dimensions case. Perhaps surprisingly, the highest-quality results use a *nonlinear* discretization. (As we will see, getting good-looking results from advection equations is harder than one expects.) Then we consider a steady-state equation which combines advection and diffusion, first in 1D and then 2D. The importance of using a high-quality advection discretization depends on the relative amounts of advection and diffusion. Advection-dominated cases require reconsideration of preconditioning. Optimal solvers are achieved by careful use of multigrid preconditioners and discretizations which respect the advection.

## Flux conservation and finite volumes

*Advection* simply refers to the motion of a substance carried along by a velocity field. Closely related terms include *transport* and *convection* [114, 119]; the former we treat as a synonym but the latter suggests a coupled dynamical model for the velocity, a notion which we do not pursue. Advection is also an aspect of the equations for electromagnetic fields, with speed constant, but without an underlying fluid; there is no ether. In any case, for simplicity and concreteness we picture advection equation solutions as a moving fluid carrying the substance passively along, sometimes called a *tracer*. We call the velocity the *wind* and the solution the *concentration*.

An equation for advection may *conserve a flux*, for instance, when the flux  $\phi = \mathbf{a}u$  is the product of wind and concentration:

$$u_t + \nabla \cdot (\mathbf{a}u) = g(u). \quad (11.1)$$

(The sense in which  $u$  is “conserved” will be addressed momentarily.) The solution  $u(t, \mathbf{x})$  to (11.1) is sought for  $t \geq 0$  and  $\mathbf{x}$  in some domain  $\Omega \subset \mathbb{R}^d$ . For simplicity we assume the wind  $\mathbf{a}(\mathbf{x})$  and source  $g(\mathbf{x}, u)$  are functions which are both  $t$  independent and continuous in  $\mathbf{x}$ , though our notation will often suppress dependence on  $\mathbf{x}$ . Furthermore we assume  $\mathbf{a}(\mathbf{x})$  has continuous derivatives and that  $g$  is Lipschitz in  $u$ . Thus the decoupled system of ODEs arising from (11.1) when  $\mathbf{a} = 0$ , namely  $u_t = g(\mathbf{x}, u)$ , is at least well posed for short times at each location  $\mathbf{x}$  separately.

If the wind  $\mathbf{a}$  is differentiable then form (11.1) describes the same class of problems as

$$u_t + \mathbf{a} \cdot \nabla u = \tilde{g}(u). \quad (11.2)$$

An easy calculation shows that  $\tilde{g} = g$  in the important case where  $\mathbf{a}$  is divergence free (Exercise 11.1). Either (11.1) or (11.2) is an *advection equation*.

Equation (11.2) can be solved by *characteristic curves* [51], namely by solving the ODE system  $d\mathbf{x}/dt = \mathbf{a}(\mathbf{x})$  describing particle motion with velocity  $\mathbf{a}$ . If  $\tilde{g} = 0$  then a solution of (11.2) is constant along such a characteristic curve. In 2D, for example, if  $\mathbf{a} = (\alpha, \beta)$  is constant and  $g = \tilde{g} = 0$  then the solution of (11.1) and (11.2) is

$$u(t, x, y) = u_0(x - \alpha t, y - \beta t) \quad (11.3)$$

for given initial values  $u_0(x, y)$ . In general the solution can be found by solving a system of ODEs which determine the characteristic curve; see Exercise 11.2 and references [51, 115, 119].

In equation (11.1) the concentration  $u$  is conserved in the sense that, for any fixed open set  $V \subset \Omega$  with well-behaved boundary, the divergence theorem implies

$$\frac{d}{dt} \left( \int_V u \right) = - \int_{\partial V} \phi \cdot \mathbf{n} + \int_V g(u), \quad (11.4)$$

where  $\mathbf{n}$  is the outward unit normal along  $\partial V$ . Thus the amount of  $u$  in  $V$  changes only through identified mechanisms, namely by the flux  $\phi$  through  $\partial V$  and creation/removal by the source  $g$ . Conversely, if (11.4) holds for all  $V$  then  $u$  solves (11.1), as long as  $u$  is sufficiently smooth.

The heat equation of Chapter 5 is also a flux conservation equation, but not an advection equation as its flux is proportional to the concentration gradient, i.e., following Fourier's or Fick's law [119]. For such a diffusive flux there is no meaningful velocity. This chapter makes a start on equations combining both flux mechanisms, but we do not consider time-dependent advection-diffusion equations.

The FV approach to discretizing (11.1) replaces it with (11.4) applied on *control volumes*. Though “volumes” suggests three-dimensionality, the 2D, structured-grid control “volumes” in this chapter are rectangles. The discretization in our first example makes three assumptions:

- (i) We enforce (11.4) only for finitely many rectangular control volumes  $V_{ij}$ , each with sides  $h_x, h_y$  and area  $|V| = |V_{ij}| = h_x h_y$  (Figure 11.1).
- (ii) We regard the discrete unknowns as the averages of the concentration over the control volumes,

$$U_{ij}(t) \approx \frac{1}{|V|} \int_{V_{ij}} u(t, x, y) dx dy. \quad (11.5)$$

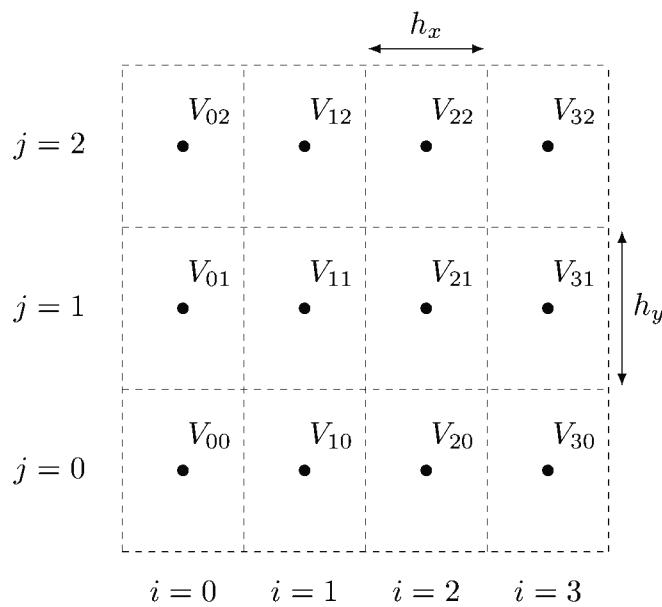
- (iii) Midpoint quadrature is applied to all integrals in (11.4).

Using the midpoint rule on the faces of the control volume, i.e., the midpoints of the segments of  $\partial V_{ij}$ , and denoting the flux components by  $\phi = (\phi^x, \phi^y)$ , we use one flux evaluation per face, signed according to the outward normal. Thus

$$\int_{\partial V_{ij}} \phi \cdot \mathbf{n} \approx h_y \phi_{i+1/2,j}^x + h_x \phi_{i,j+1/2}^y - h_y \phi_{i-1/2,j}^x - h_x \phi_{i,j-1/2}^y. \quad (11.6)$$

Spatial discretizations (11.5) and (11.6) applied to (11.4), plus division by cell area  $|V| = h_x h_y$ , generate the following method of lines (MOL; Chapter 5):

$$U'_{ij} = -\frac{\phi_{i+1/2,j}^x - \phi_{i-1/2,j}^x}{h_x} - \frac{\phi_{i,j+1/2}^y - \phi_{i,j-1/2}^y}{h_y} + g(U_{ij}). \quad (11.7)$$



**Figure 11.1.** Control volumes  $V_{ij}$  with cell centers at  $(x_i, y_j)$ .

Equation (11.7) is not a surprise. Applying a centered FD scheme to (11.1) will yield the same equation. On the other hand, solving (11.7) numerically will require a scheme for the face-center fluxes, and our FV derivation identifies staggered-grid fluxes with a quadrature choice over faces. This way of thinking will suggest increased-accuracy methods through better quadrature, and it relates the discretization to the goal of conservation.

The discrete solution is globally conservative if fluxes have the same values when viewed from either side of a face, as our notation in (11.6) already assumes. In fact, suppose we have periodic boundary conditions on  $\Omega$ . (This applies to our first code; see below.) Summing (11.7) over all cells eliminates the fluxes through cancellation, hence the rate of change of the sum of  $U_{ij}$  depends only on the source term. Then discrete global conservation is analogous to continuum conservation derived from (11.4) when  $\partial\Omega = \emptyset$ :

$$\frac{d}{dt} \left( \sum_{ij} U_{ij} \right) = \sum_{ij} g(U_{ij}) \quad \sim \quad \frac{d}{dt} \left( \int_{\Omega} u \right) = \int_{\Omega} g(u). \quad (11.8)$$

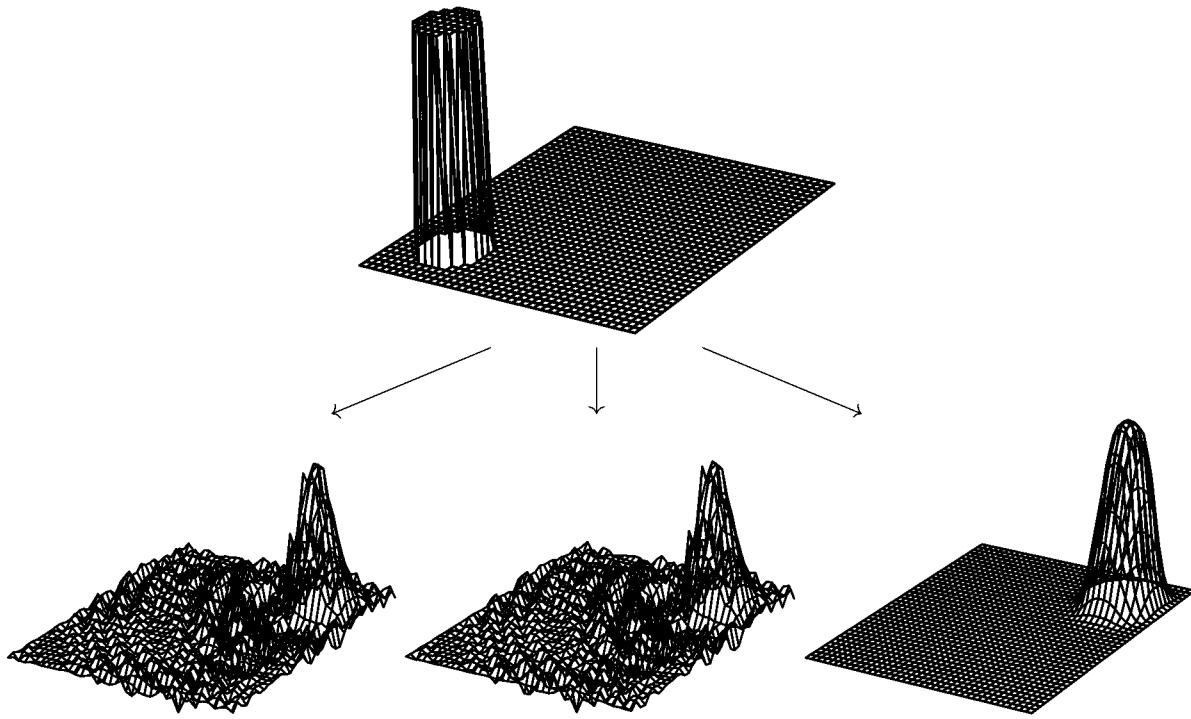
Equation (11.8) is motivation for adding a function which monitors the conserved quantity  $\sum_{ij} U_{ij}$  (Exercise 11.10).

The *centered* flux approximation is an obvious choice for discretization. This formula, for flux  $\phi = au$ , uses the average of the cell-center values,

$$\phi_{i+1/2,j}^x = a^x(x_{i+1/2,j}) \frac{U_{i,j} + U_{i+1,j}}{2}, \quad (11.9)$$

with similar formulas for other face centers. Combining equations (11.7) and (11.9) gives an MOL scheme with second-order truncation error (Exercise 11.3), as expected for the centered FD discretizations in previous chapters.

Sadly, however, this centered scheme is not adequate to the job. In a problem for which the exact solution is simple translation, i.e., equation (11.3), the scheme generates the ugly results in Figure 11.2. The two ugly results differ only in the choice of  $O(\Delta t^2)$  time-stepping methods, namely RK2a and the trapezoid (CN) rule; see Chapter 5. (We compare explicit and implicit methods here to suggest that the issues must be different from the stability concerns for the heat



**Figure 11.2.** Given a discontinuous initial condition (top), the centered flux (11.9) produces poor results with either RK2a (left) or trapezoid/CN (middle) time-stepping. RK2a with a flux-limited high-resolution flux (11.25) (right) approximates the (translation) exact solution better.

equation.) The rightmost figure shows that RK2a time-stepping and a different flux discretization, as described next, does a much better job. Apparently some effort is needed in dealing with advection!

## Upwinding, and the goals of advection schemes

Method-of-lines (MOL) system (11.7) is not actually a “method” until we determine how the face-centered fluxes are computed, and then we choose a time-stepping scheme. The first choice would seem to be easy because the flux formula  $\phi = au$  is so simple, but difficulties are already apparent in the constant velocity case. The reason is that translation is not a smoothing process, thus a discrete version of translation can only be of high quality if all frequencies (spatial modes) present in the solution are

- well approximated on the grid and
- translated at identical rates.

The noise seen in Figure 11.2 for flux discretization (11.9) arises from the poor gridded approximation of high-frequency modes as well as translation of these modes at the wrong speeds (*dispersion*).

Some amount of *numerical diffusion*, i.e., diffusion present in the numerical scheme but not in the PDE, is desirable because it can hide translation and approximation errors by decaying them away. All successful advection schemes hide high-frequency translation errors in this way, but the effect must be used sparingly. Our best results, measured as usual by verification against an exact solution, but also in an informal and visual sense, will come from schemes which will use targeted numerical diffusion at locations where the numerical solution is changing rapidly.

These *high-resolution* schemes are modifications of *first-order upwinding*, introduced momentarily, which has plenty of diffusion. The high-resolution schemes add flux corrections to the first-order upwinding formula at places where the solution is smooth. Where the solution has an accurate gridded approximation the scheme therefore has little numerical diffusion. However, where such methods detect rapid spatial variations the formulas turn off the corrections and only apply diffusive first-order upwinding. This approach, which is necessarily nonlinear, is appropriate even for the simplest linear advection equations like (11.2).

First-order upwinding, also called the *donor cell method* [103], calculates the face-centered flux  $\phi = au$  by choosing a solution value according to the sign of the wind:

$$\begin{aligned}\phi_{i+1/2,j}^x &= a_{i+1/2,j}^x \begin{cases} U_{i,j}, & a_{i+1/2,j}^x \geq 0, \\ U_{i+1,j}, & a_{i+1/2,j}^x < 0, \end{cases} \\ &= \max(a_{i+1/2,j}^x, 0) U_{i,j} + \min(a_{i+1/2,j}^x, 0) U_{i+1,j}.\end{aligned}\quad (11.10)$$

Similar formulas apply to the other fluxes appearing in (11.7) (Exercise 11.4).

Consider the 1D case and suppose the wind is a positive constant  $\alpha > 0$ :

$$u_t + \alpha u_x = 0. \quad (11.11)$$

Spatial discretization (11.7) and first-order upwinding yield the MOL system

$$U'_i + \alpha \frac{U_i - U_{i-1}}{h} = 0. \quad (11.12)$$

We can make three closely related observations about (11.12):

- (i) Applying forward Euler time-stepping yields

$$U_i^{n+1} = (1 - \gamma)U_i^n + \gamma U_{i-1}^n, \quad (11.13)$$

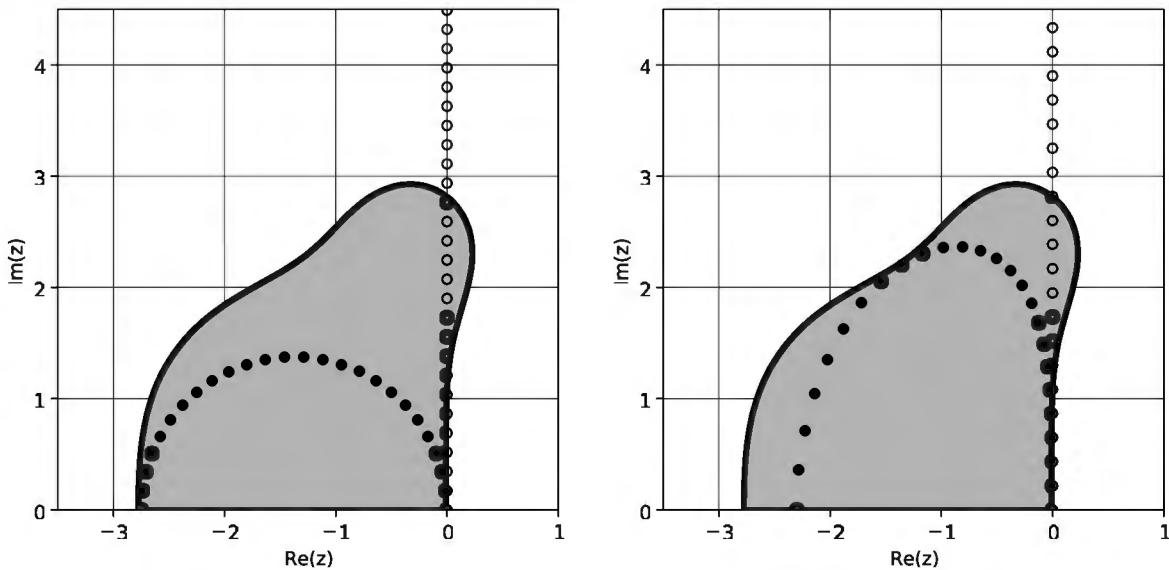
where  $\gamma = \Delta t \alpha / h$ . If  $0 \leq \gamma \leq 1$  then this formula computes the new value  $U_i^{n+1}$  as an average of the current values  $U_i^n, U_{i-1}^n$ . The condition  $\gamma \geq 0$  is automatic with upwinding, but the condition  $\gamma \leq 1$ , required so that the coefficients in (11.13) are nonnegative, is a restriction on the size of the time step  $\Delta t$ . If (11.13) is subject to this restriction and if  $U_j^n \geq 0$  for all  $j$  then  $U_j^{n+1} \geq 0$  also. Oscillations of the type seen in Figure 11.2, where a nonnegative initial condition evolves into an oscillatory state of both signs, cannot occur.

For 2D upwinding (11.10) the condition generalizes to

$$\Delta t \max \left\{ \frac{|a_x^x|}{h_x}, \frac{|a_y^y|}{h_y} \right\} \leq 1 \quad (11.14)$$

(see Exercise 11.4). In their 1928 paper [38], Courant, Friedrichs, and Lewy (CFL) interpreted (11.14) as requiring that the (approximate) characteristic curves from  $(t_{n+1}, x_i, y_j)$  pass through the interior of the stencil at  $t = t_n$ , allowing interpolation to compute an updated value from the current (time  $t_n$ ) gridded values. Many advection schemes, including first-order upwind (11.12), centered fluxes (11.9), and the third-order upwind-biased scheme (11.22) below, can be interpreted as arising from interpolation [115]. The *CFL condition* (11.14) keeps these formulas from becoming extrapolation (Exercise 11.5).

- (ii) MOL equations (11.12) form a linear ODE system  $U' = AU$ . With periodic boundary conditions on the interval  $(-1, 1)$ , and a grid of  $m$  points with spacing  $h = 2/m$ , the



**Figure 11.3.** The exact eigenvalues of PDE (11.11) (open circles) are imaginary. Those from first-order upwinding on the MOL system (11.12) (left; solid) or a third-order upwind-biased scheme (11.22) (right; solid) are damped by negative real parts. The absolute stability region of RK4 is shaded.

matrix is

$$A = -\frac{\alpha}{h} \begin{bmatrix} 1 & & & -1 \\ -1 & 1 & & \\ & -1 & 1 & \\ & & \ddots & \ddots \\ & & & -1 & 1 \end{bmatrix} \quad (11.15)$$

with (exact) eigenvalues

$$\mu_p = -i\frac{\alpha}{h} \sin(\pi ph) - \frac{\alpha}{h} (1 - \cos(\pi ph)) \quad (11.16)$$

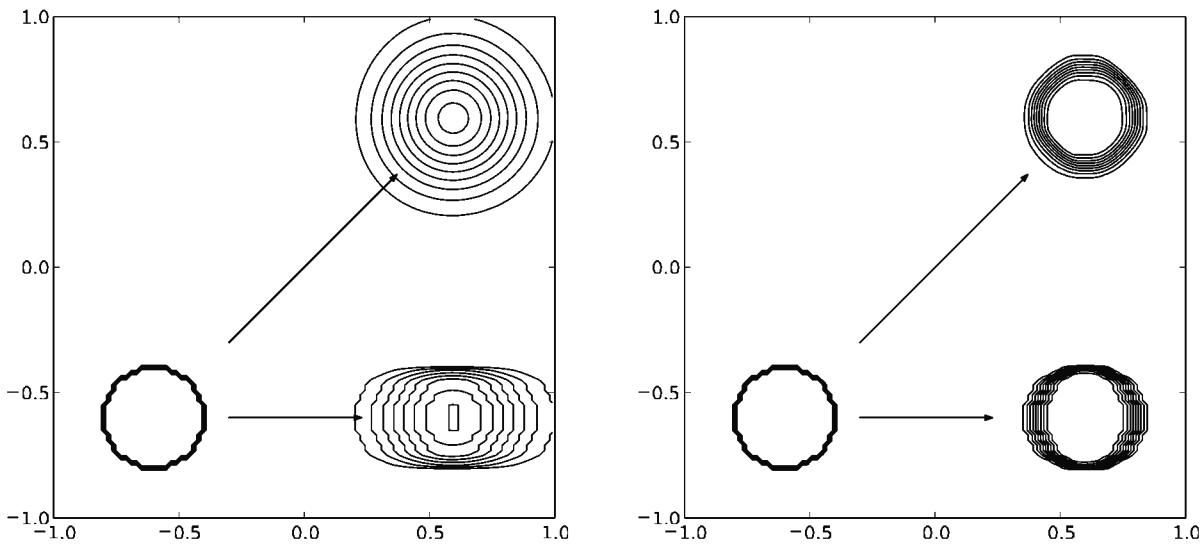
for  $p = 1, \dots, m$ . (See [104] and the eigenvalue calculation in Exercise 11.3.) For comparison, the exact eigenvalues of advection equation (11.11) can be found by Fourier series:  $\lambda_p = -i\alpha p \pi$ , for all integers  $p$ .

Now recall that a time-stepping scheme for  $U' = AU$  is absolutely stable if the scaled eigenvalues of  $A$  are inside the scheme's stability region (Chapter 5). In the example shown in Figure 11.3, using RK4 time-stepping for illustration, the discrete first-order upwind eigenvalues are inside the region. However, the figure also shows that these discrete eigenvalues have significantly negative real parts, representing substantial damping which is not present in the PDE.

The figure suggests the goals in designing stable, higher-order advection schemes. High-frequency modes of the MOL system must indeed have eigenvalues inside the absolute stability region, but superior schemes will also push more eigenvalues toward their exact values on the imaginary axis. Figure 11.3 shows that scheme (11.22) below does this.

- (iii) Scheme (11.12) has truncation error  $O(h^1)$  as an approximation of (11.11). However, the same scheme is an  $O(h^2)$  approximation to a different PDE, namely

$$v_t + \alpha v_x = \frac{\alpha h}{2} v_{xx}, \quad (11.17)$$



**Figure 11.4.** Translation computed by first-order upwinding and RK2a time-stepping (left) over-smoothes especially in the direction of motion. A high-resolution flux-limited method (11.25) does much better (right).

an advection-diffusion equation with diffusivity constant  $D = \alpha h/2 > 0$ . (Such an observation is a *modified equation analysis* [84] of the scheme.) As  $h \rightarrow 0$  the solutions to (11.12) will converge to the solution of advection equation (11.11), because  $D \rightarrow 0$ , but along the way they are higher-quality approximations of (11.17). In fact, the centered difference scheme applied to (11.17) is identical to the upwind scheme (11.12) applied to (11.11); compare Exercise 11.7.

These three observations together say that the first-order upwind scheme (11.12) is stable when subject to the CFL condition but that it is excessively smoothing. The numerical results in the left part of Figure 11.4, including results for the same diagonal-translation problem as in Figure 11.2, confirm this. On the other hand, at least there is no hint of oscillation in the figure, as would be the case with the centered flux scheme.

Overdamping, as occurs in first-order upwinding, is not a good strategy for high-quality numerical advection. Instead we want many modes, those with medium spatial frequency, to be translated at nearly the correct rates with minimal damping. At the same time, the highest-frequency modes must indeed be damped because we will not be able to translate them at the right rates; they are not faithfully represented on the grid. Thus first-order upwinding is kept as a useful tool, but we apply it in a targeted manner while also improving the order of accuracy in most locations.

An alternate approach would be to seek schemes for (11.11) with no damping whatsoever. The *leapfrog* scheme [115] is a well-known example. It uses a centered spatial difference, giving purely imaginary eigenvalues for the MOL system, and the midpoint rule in time. As an ODE scheme, the midpoint rule has a segment of the imaginary axis as its stability region [104], and the CFL condition puts the discrete eigenvalues inside this segment. However, because high-frequency modes are (inevitably) translated at the wrong speeds, and because there is no damping to hide this error, one sees obvious oscillations in typical leapfrog results (Exercise 11.3).

A further alternative uses high-degree trigonometric or polynomial approximations. While such spectral methods can achieve superior advection results [142], they are beyond our scope and more difficult to use in parallel.

## High-resolution flux discretizations

*Nonoscillatory discretizations* [84], also known as *flux-limiters* or *high-resolution schemes*, are built upon first-order upwinding. To explain how, we present the ideas here in 1D and then implement them in 2D in the next section.

For the linear flux-conservation equation

$$u_t + f_x = 0, \quad (11.18)$$

centered spatial finite differences gives an MOL system

$$U'_i + \frac{f_{i+1/2} - f_{i-1/2}}{h} = 0. \quad (11.19)$$

Suppose  $f = a(x)u$  with continuous wind  $a(x)$ . To specify an MOL scheme the face-center fluxes

$$f_{i+1/2} = a(x_{i+1/2}) u(t, x_{i+1/2})$$

must be approximated by a function of the grid values  $U_i(t) \approx u(t, x_i)$ . Note we evaluate  $a(x)$  at the staggered-grid points  $x_{i+1/2}$  in all cases; we denote  $a = a(x_{i+1/2})$ .

Both first-order upwinding (11.10),

$$f_{i+1/2} = \begin{cases} a U_i, & a \geq 0, \\ a U_{i+1}, & a < 0, \end{cases} \quad (11.20)$$

and the centered flux formula

$$f_{i+1/2} = a \frac{U_i + U_{i+1}}{2} \quad (11.21)$$

use only the two regular-grid values  $U_i, U_{i+1}$  straddling  $x_{i+1/2}$ . We may also expand the stencil and consider the *third-order upwind-biased* formula

$$f_{i+1/2} = \begin{cases} \frac{1}{6}a(-U_{i-1} + 5U_i + 2U_{i+1}), & a \geq 0, \\ \frac{1}{6}a(2U_i + 5U_{i+1} - U_{i+2}), & a < 0. \end{cases} \quad (11.22)$$

This formula may use all four values  $U_{i-1}, U_i, U_{i+1}, U_{i+2}$  nearest to  $x_{i+1/2}$ .

Although it is not so obvious, it is true that both formulas (11.21) and (11.22) modify (11.20) by adding a higher-order correction:

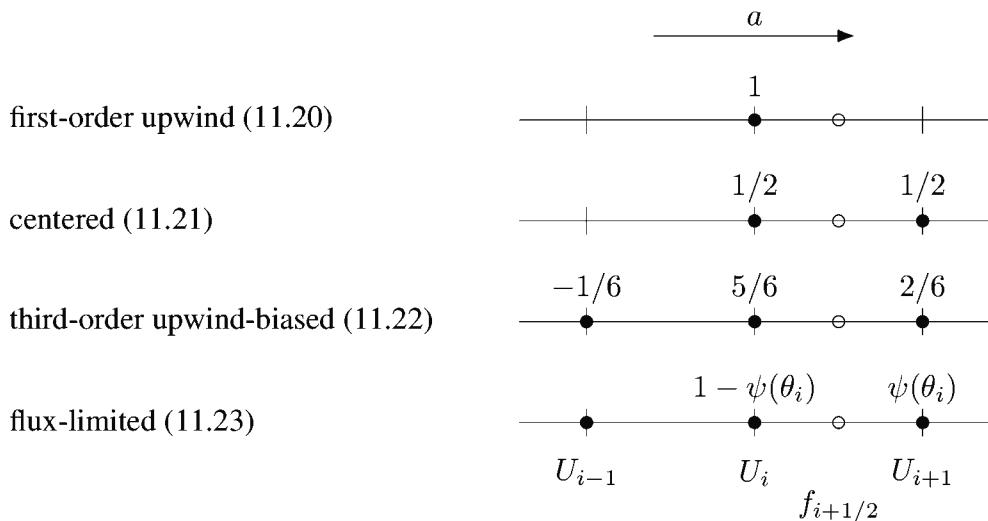
$$f_{i+1/2} = \begin{cases} a[U_i + \psi(\theta_i)(U_{i+1} - U_i)], & a \geq 0, \\ a[U_{i+1} + \psi(1/\theta_{i+1})(U_i - U_{i+1})], & a < 0. \end{cases} \quad (11.23)$$

Here  $\psi(\theta)$  is a nonnegative function called the *flux-limiter*, defined for all  $\theta \in \mathbb{R}$ , and  $\theta_i$  is a ratio which quantifies the variation of  $U$  around  $x_i$ :

$$\theta_i = \frac{U_i - U_{i-1}}{U_{i+1} - U_i}. \quad (11.24)$$

The second case in (11.23) reflects the first case, as though viewed from the other side. Observe that if  $U_i = U_{i+1}$  then  $\theta_i$  is not defined, but that in that case we define (11.23) as computing a zero correction since the other factor is zero.

If  $U_i$  is either the minimum or maximum of the values  $\{U_{i-1}, U_i, U_{i+1}\}$ , so that the slope changes sign at  $x_i$ , then  $\theta_i \leq 0$ . Thus, to turn off the higher-order correction near extrema we



**Figure 11.5.** When  $a = a(x_{i+1/2}) \geq 0$ , schemes (11.23) all use the same three regular grid points, with coefficients as shown.

require  $\psi(\theta) = 0$  for  $\theta \leq 0$ . On the other hand, if the slope of the piecewise-linear function defined by  $\{U_i\}$  is nearly equal to a nonzero constant then  $\theta_i \approx 1$ .

First-order upwinding (11.20) uses no correction, i.e.,  $\psi(\theta) = 0$ , the centered flux (11.21) uses  $\psi(\theta) = 1/2$ , and the third-order upwind-biased flux (11.22) uses  $\psi(\theta) = 1/3 + (1/6)\theta$  (Exercise 11.8). Note that when  $a \geq 0$  then, for any function  $\psi$ , (11.23) uses the three-point stencil  $\{U_{i-1}, U_i, U_{i+1}\}$  (Figure 11.5).

We may say that a piecewise-linear function represented by grid values  $\{U_j\}$  is *smooth at  $x_i$*  if the change in slope is small in the sense that  $\theta_i$  is roughly equal to one,  $\theta_i \approx 1$ . Because a higher-order correction to first-order upwinding is desirable wherever the solution is smooth, we want  $\psi(\theta)$  to match a second- or third-order formula when  $\theta_i \approx 1$ .

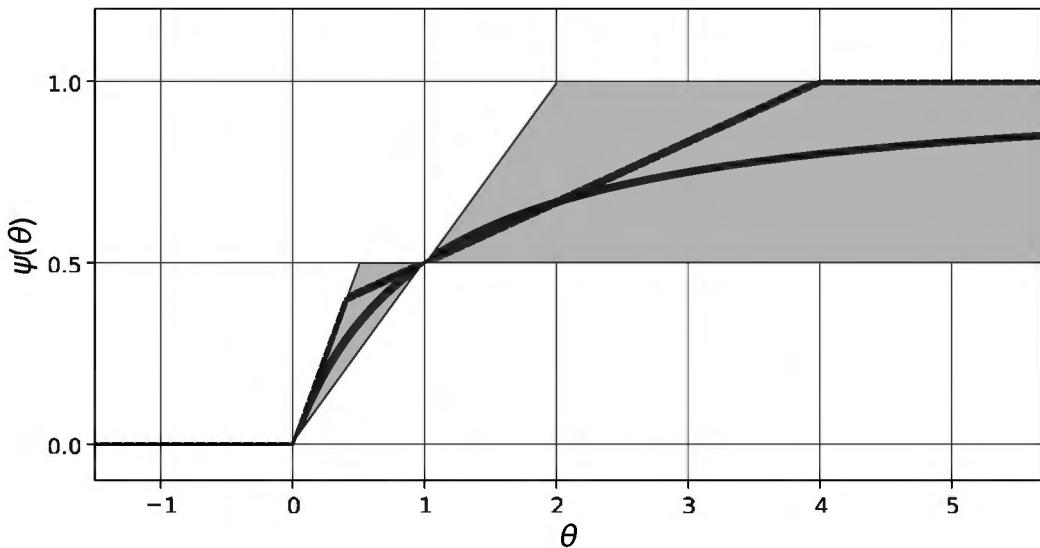
Thus, as an informal definition, an MOL flux-conservation scheme for advection is *nonoscillatory* or *high resolution* if it has at least second-order accuracy in regions where the solution is smooth and if it captures discontinuities of  $u(x, t)$  as “narrow, monotone structures” [148] instead of generating oscillations with multiple minima and maxima. When used without modification the second- and third-order linear schemes (11.21) and (11.22) both propagate oscillations (e.g., Figure 11.2) when  $u(x, t)$  is nonsmooth, and thus they do not satisfy the definition. High-resolution methods are constructed from the additional insight that the corrections to first-order upwinding made in oscillatory higher-order linear schemes are too large near discontinuities. These corrections should be limited, as in (11.23), by making the value of  $\psi(\theta)$  smaller both when  $\theta \rightarrow 0$  and when  $\theta \rightarrow \infty$ . In fact, linearity of the scheme is at odds with this high-resolution goal for the corrections, and *Godunov’s barrier theorem* [62] (below) confirms this idea.

In (11.23) the flux itself is not actually limited because the first-order flux can be arbitrary. However, the higher-order correction, the part which is proportional to  $\psi(\theta_i)$ , is limited. A better name for  $\psi$  would be “flux-correction-limiter” [148] but “flux-limiter” is traditional.

We propose to test the two high-resolution flux-limiters shown in Figure 11.6. The first is the *Koren limiter* [97],

$$\psi(\theta) = \max \left\{ 0, \min \left\{ 1, \frac{1}{3} + \frac{1}{6}\theta, \theta \right\} \right\}. \quad (11.25)$$

On the interval  $2/5 \leq \theta \leq 4$  around  $\theta = 1$  we have  $\psi(\theta) = 1/3 + (1/6)\theta$  in agreement with (11.22). Thus we get third-order accuracy where the solution is smooth. Because  $\psi(\theta) \rightarrow 0$  as  $\theta \rightarrow 0$  and  $\psi(\theta)$  is bounded as  $\theta \rightarrow \infty$  the correction is limited when the solution is bumpier.



**Figure 11.6.** The Koren (11.25) (dashed) and van Leer (11.26) (solid) limiters are two of many possible flux-limiters  $\psi(\theta)$ . Flux-limiters give second-order results and avoid spurious oscillations if they lie in the shaded Sweby region [138].

However, the Koren limiter  $\psi(\theta)$  is not differentiable, which explains why we also test the *van Leer limiter* [147]:

$$\psi(\theta) = \frac{1}{2} \frac{\theta + |\theta|}{1 + |\theta|}. \quad (11.26)$$

The resulting scheme does not have third-order accuracy, but it is second order where the solution is smooth because at least  $\psi(1) = 1/2$ .

To explain the shaded region shown in Figure 11.6, consider a periodic grid with solution values  $\{U_i\}_{i=0}^{m-1}$  and  $U_m = U_0$ . Define

$$\text{TV}(U) = \sum_{i=1}^m |U_i - U_{i-1}| \quad (11.27)$$

as the *total variation* of  $U$ . A scheme for  $u_t + a(x)u_x = 0$  is *total variation diminishing* (TVD) if  $\text{TV}(U)$  is nonincreasing in time [76]. Because periodic solutions of the continuum problem also have constant total variation (Exercise 11.9), a TVD scheme mimics a property of the exact solution.

A scheme of form (11.23), which

- (i) is TVD,
- (ii) is at least second-order for smooth functions, and
- (iii) avoids evolving sine wave solutions into square waves (i.e., is not “overcompressive” [84]),

must use a limiting function  $\psi(\theta)$  with graph in the shaded region in Figure 11.6 [76, 138]. In particular, one can rigorously show that a scheme of form (11.23) which satisfies (i) and (ii) must have  $0 \leq \psi(\theta) \leq 1$  and  $0 \leq \psi(\theta) \leq \theta$ . This defines two of the several boundaries of the shaded region, but the other boundaries arise, at least in part, from considering numerical results [138].

The use of such relatively complicated high-resolution schemes is justified by the following theorem.

**Theorem 11.1.** (Godunov’s barrier theorem [62]) *A monotonicity-preserving linear scheme for the 1D constant-coefficient equation  $u_t + \alpha u_x = 0$  cannot have second-order (or higher) local truncation error in  $x$ .*

To explain this statement, we define a grid function  $\{U_i\}$  to be *monotone* if either  $U_i \leq U_{i+1}$  for all  $i$  or  $U_i \geq U_{i+1}$  for all  $i$ . A numerical scheme is *monotonicity preserving* if  $\{U_i^n\}$  monotone implies  $\{U_i^{n+1}\}$  monotone. Note that TVD implies monotonicity preserving but not vice versa [103].

Godunov's theorem, which applies to both one-step and multistep time-integration [153], excludes *linear* second-order TVD schemes. The above high-resolution methods overcome this barrier by incorporating an appropriately bounded limiter, thus the schemes are nonlinear maps from one time step to the next.

There are of course many high-resolution limiters other than the two above [84], and there are also other ways of describing high-resolution schemes. For example one may “slope limit” a reconstructed solution within a finite volume cell [103]. In any case, as we show next in a 2D implementation, the numerical results produced by flux-limiters justify their relative complexity.

## Implementation in 2D

Our program `advect.c` solves (11.1) on the square  $\Omega = (-1, 1)^2$ , with periodic boundary conditions, using the structured grid of  $m_x \times m_y$  points shown in Figure 11.1. Spatial discretization again leads to form (11.7). A flux discretization—we will test all the possibilities discussed above—then generates an MOL ODE system of dimension  $N = m_x m_y$ :

$$\mathbf{U}' = \mathbf{G}(t, \mathbf{U}), \quad \mathbf{U}(t) = \{U_{i,j}(t)\} \in \mathbb{R}^N. \quad (11.28)$$

With nontrivial flux-limiting, the right-hand side function  $\mathbf{G}$  is nonlinear in  $\mathbf{U}$ .

The way `advect.c` uses DMDA and TS objects is similar to `heat.c` in Chapter 5. As with that code we may compare different time-stepping methods at run time, using a TS option to choose. If the method is implicit, TS will build an internal SNES/KSP/PC “stack” of solver components (e.g., Figure 5.5), while for an explicit scheme the TS sets up no solver stack, but our code is the same either way.

```
typedef enum {STRAIGHT, ROTATION} ProblemType;
static const char *ProblemTypes[] = {"straight", "rotation",
                                     "ProblemType", "", NULL};

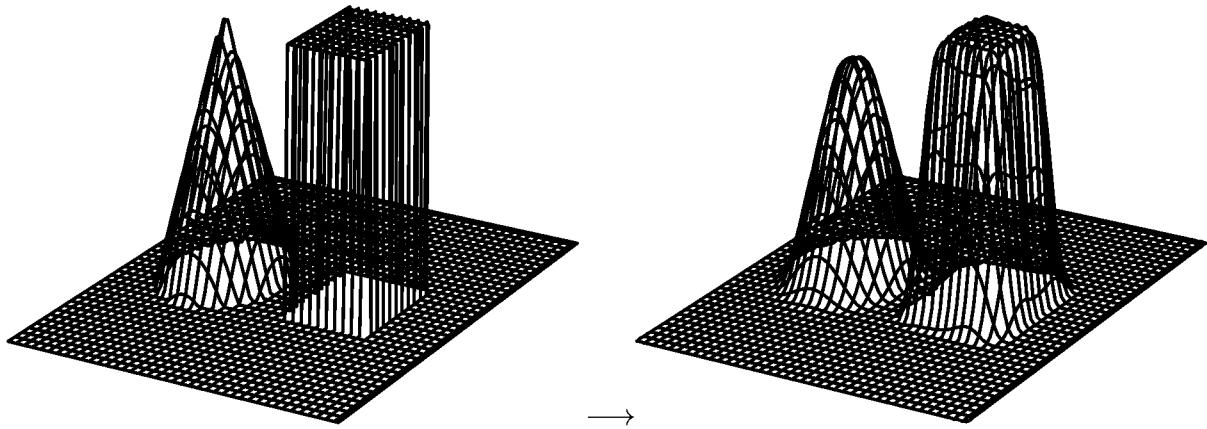
typedef enum {STUMP, SMOOTH, CONE, BOX} InitialType;
static const char *InitialTypes[] = {"stump", "smooth", "cone", "box",
                                     "InitialType", "", NULL};

typedef enum {NONE, CENTERED, VANLEER, KOREN} LimiterType;
static const char *LimiterTypes[] = {"none", "centered", "vanleer", "koren",
                                     "LimiterType", "", NULL};

typedef struct {
    ProblemType problem;
    PetscReal windx, windy,           // x,y velocity in STRAIGHT
    (*initial_fcn)(PetscReal, PetscReal), // for STRAIGHT
    (*limiter_fcn)(PetscReal), // limiter used in RHS
    (*jac_limiter_fcn)(PetscReal); // used in Jacobian
} AdvectCtx;
```

**Code 11.1.** `c/ch11/advect.c`, part I. Enumerate types and a context struct.

We show extracts from `advect.c` in Codes 11.1–11.4. The first shows a context struct along with type declarations for use with `PetscOptionsEnum()`. For each such option one defines a C `enum` type [90], plus an array of strings, including some for the `-help` documentation. Here we solve two problems in form (11.1), each with  $g = 0$ , namely `-adv_problem`



**Figure 11.7.** In problem `-adv_problem rotation` the initial condition (left) combines “cone” and “box” functions. With `-ts_max_time 6.2832`, i.e. for  $0 \leq t \leq 2\pi$ , the solution is one complete rotation. The combination of the Koren (11.25) flux-limiter and third-order adaptive RK time-stepping gives this good-looking result (right) on a relatively coarse grid.

`straight|rotation`. The default problem `straight` (Figure 11.2) has constant wind  $\mathbf{a} = \langle 2, 2 \rangle$  while `rotation` (Figure 11.7) has  $\mathbf{a} = \langle y, -x \rangle$ , a rigid rotation in the plane. Figures 11.2 and 11.7 also show three of the four possible initial conditions, namely `-adv_initial stump|smooth|cone|box`; the corresponding formulas are in Code 11.2.

```
// equal to 1 in a disc of radius 0.2 around (-0.6,-0.6)
static PetscReal stump(PetscReal x, PetscReal y) {
    const PetscReal r = PetscSqrtReal((x+0.6)*(x+0.6) + (y+0.6)*(y+0.6));
    return (r < 0.2) ? 1.0 : 0.0;
}

// smooth (C^6) version of stump
static PetscReal smooth(PetscReal x, PetscReal y) {
    const PetscReal r = PetscSqrtReal((x+0.6)*(x+0.6) + (y+0.6)*(y+0.6));
    if (r < 0.2)
        return PetscPowReal(1.0 - PetscPowReal(r / 0.2,6.0),6.0);
    else
        return 0.0;
}

// cone of height 1 of base radius 0.35 centered at (-0.45,0.0)
static PetscReal cone(PetscReal x, PetscReal y) {
    const PetscReal r = PetscSqrtReal((x+0.45)*(x+0.45) + y*y);
    return (r < 0.35) ? 1.0 - r / 0.35 : 0.0;
}

// equal to 1 in square of side-length 0.5 (0.1,0.6) x (-0.25,0.25)
static PetscReal box(PetscReal x, PetscReal y) {
    if ((0.1 < x) && (x < 0.6) && (-0.25 < y) && (y < 0.25))
        return 1.0;
    else
        return 0.0;
}

typedef PetscReal (*PointwiseFcn)(PetscReal,PetscReal);

static PointwiseFcn initialptr[] = {&stump, &smooth, &cone, &box};
```

**Code 11.2.** `c/ch11/advect.c`, part II. Initial conditions.

Code 11.3 shows the implementations of the limiters: `-adv_limiter none|centered|vanleer|koren`. Note that the `none` limiter is simply a NULL pointer.

```

/* the centered-space method is linear */
static PetscReal centered(PetscReal theta) {
    return 0.5;
}

/* van Leer (1974) limiter is formula (1.11) in section III.1 of
Hundsdorfer & Verwer (2003) */
static PetscReal vanleer(PetscReal theta) {
    const PetscReal abstheta = PetscAbsReal(theta);
    return 0.5 * (theta + abstheta) / (1.0 + abstheta);
}

/* Koren (1993) limiter is formula (1.7) in section III.1 of
Hundsdorfer & Verwer (2003) */
static PetscReal koren(PetscReal theta) {
    const PetscReal z = (1.0/3.0) + (1.0/6.0) * theta;
    return PetscMax(0.0, PetscMin(1.0, PetscMin(z, theta)));
}

typedef PetscReal (*LimiterFcn)(PetscReal);

static LimiterFcn limiterptr[] = {NULL, &centered, &vanleer, &koren};

```

**Code 11.3.** *c/ch11/advect.c, part III. Flux-correction limiters  $\psi(\theta)$ .*

The right-hand-side function `FormRHSFunctionLocal()` computes  $\mathbf{G}(t, \mathbf{U})$  in system (11.28) (Code 11.4). For explicit time-stepping schemes this is almost the entire implementation. However, for implicit time-stepping we have also implemented the Jacobian of  $\mathbf{G}$  in `FormRHSJacobianLocal()` (not shown, but addressed below).

```

PetscErrorCode FormRHSFunctionLocal(DMDALocalInfo *info, PetscReal t,
    PetscReal **au, PetscReal **aG, AdvectCtx *user) {
    PetscInt i, j, q, dj, di;
    PetscReal hx, hy, halfx, halfy, x, y, a,
        u_up, u_dn, u_far, theta, flux;

    // clear G first
    for (j = info->ys; j < info->ys + info->ym; j++)
        for (i = info->xs; i < info->xs + info->xm; i++)
            aG[j][i] = 0.0;
    // fluxes on cell boundaries are traversed in E,N order with indices
    // q=0 for E and q=1 for N; cell center has coordinates (x,y)
    hx = 2.0 / info->mx; hy = 2.0 / info->my;
    halfx = hx / 2.0; halfy = hy / 2.0;
    for (j = info->ys-1; j < info->ys + info->ym; j++) { // note -1 start
        y = -1.0 + (j+0.5) * hy;
        for (i = info->xs-1; i < info->xs + info->xm; i++) { // -1 start
            x = -1.0 + (i+0.5) * hx;
            if ((i >= info->xs) && (j >= info->ys)) {
                aG[j][i] += g_source(x,y,au[j][i],user);
            }
            for (q = 0; q < 2; q++) { // E (q=0) and N (q=1) bdry fluxes
                if (q == 0 && j < info->ys) continue;
                if (q == 1 && i < info->xs) continue;
                di = 1 - q;
                if (di > 0) aG[j][i] += flux;
            }
        }
    }
}

```

```

dj = q;
a = a_wind(x + halfx*di, y + halfy*dj, q, user);
// first-order flux
u_up = (a >= 0.0) ? au[j][i] : au[j+dj][i+di];
flux = a * u_up;
// use flux-limiter
if (user->limiter_fcn != NULL) {
    // formulas (1.2), (1.3), (1.6); H&V pp 216--217
    u_dn = (a >= 0.0) ? au[j+dj][i+di] : au[j][i];
    if (u_dn != u_up) {
        u_far = (a >= 0.0) ? au[j-dj][i-di]
                             : au[j+2*dj][i+2*di];
        theta = (u_up - u_far) / (u_dn - u_up);
        flux += a * (*user->limiter_fcn)(theta)
                * (u_dn - u_up);
    }
}
// update owned G_ij on both sides of computed flux
if (q == 0) {
    if (i >= info->xs)
        aG[j][i] -= flux / hx;
    if (i+1 < info->xs + info->xm)
        aG[j][i+1] += flux / hx;
} else {
    if (j >= info->ys)
        aG[j][i] -= flux / hy;
    if (j+1 < info->ys + info->ym)
        aG[j+1][i] += flux / hy;
}
}
}
return 0;
}

```

**Code 11.4.** *c/ch11/advect.c, part IV. Right side function  $\mathbf{G}(t, \mathbf{U})$  in (11.28).*

The flux discretizations in `advect.c` use 2D versions of limiter formula (11.23). As shown in Figure 11.8, this generally needs a nine-point, star-type stencil with width (Chapter 3) of two. To supply a value for each control-volume face center, first-order upwinding merely identifies a regular grid point which we call `up`. The flux-limited schemes also need a `far` and a `dn` (downwind) point for each face center. See Code 11.4 for the implementation.

The structured-grid DMDA object is created by this call:

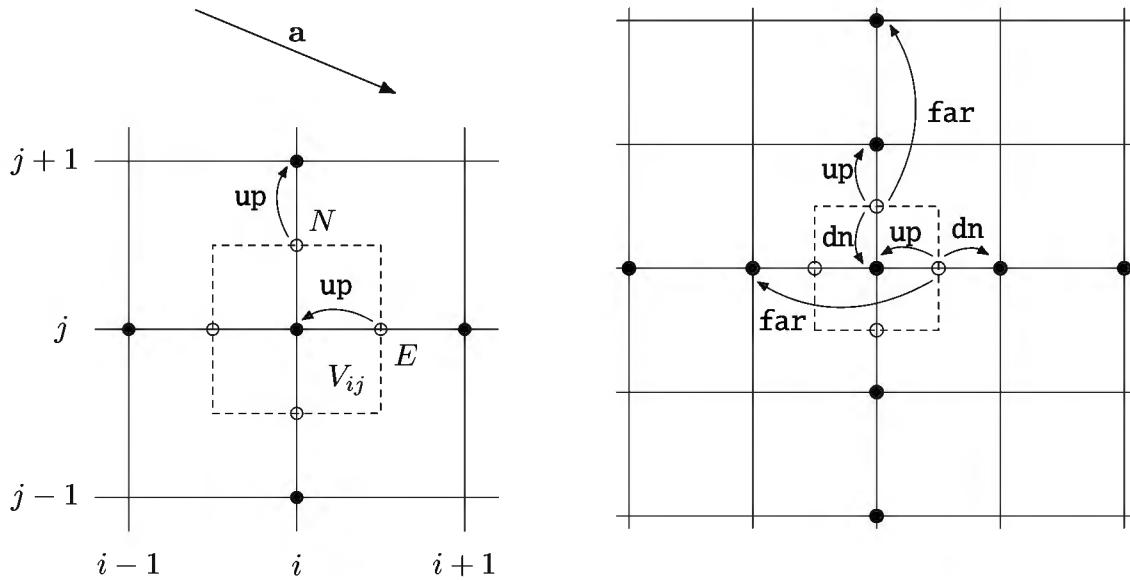
```

DMDACreate2d(PETSC_COMM_WORLD,
    DM_BOUNDARY_PERIODIC, DM_BOUNDARY_PERIODIC,
    DMDA_STENCIL_STAR, 5, 5, PETSC_DECIDE, PETSC_DECIDE,
    1, 2, NULL, NULL, &da);

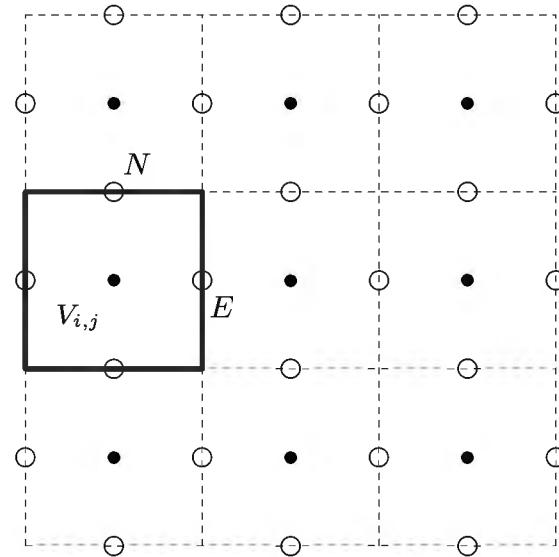
```

The default periodic grid has five points in each direction so that all nine grid points in the stencil (Figure 11.8, right) are distinct, thus that finite-difference-by-coloring evaluation of the Jacobian (Chapter 4) can be used.

When computing the right-hand-side function  $\mathbf{G}(t, \mathbf{U})$  in (11.28) we evaluate each face-center flux only once, as follows. For volume  $V_{ij}$  we label the face centers  $(x_{i+1/2}, y_j)$  as “*E*” (east) and  $(x_i, y_{j+1/2})$  as “*N*” (north); see Figure 11.9. Then `FormRHSFunctionLocal()` traverses the locally owned portion of the grid using indices  $i$  and  $j$ , computing the *E* and *N* fluxes for each control volume. It is not necessary to save these flux values in an array because we can immediately add the contribution to the correct  $G_{kl}$ . This technique requires redundant



**Figure 11.8.** Assume the wind  $\mathbf{a}$  is directed as shown at top. First-order upwind (left) and flux-limited (right) schemes make decisions at  $E$  and  $N$  face centers (circles) based on a star stencil.



**Figure 11.9.** Every regular-grid point (solid) is at the center of a control volume  $V_{i,j}$  (outlined in gray), which has four face-center (staggered-grid) points. Fluxes at  $E$  and  $N$  face-center points are computed once for each pair  $i, j$ .

evaluation only along the boundaries of each process-owned subdomain. Note that periodic ghost points are used to compute the fluxes along the edges of the global domain  $\Omega$ .

There is little more to say about the implementation. As usual, `main()` starts by checking options and creating the DMDA and TS objects. Because performance measurements suggest it is fastest (next), we set the default TS type to Runge-Kutta. We choose an initial time step from CFL condition (11.14), but after that the adaptive time-steppers determine the time step according to their internal estimates of one-step error.

The last action of `main()` is to check whether the exact solution of the problem is known, and measure the numerical error if so. Specifically, if the problem is `straight`, the final

time  $t_f$  is an integer, and the velocity components are even integers, then the initial condition  $w(x, y) = u(0, x, y)$  is also the exact solution because of periodic wrapping. In that case a single `VecNorm()` call with norm type `NORM_1_AND_2`, followed by a scaling, gives the  $L^1$  and  $L^2$  error norms:

$$\begin{aligned}\|U - w\|_{1,h} &= h_x h_y \sum_{i,j} |U_{ij} - w(x_i, y_j)|, \\ \|U - w\|_{2,h} &= \left( h_x h_y \sum_{i,j} |U_{ij} - w(x_i, y_j)|^2 \right)^{1/2}.\end{aligned}\tag{11.29}$$

(Compare Exercise 11.6.)

## Advection results

Let us make sure that `advect.c` produces reasonable-looking and convergent results. First we visualize two laps of diagonal motion on an  $80 \times 80$  grid:

```
| $ cd c/ch11/ && make advect
| $ ./advect -da_refine 4 -ts_max_time 2.0 -ts_monitor_solution draw
```

This is the `straight` problem with a discontinuous `stump` initial condition, namely using defaults

```
-adv_problem straight -adv_initial stump -adv_windx 2.0 -adv_windy 2.0
```

Regarding numerical choices, the defaults are the Koren limiter (11.25) and the adaptive Runge-Kutta 3bs scheme. The results look good, visually identical to those in Figure 11.4 (right); Figure 11.2 shows results for the same problem but without a flux limiter.

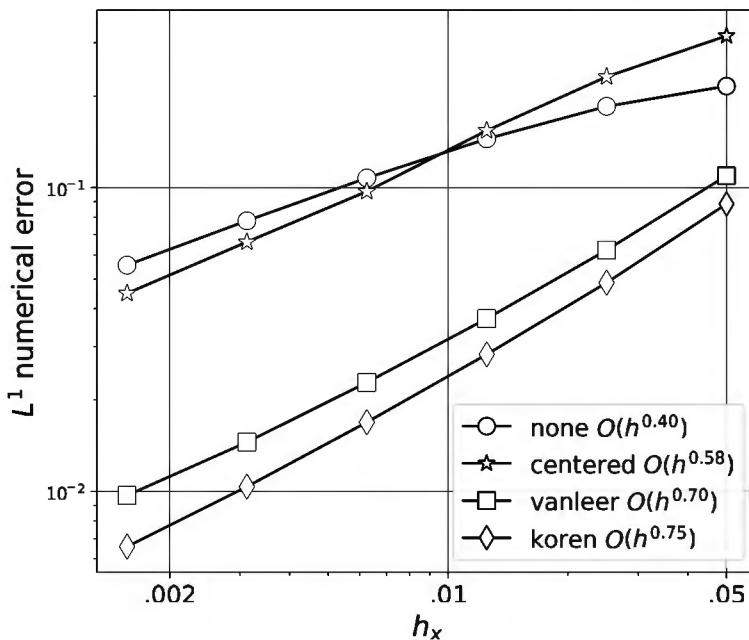
On a given grid the numerical error, and even the wall clock time, can be relatively insensitive to the choice of adaptive time-stepping method. For example, restricting the comparison to the Runge-Kutta family (Chapter 5) for simplicity, we measure the (serial) run time and error norms for a one-lap run of the default problem on a  $320 \times 320$  grid:

```
| $ ./advect -da_refine 6 -ts_max_time 1.0 -ts_rk_type xx
```

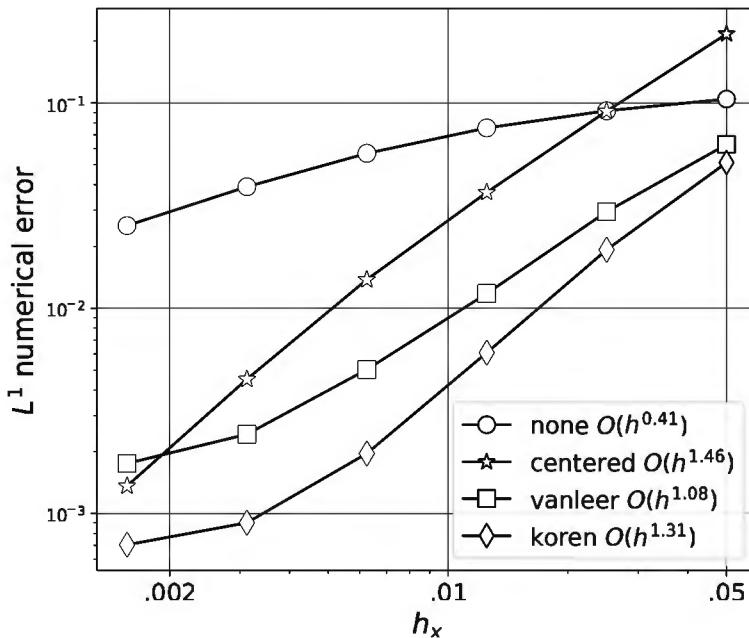
This generates Table 11.1. While the number of steps varies substantially, because different schemes have different regions of absolute stability (Chapter 5), adaptive time-stepping has enforced that the final errors are nearly the same. Furthermore, because lower-order schemes do more steps with less computation per step, and higher-order schemes vice versa, the run times are more similar than the number of steps. However, based on these results the 3bs RK solver, namely the PETSC default for `-ts_rk_type`, is the default solver in `advect.c`.

**Table 11.1.** Wall clock time, number of steps, and numerical error norms (11.29) for adaptive Runge-Kutta time-stepping types on a 2D advection problem.

| <code>-ts_rk_type</code> | <code>time (s)</code> | <code>steps</code> | $L^1$ error          | $L^2$ error          |
|--------------------------|-----------------------|--------------------|----------------------|----------------------|
| 2a                       | 17.4                  | 1920               | $1.6 \times 10^{-2}$ | $7.0 \times 10^{-2}$ |
| 3bs                      | 12.7                  | 616                | $1.7 \times 10^{-2}$ | $7.2 \times 10^{-2}$ |
| 5bs                      | 12.5                  | 322                | $1.9 \times 10^{-2}$ | $7.4 \times 10^{-2}$ |
| 5dp                      | 14.8                  | 450                | $1.6 \times 10^{-2}$ | $7.0 \times 10^{-2}$ |
| 5f                       | 15.7                  | 432                | $1.6 \times 10^{-2}$ | $7.0 \times 10^{-2}$ |

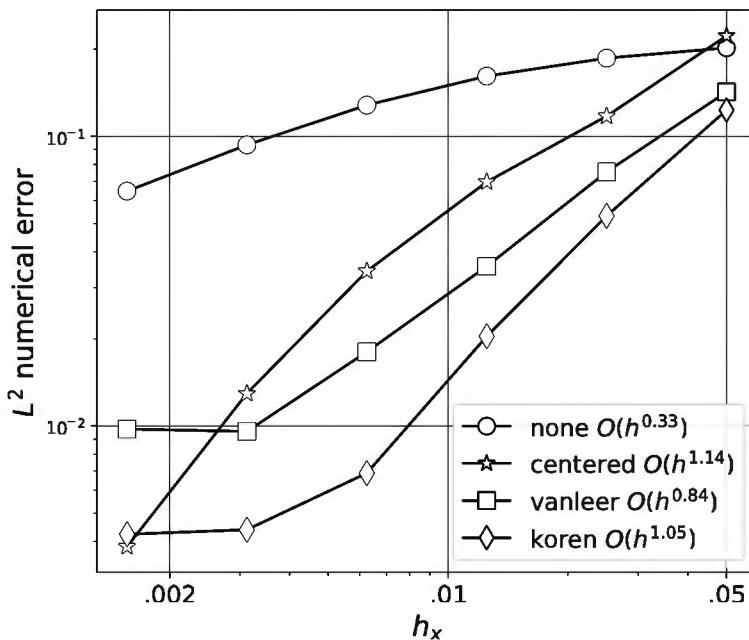


**Figure 11.10.** High-resolution flux-correction-limiting schemes yield smaller errors for the discontinuous *stump* initial condition. The convergence rate is poor because of the low regularity of the solution.



**Figure 11.11.** Replacing the *stump* initial condition with smooth improves convergence rates and redeems the centered flux scheme.

To examine convergence we use one lap (`-ts_max_time 1.0`) of the default problem and refine from 40 to 1280 grid points in each dimension, i.e., `-da_refine 3, ..., 8`. Figure 11.10 shows the  $L^1$  error norm results. The numerical error is substantially smaller for the high-resolution vanleer and koren schemes, confirming results in earlier figures. The convergence rate is also somewhat better, but the low regularity of the exact solution limits the rates to worse than  $O(h^1)$ . However, replacing *stump* with a smoother initial condition yields Figure 11.11.



**Figure 11.12.** With the smooth initial condition, the result using an  $L^2$  error norm (11.29) is similar to that from  $L^1$  (Figure 11.11).

Again the high-resolution methods are notably better than first-order upwinding, but now the centered flux scheme (11.21) is competitive. On sufficiently fine grids the centered beats the high-resolution schemes because at a smooth local extrema the limiter imposes (local) first-order upwinding while the centered fluxes remain second-order. Note that convergence with the  $C^0$  initial condition cone is not significantly worse than for smooth (Exercise 11.12).

From the same runs we can explore the role of the choice of error norm. For example, Figure 11.12 suggests that for smooth solutions the convergence pattern is similar between  $L^1$  and  $L^2$  norms, but compare Exercise 11.6.

Recall Figure 11.7. It reproduces Figure 20.5 from [103], showing the result after one lap of circular motion with wind  $\mathbf{a} = \langle y, -x \rangle$ , namely rigid motion counterclockwise around the origin. This can be computed in parallel, and viewed as a movie, by a run like

```
$ mpexec -n 4 ./advect -da_grid_x 80 -da_grid_y 80 -ts_max_time 6.283185 \
    -adv_problem rotation -adv_limiter XX -ts_monitor_solution draw
```

The  $XX = \text{vanleer}$  (not shown) and  $\text{koren}$  (Figure 11.7) results are indistinguishable at screen resolution. As the reader should confirm, both are much better than  $\text{none}$  or  $\text{centered}$ .

The major messages about numerical advection are now clear. High-resolution schemes are effective when compared by solution appearance (Figures 11.2 and 11.4) and in the sense of convergence rate (Figures 11.10–11.12). They are superior to the oversmoothed results from first-order upwinding in all cases, and to results from the centered scheme for discontinuous solutions. (The centered scheme generally shows “ringing,” the erroneous propagation of high frequencies throughout the domain.) The  $\text{koren}$  and  $\text{vanleer}$  high-resolution corrections are comparable, with  $\text{koren}$  slightly better. This situation reflects a numerical fact of life.

**Fact 18.** Achieving good-looking numerical advection results requires effort. *Numerical results for simple advection tend to reveal that high-frequency components are transported at the wrong rates. Nonlinear flux-limiters or slope-limiters can correct this, and are worth the effort.*

## Implicit time-stepping for advection

In Chapter 5 the evidence showed that implicit time-stepping schemes were more efficient than explicit schemes for diffusion equations. From the same empirical perspective, for advection problems we will find instead that explicit schemes are superior. Though this is no surprise—the standard literature [84, 103, 115] assumes it to be so—the reasons are worth examining. Furthermore, after this section we will solve equations which combine advection and diffusion, so we want our eyes open to the trade-offs. We start by making sure that implicit methods do indeed converge on advection problems, and then we measure their performance relative to explicit methods.

Note that our high-resolution spatial discretization schemes are nonlinear. We will use SNES solvers for the implicit step equations, but we need adequate smoothness of the scheme equations to allow the Newton iteration to converge. Thus the smoother `vanleer` limiter may have an advantage over the piecewise-linear `koren` limiter (Figure 11.6), but we will also test this belief.

The code `advect.c` only implements the Jacobian for limiter types `none` and `centered` (`FormRHSJacobianLocal()`; not shown). To check the correctness of such new code, one should both use `-snes_test_jacobian` (Chapter 4) and check that convergence occurs in one Newton step when the scheme and problem are linear. For example, consider a run using one step of the Crank-Nicolson (CN) method:

```
$ ./advect -da_refine 3 -ts_max_time 0.01 -ts_dt 0.01 -ts_type cn \
-snes_converged_reason -adv_limiter LIM -adv_jac_limiter JLIM
```

This implicit time-stepper creates the usual stack of algebraic equation solvers, so all SNES, KSP, and PC options become available. Using `LIM = JLIM = none` and adding `-snes_test_jacobian` yields a relative Jacobian difference norm of order  $O(10^{-10})$ , and we see convergence in one step. Repeating these tests with `LIM = JLIM = centered`, and adding `-ksp_rtol 1.0e-12` to make sure the linear equations are solved accurately, we get the same results. Thus we have strong evidence of Jacobian correctness.

Though we have not implemented an analytical Jacobian for `vanleer` (or `koren`), the Jacobians of the first-order upwind and/or centered formulas might be adequate replacements if we accept subquadratic convergence of the SNES iteration. Alternatively we may use a `none` or `centered` Jacobian to precondition a Jacobian-free Newton-Krylov step (Chapter 4) for one of the high-resolution methods. We will test these ideas empirically.

The following runs do a single time-step of CN, yielding Table 11.2.

```
$ ./advect -da_refine 4 -ts_dt 0.01 -ts_max_time 0.01 -ts_type cn \
-ksp_rtol 1.0e-12 -snes_converged_reason \
-adv_limiter LIM -adv_jac_limiter JLIM
```

The cases in the Table with one iteration use the correct Jacobian (on this linear problem). Otherwise, the `vanleer` limiter generates more rapid convergence than `koren` when an implemented Jacobian is used directly. This is expected; the smoothness of the limiter determines the

**Table 11.2.** Number of SNES iterations to compute one CN time step.

|                             |          | limiter used in residual |          |         |       |
|-----------------------------|----------|--------------------------|----------|---------|-------|
|                             |          | none                     | centered | vanleer | koren |
| limiter used in Jacobian    | none     | 1                        |          | 25      | 95    |
|                             | centered |                          | 1        | 27      | 86    |
| same +<br>-snes_mf_operator | none     |                          |          | 7       | 6     |
|                             | centered |                          |          | 7       | 9     |

rate of Newton convergence. The piecewise form of the `koren` limiter (Figure 11.6) is generally undesirable in a SNES residual function. However, the SNES iterations are greatly reduced for both limiters when `-snes_mf_operator` is used (i.e., preconditioned JFNK).

Thus our implicit, high-resolution solvers converge. However, relative to Runge-Kutta solvers, implicitness imposes a substantial performance cost with no improved accuracy outcome. To show this, consider runs of the form

```
$ ./advect -adv_initial smooth -ts_max_time 1.0 -da_refine 5 \
    -ts_type TYPE -adv_limiter LIM [+ other solver options]
```

with `TYPE` of `cn` or `rk`. The problem is one lap of `straight` on a  $160 \times 160$  grid.

Table 11.3 shows that the implicit run times, with any of the above limiter/Jacobian combinations, are one or two orders of magnitude greater than for the explicit (RK) rule. The numerical error magnitudes are essentially determined by the choice of limiter (i.e., by spatial truncation error) and not by the time-stepping procedure.

**Table 11.3.** Run times and  $L^1$  errors for implicit (`cn`) and explicit (`rk`) time-stepping on a pure advection problem.

| TS type | limiter  | solver options                         | time (s) | $L^1$ error          |
|---------|----------|----------------------------------------|----------|----------------------|
| cn      | none     | <code>-adv_jac_limiter none</code>     | 6.4      | $7.6 \times 10^{-2}$ |
|         | centered | <code>-adv_jac_limiter centered</code> | 12.8     | $5.0 \times 10^{-2}$ |
|         | vanleer  | <code>-adv_jac_limiter none</code>     | 67.6     | $1.0 \times 10^{-2}$ |
|         | vanleer  | <code>same + -snes_mf_operator</code>  | 54.8     | $1.0 \times 10^{-2}$ |
| rk      | none     |                                        | 0.5      | $7.6 \times 10^{-2}$ |
|         | centered |                                        | 0.5      | $3.7 \times 10^{-2}$ |
|         | vanleer  |                                        | 1.0      | $1.2 \times 10^{-2}$ |

While more testing could be done, for example with backward Euler or BDF implicit time-stepping schemes (Chapter 5), the evidence is already pretty clear. For pure advection problems, adaptive explicit time-stepping schemes are distinctly faster and just as accurate as adaptive implicit schemes. Explicit schemes are also simpler because no effort is expended on setting up a solver for algebraic equations, nor on the Jacobian matrix.

Regarding solver complexity, observe that in an advection problem each space-time grid location corresponds to a solution degree of freedom, essentially as worthwhile as any other. That is, when the time step has the value imposed by the CFL condition (11.14), an explicit solver already has “optimal” algorithmic complexity because it expends a fixed amount of work per degree of freedom.

Though the situation is now clear for time-dependent advection problems, the implicit/explicit tradeoff must be reconsidered when diffusion is included. In particular, pure advection problems often do not possess steady states. For example, all of the initial value problems solved by `advect.c` have no  $t \rightarrow \infty$  limit. However, even a small amount of diffusion allows a steady state if the source term and boundary terms are time independent.

## Steady-state advection-diffusion problems

So now we combine advection and diffusion into one time-independent equation. The problem is linear and elliptic, but the advection breaks the symmetry of the Laplacian operator and makes solver choice interesting. A key concern remains that the advection discretization be suitably nonoscillatory. In contrast to time-dependent pure advection problems, where explicit methods are well-suited, the construction of an optimal solver is nontrivial.

Suppose  $\Omega \subset \mathbb{R}^2$  is a bounded domain with well-behaved boundary. For a given diffusion constant  $\epsilon > 0$ , consider the following linear Dirichlet problem:

$$-\epsilon \nabla^2 u + \nabla \cdot (\mathbf{a}u) = g, \quad u|_{\partial\Omega} = b. \quad (11.30)$$

If the wind  $\mathbf{a}(\mathbf{x})$  is bounded and integrable, and if the source  $g(\mathbf{x})$  and boundary data  $b(\mathbf{x})$  are merely integrable, then a unique solution to the weak form of (11.30) exists in  $H^1(\Omega)$  [114]; it is well posed. However, we will assume further that  $\mathbf{a}$ ,  $g$ , and  $b$  are regular enough for strong-form solutions. Conditions for strong-form well-posedness are in [60] but see also Exercises 11.13 and 11.14.

A solution of (11.30) is the steady state of a time-dependent equation similar to the heat equation in Chapter 5, but with added advection, namely

$$u_t + \nabla \cdot (\mathbf{a}u) = \epsilon \nabla^2 u + g. \quad (11.31)$$

Said the other way, this equation adds diffusion to the advection problem (11.1). Observe that elliptic problems like (11.30) arise at each time step of an implicit method for (11.31).

Our examples will all have divergence-free wind ( $\nabla \cdot \mathbf{a} = 0$ ). Recall that the equation can be regarded either in flux-conservation form (11.30) or advective form  $-\epsilon \nabla^2 u + \mathbf{a} \cdot \nabla u = g$ . For a general wind one may convert between the two forms by adjusting  $g$ .

Advection-diffusion equation (11.30) is a singular perturbation [114] of the *reduced problem* where  $\epsilon \rightarrow 0$ , thus

$$\mathbf{a} \cdot \nabla u = g. \quad (11.32)$$

At least when  $\epsilon > 0$  is small, the reduced equation may tell us where the solution of (11.30) will have steep gradients, and thus where it will be hard to approximate. However, note that well-posed boundary conditions for (11.30) and (11.32) are generally different.

The reduced problem (11.32) can be understood and solved by calculus, that is, by characteristics. If  $u(\mathbf{x})$  solves (11.32) and  $\mathbf{x}(s)$  satisfies  $d\mathbf{x}/ds = \mathbf{a}(\mathbf{x})$  then  $U(s) = u(\mathbf{x}(s))$  solves  $dU/ds = g(\mathbf{x}(s))$ . Thus the value of  $u$  is determined along each characteristic  $\mathbf{x}(s)$  by a single value  $u(\mathbf{x}_0)$  and then by the solution of an ODE initial value problem. A Dirichlet boundary value problem for (11.32) is thus well posed if every such characteristic crosses  $\partial\Omega$  and if a boundary value is imposed exactly once per characteristic. Because of the sign of the diffusion term<sup>36</sup> in (11.30), one imposes an *inflow boundary condition* on (11.32), namely  $u|_{\Omega_-} = b$  where  $\Omega_-$  denotes the set of points  $\mathbf{x} \in \partial\Omega$  such that  $\mathbf{a}(\mathbf{x}) \cdot \mathbf{n} < 0$ , and where  $\mathbf{n}$  denotes the outward unit normal on  $\partial\Omega$  (as usual). In summary, the reduced problem is well posed, and its solution constructable by the method of characteristics, if we have only inflow (upstream) Dirichlet boundary conditions (Exercise 11.14).

Given a distance scale  $\ell$  and a scale  $a$  for the wind, the dimensionless *Peclet number*  $\mathcal{P} = a\ell/\epsilon$  [49] measures the relative roles of advection versus diffusion in PDE (11.30). We say the equation is *advection dominated* if  $\mathcal{P} \geq 1$ , and our primary goal will be to construct an effective solver in such cases. When  $\mathcal{P} \gg 1$  a solution of (11.30) will generally exhibit *boundary layers* [55] where it rapidly, though smoothly, changes to match Dirichlet boundary conditions at non-inflow locations. Where the solution of the reduced problem would propagate a discontinuity of the boundary data  $b$  into the interior, the advection-diffusion equation generates an *internal layer*. (Studying these layers would take us too far afield, but see [49, 55, 84, 114].) Our examples follow [49], to which the reader should refer for physical meaning and applicable FE methodology, but our goal is to build effective and even optimal solvers from the same FV point of view already applied to the advection equation.

<sup>36</sup>The reduced problem (11.32) treats the two directions along a characteristic symmetrically, but the time-dependent equation (11.31) with  $\epsilon > 0$  is only well posed forward in time.

Regarding the Peclet number, the distance scale that concerns us most is the grid spacing  $h$ . The upcoming discussion of discretization schemes will treat the *mesh Peclet number*

$$\mathcal{P}^h = \frac{ah}{\epsilon} \quad (11.33)$$

as determining whether a scheme is suitable for (11.30) on a particular grid. (Note cells are square with  $h_x = h_y = h$  in the computed examples.)

The earlier material regarding numerical solutions of pure advection equations, i.e., the concerns and schemes which went into `advect.c` above, should have convinced the reader that we know how to discretize advection. Now, supposing we have an adequate advection discretization, we want to know how to use it in an efficient, multigrid-based solver of steady-state equation (11.30).

Our code `both.c` (not shown) solves (11.30) by a FV method similar to the one in `advect.c`, but `both.c` calls SNES at the top level instead of TS. The method uses a grid of points  $(x_i, y_j)$  with spacing  $h_x, h_y$  as illustrated in Figures 11.1 and 11.9, but now with Dirichlet boundary conditions. Similarly to scheme (11.7) for equation (11.1), at a generic interior point the residual has the form

$$\begin{aligned} F_{ij} = & -\epsilon \frac{U_{i+1,j} - 2U_{ij} + U_{i-1,j}}{h_x^2} - \epsilon \frac{U_{i,j+1} - 2U_{ij} + U_{i,j-1}}{h_y^2} \\ & + \frac{\phi_{i+1/2,j}^x - \phi_{i-1/2,j}^x}{h_x} + \frac{\phi_{i,j+1/2}^y - \phi_{i,j-1/2}^y}{h_y} - g_{ij}, \end{aligned} \quad (11.34)$$

where  $U_{ij} \approx u(x_i, y_j)$ . The face-center fluxes are computed as in (11.23), allowing a flux-correction limiter  $\psi(\theta)$ :

$$\phi_{i+1/2,j}^x = \begin{cases} a [U_{ij} + \psi(\theta_{ij})(U_{i+1,j} - U_{ij})], & a \geq 0, \\ a [U_{i+1,j} + \psi(1/\theta_{i+1,j})(U_{ij} - U_{i+1,j})], & a < 0, \end{cases} \quad (11.35)$$

where  $a = a^x(x_{i+1/2,j})$  and  $\theta_i = (U_{ij} - U_{i-1,j})/(U_{i+1,j} - U_{ij})$ , with similar formulas for the other four faces of  $V_{ij}$ .

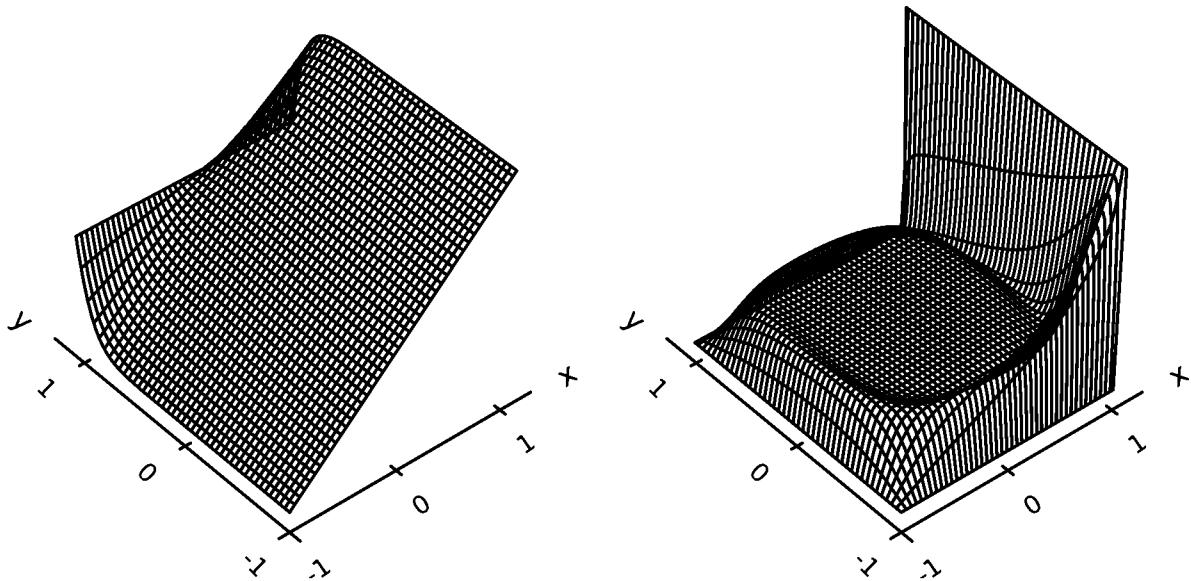
Code `both.c` solves the following three problems (option `-bth_problem`) for advection-diffusion equation (11.30):

**layer:** This problem has an *exponential boundary layer* of  $O(\epsilon)$  width [49, 55] at  $y = 1$  and an exact solution  $u(x, y) = x(1 - e^{(y-1)/\epsilon})/(1 - e^{-2/\epsilon})$ . [Data:  $\mathbf{a} = (0, 1)$ ,  $g = 0$ , and  $b$  is determined from the exact solution.]

**glaze:** This “double-glazing” problem is an advection-diffusion model which applies to the temperature of a convecting fluid in a cell with one hot wall [154]. Unlike **layer** where the wind blows only in the  $y$  direction, the wind here is “recirculating” and it blows in all directions. There are characteristic boundary layers [49] from the corners where the boundary values  $b(\mathbf{x})$  are discontinuous. [Data:  $\mathbf{a} = (2y(1 - x^2), -2x(1 - y^2))$ ,  $g = 0$ , and  $b = 1$  where  $x = 1$  (otherwise zero); see Exercise 11.19.]

**nowind:** The same Poisson problem as solved by `ch6/fish.c`, with exact solution  $u(x, y) = \epsilon x e^y$ . [Data:  $\mathbf{a} = (0, 0)$ ,  $g$  and  $b$  determined from  $u$ .]

Solutions to **layer** and **glaze** are shown in Figure 11.13; these problems are the same as Examples 6.1.1 and 6.1.4 from [49], respectively. The default value  $\epsilon = 1/200$  can be adjusted by option `-bth_eps`.



**Figure 11.13.** Results from both.c. Problem layer has an exact solution and a downstream boundary layer at  $y = 1$  (left;  $\epsilon = 1/10$  in (11.30)). Problem glaze shows the flow of heat for a circulating wind and a hot wall at  $x = 1$  (right;  $\epsilon = 1/200$ ; compare Figure 6.5 in [49]).

Regarding the limiter  $\psi$  in (11.35), set by `-bth_limiter`, we implement:  $\psi = 0$  (`none`; first-order upwinding),  $\psi = 1/2$  (`centered`), and van Leer (`vanleer`; (11.26)). For the van Leer limiter the star stencil has width two.

As usual we implement the residual in `FormFunctionLocal()` (not shown). However, based on prior positive experiences using `-snes_fd_color` on structured grids (Chapters 6–9), `both.c` does not include an analytic Jacobian. (Writing one is not hard, especially for the `none` and `centered` limiters; see Exercises 11.23 and 11.15.) We also implement a feature which will be explained below when testing multigrid, namely `-bth_none_on_peclet`. It causes the residual code to switch the limiter to `none` on any grid such that  $P^h > 1$ .

## Advection is not stagnation

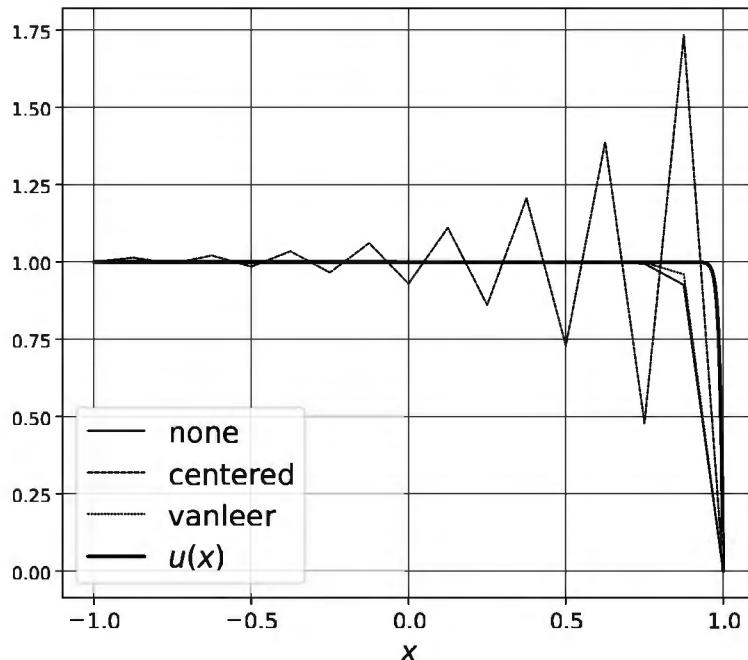
After compiling `both.c` in the usual way, the brave reader might set aside this narrative and just run it with the goal of finding optimal solver options on high-resolution grids. Here are some hints:

- (i) Problem (11.30) is linear so `-snes_type ksponly` works (except for the van Leer limiter, which is nonlinear).
- (ii) The system matrix is nonsymmetric so GMRES is a reasonable KSP type (but pay attention to restarts).
- (iii) Based on prior experience with advection, the `none` limiter should generate reasonable-looking results (but with low accuracy).

Returning to the narrative, consider the simpler 1D version of (11.30),

$$-\epsilon u'' + u' = 0, \quad u(-1) = 1, \quad u(1) = 0, \quad (11.36)$$

an ODE boundary value problem. It is straightforward to calculate the solution  $u(x) = (1 - e^{(x-1)/\epsilon})/(1 - e^{-2/\epsilon})$ . Note that problem layer in `both.c` is a 2D extension of this problem.



**Figure 11.14.** If the mesh Peclet number significantly exceeds one—here  $\mathcal{P}^h = 12.5$ —then the centered advection scheme is not acceptable, but first-order upwinding (none) and the vanleer limiter do okay.

Exercise 11.15 asks the reader to build a code `both1d.c` which solves (11.36). As in the 2D case, the diffusion is discretized by a centered FD scheme, while advection is by either first-order upwinding (`none`), `centered`, or `vanleer` limiter. A Jacobian is easy to implement, at least for the `none` and `centered` cases, and as with `both.c` we allow different limiters in the residual and Jacobian evaluations.

Assuming that this code has been written, we use it to demonstrate and visualize some important aspects. The result in an advection-dominated ( $\epsilon = 1/100$ ) case on a coarse ( $N = 17$ ) grid, with a direct solver (`-pc_type lu`), is shown in Figure 11.14. The centered limiter gives unacceptable oscillations, but the schemes which were stable for time-dependent advection also give reasonable solutions here. Observe that the  $L^\infty$  errors from the stable schemes are large where the exact solution changes rapidly within a single grid space. (This would also be true of an interpolant of the exact solution.) This observation, key to the analysis of an FE solution [49] of the problem, suggests measuring convergence in more forgiving norms (Exercise 11.6).

Under grid refinement we see that the `centered` limiter works well once the mesh Peclet number  $\mathcal{P}^h$  is small enough. Figure 11.15 shows the numerical errors for a broad range of grid sizes:

```
for LEV in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16; do
    ./both1d -snes_rtol 1.0e-11 -snes_converged_reason \
        -ksp_type preonly -pc_type lu -da_refine $LEV \
        -b1_limiter LIM -b1_jac_limiter JLIM
done
```

We have used `LIM = none,centered,vanleer` and `JLIM = none,centered,none`, respectively.

Figure 11.15, which also serves as a verification of `both1d.c`, shows that the `vanleer` limiter generates uniformly small errors across the full range of values of  $\mathcal{P}^h$ ; it combines the

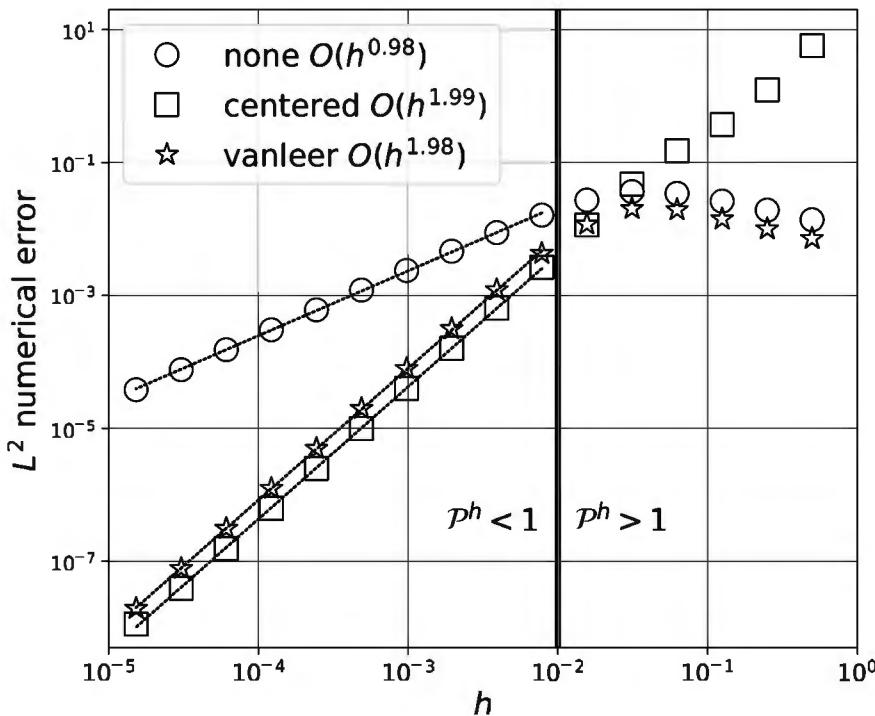


Figure 11.15. Numerical errors for both1d.c solving (11.36) with  $\epsilon = 1/100$ .

best of the linear flux schemes. However, being nonlinear, it is substantially more expensive because nontrivial Newton iterations are required (Exercise 11.17). These results also show that errors from first-order upwinding are not significantly larger than from the van Leer scheme on coarser grids where  $P^h > 1$ , while for  $P^h < 1$  the centered scheme does just as well as vanleer.

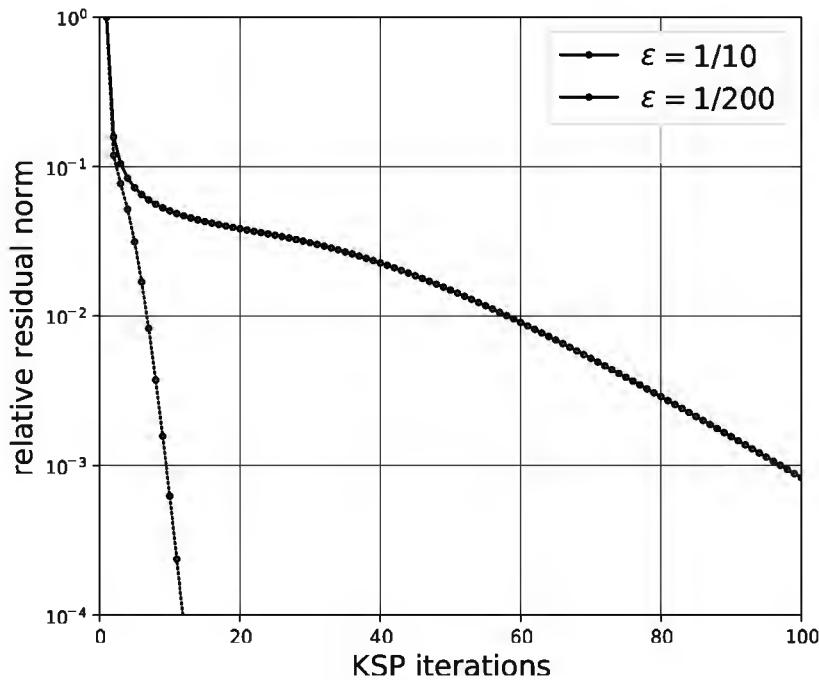
Thus, a natural strategy emerges which can be used in constructing a geometric multigrid (GMG) solver (Chapter 6). Namely, we apply the centered discretization on grids sufficiently fine so that  $P^h < 1$ , but revert to first-order upwinding when rediscretizing on coarser grids with  $P^h > 1$ . This strategy, which avoids adding a nonlinear limiter to a linear PDE, will yield optimal computational complexity in 2D cases solved by both.c, and it extends to 3D as well. (Note that a direct solver like LU, i.e., -ksp\_type preonly -pc\_type lu, is already optimal in the tridiagonal or pentadiagonal 1D problem solved by both1d.c.)

The application of multigrid to diffusion problems is motivated by the stagnation of the classical iterations on fine grids, which are slow to reduce low-frequency error components (Chapter 6 and [21]). However, adding advection does *not* make stagnation worse; the opposite is true. For example, suppose we choose  $\epsilon = 1/10, 1/200$  and a classical Gauss-Seidel (GS) iteration as a solver:

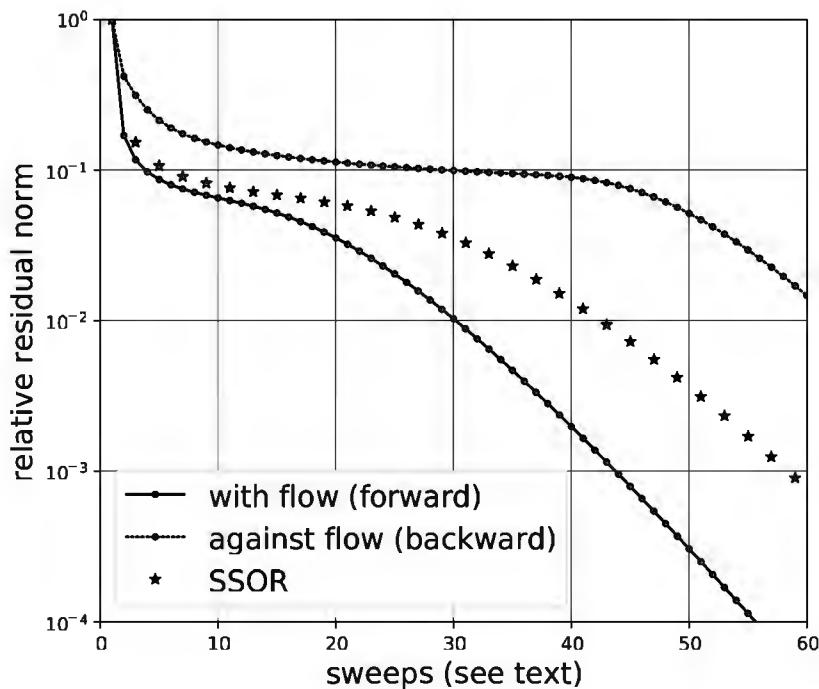
```
$ ./both1d -b1_eps EPS -da_refine 4 -snes_type ksponly \
-ksp_type richardson -pc_type sor -pc_sor_forward \
-ksp_monitor_true_residual
```

The result is shown in Figure 11.16; compare [49] Figure 7.2. Apparently having more advection is better; the  $\epsilon = 1/200$  result suggests no stagnation at all in reducing the residual norm.

This result is dependent on variable order, however, and an effective 2D or 3D iteration will need to follow the flow. To illustrate in 1D, suppose we fix  $\epsilon = 1/25$  in the above run and we try -pc\_sor\_forward, -pc\_sor\_backward, or no option (SSOR). This yields Figure 11.17;



**Figure 11.16.** GS iterations can be fast for advection-dominated problems.



**Figure 11.17.** Flow-following variable ordering is important for iterations on advection problems.

compare Figure 7.4 in [49]. Because the SSOR iteration does two sweeps per iteration (Chapter 6), we double its iteration count for the figure. The forward method causes the greatest residual reduction because the wind  $a = 1$  is in the same direction as the sequence in which forward updates the variables. The backward ordering is especially slow to get even the first order-of-magnitude reduction (Exercise 11.16), but note it does no harm—residual norms do not

grow though little progress is made. SSOR is slower in iterations than forward for the simple reason that it does both a forward and a backward sweep during each iteration.

In planning for a multigrid solver we may draw the conclusions that the advection operator must be discretized in a suitable way *and* that the smoother must follow the flow [49, 144, and references therein]. In fact, the following principle is expanded in the next section:

**Fact 19.** Advection is not stagnation. *Discretizations and iterations for advection-diffusion equations, either as solvers or multigrid smoothers, must be stable on coarse grids and follow the flow. Stagnation is only an issue when the mesh Peclet number is small and the choice of advection discretization unimportant.*

We do one more experiment with `both1d.c`, using  $\epsilon = 1/100$ , GMRES, GMG preconditioning, and a flow-following GS smoother:

```
$ ./both1d -b1_eps 0.01 -snes_type ksponly -da_refine $LEV \
    -ksp_converged_reason -pc_type mg -mg_levels_ksp_type richardson \
    -mg_levels_pc_type sor -mg_levels_pc_sor_forward
```

The result for very fine grids,  $LEV=15, \dots, 20$ , corresponding to  $N = 6.6 \times 10^5$  to  $2.1 \times 10^7$  degrees of freedom, is that six KSP iterations suffice in each case. The run time of this apparently optimal solver is within a factor of three of the  $O(N)$  LU direct solver on all grids. This is a very promising result for 2D or 3D problems because a GMRES+GMG strategy should be (roughly) dimension independent, while a direct-solver strategy falls apart with increasing dimension.

## Multigrid for advection-diffusion problems

We now compose geometric multigrid (GMG) solvers for 2D advection-diffusion equation (11.30). Based on three core choices,

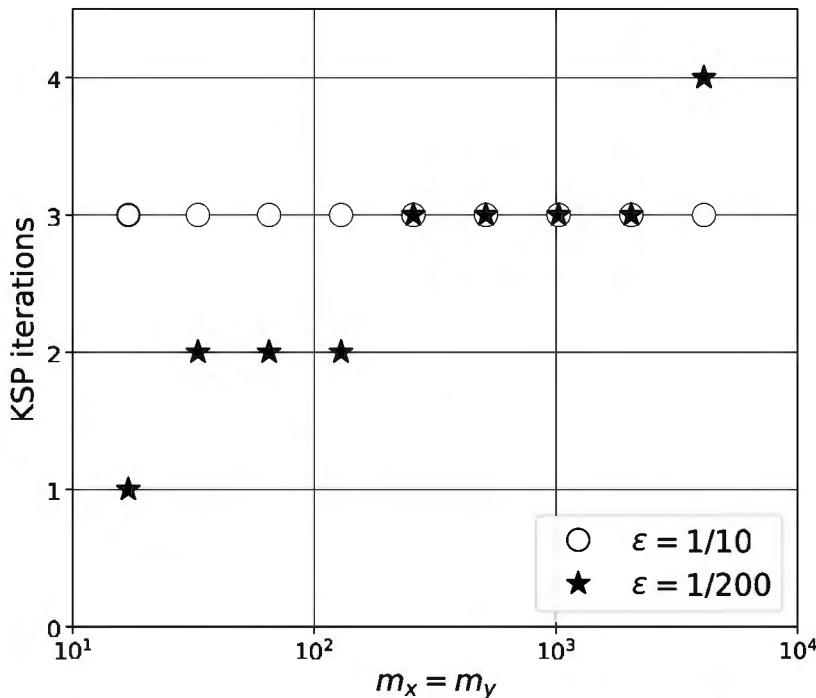
- (i) V-cycles,
- (ii) rediscretization on each grid level, using first-order upwinding if the mesh Peclet number exceeds one ( $\mathcal{P}^h > 1$ ), and
- (iii) smoothing operators which respect the flow [49],

these solvers show good performance across a range of resolution ( $h$ ), diffusion constant ( $\epsilon$ ), and processor count. Though particular runs involve further specific choices, advection-diffusion equation solvers with these basic features are often optimal and weak scalable (Chapter 8).

Choice (i) means simply using `-pc_type mg`, which defaults to V-cycles. Because each V-cycle makes significant progress, we use the default of one application per Krylov iteration (`-pc_mg_multiplicative_cycles 1`). Recall that W-cycles are unwise in parallel because of the communication costs imposed by coarse-grid visits (Chapter 7).

The easiest version of choice (ii) is to use the `none` limiter on all grid levels, the default in `both.c`. However, using the `centered` limiter when a finer grid satisfies  $\mathcal{P}^h < 1$  gives excellent results which converge at  $O(h^2)$ . Note that Galerkin coarse grid matrices are an alternative to rediscretization (Chapter 6), but naive application does not work here (Exercise 11.22). We do not pursue the necessary operator-dependent interpolation/restriction approach, but see [144].

For (iii), if the wind is in positive-coordinate directions only, as in the default problem `layer`, it suffices to use GS as the smoother (`-mg_levels_pc_type sor -mg_levels_pc_sor_forward`). For recirculating flows, however, we will compare SSOR and ILU smoothers and see that both are acceptable. Because the default Chebyshev iteration has trouble estimating the eigenvalue distribution for the highly nonsymmetric system matrix,



**Figure 11.18.** Number of GMG-preconditioned `bcgs` iterations using first-order upwinding and GS smoothing for problem layer in both .c.

at least in advection-dominated cases (below), we will only apply the smoother in the classical manner: `-mg_levels_ksp_type richardson`. (The reader can check that with `chebyshev` either the number of iterations is larger or the solver does not converge.) In parallel the smoother adds domain decomposition (Chapter 6).

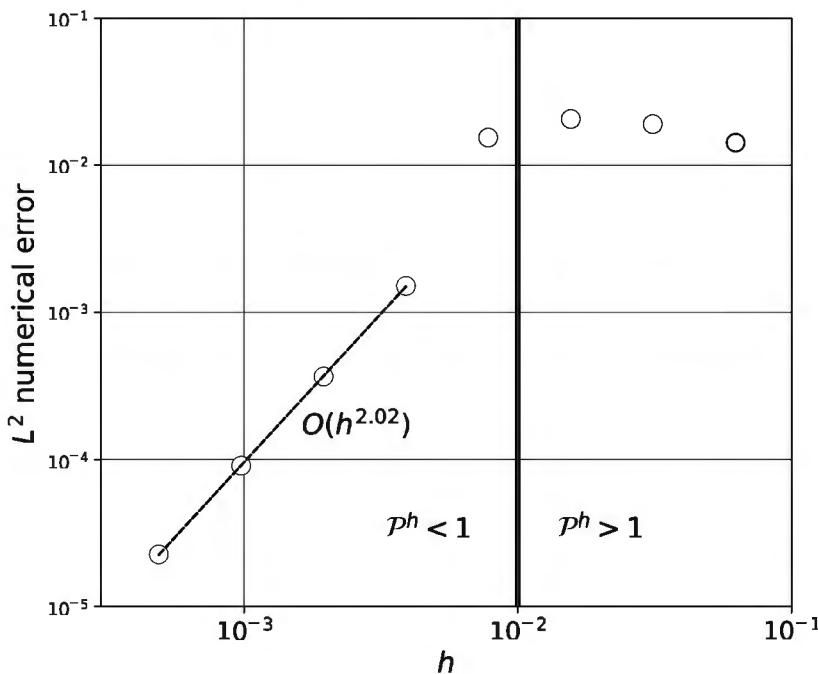
For nonsymmetric advection-diffusion problems the obvious KSP choice is GMRES. However, the fact that memory usage then grows as iterations proceed has motivated the construction of nonsymmetric Krylov iterations with fixed memory requirements. Among the best established are CGS (conjugate gradient squared [136]; `cgs`), BiCGSTAB (biconjugate gradient stabilized [146]; `bcgs`), and TFQMR (transpose-free quasiminimum residuals [56]; `tfqmr`). Any discussion of these methods would be tangential here, but see [66] for an overview. For now a quick test on a fine-grid  $\epsilon = 1/200$  glaze problem suggests that the run times of `gmres` with restart 30, `cgs`, `bcgs`, and `tfqmr` are all quite comparable, with a slight advantage to `bcgs`. Without any claim that it is superior across the category of advection-diffusion equations, we use `bcgs` in the following tests with, at least, assurance that the memory footprint is small.

Thus, as a first experiment using the default `layer` problem, we count iterations in serial solves which use first-order upwinding (on all levels), `bcgs`, and a preconditioner of one V-cycle down to a  $3 \times 3$  coarse grid, an `lu` direct solve on that coarse grid, and two sweeps of classical GS as pre/post-smoothers:

```
$ ./both -bth_eps EPS -snes_type ksponly -ksp_type bcgs -da_refine LEV \
-ksp_converged_reason -pc_type mg -mg_levels_ksp_type richardson \
-mg_levels_pc_type sor -mg_levels_pc_sor_forward
```

We do runs for  $\epsilon = 1/10, 1/200$  and  $LEV=3, \dots, 11$ , square grids  $m \times m$  for  $m = 17, 33, 65, \dots, 4097$ ; the finest has  $N > 10^7$  degrees of freedom and needs more than 16 GB memory. The result is shown in Figure 11.18.

Four KSP iterations suffice on all grids; fewer are needed when the advection is most dominant. The cost of a V-cycle is a small multiple of the cost of a smoother on the finest grid,



**Figure 11.19.** Numerical errors for problem 1 layer with a GMG solver which rediscretizes by first-order upwinding if  $\mathcal{P}^h > 1$ , but which uses the  $O(h^2)$  centered scheme otherwise.

and likewise for the cost of a few BCGS iterations, thus we have an optimal solver and even “textbook multigrid efficiency,” a result also supported by theory [49, 144].

Recall that option `-bth_none_on_peclet` adds a  $\mathcal{P}^h > 1$  test in the residual evaluation and switches the limiter to `none` if the test succeeds.<sup>37</sup> This stable rediscretization on coarse grids applies inside V-cycles. Fixing  $\epsilon = 1/200$ , and adding the following options to the above runs,

```
-bth_limiter centered -bth_none_on_peclet -ksp_rtol 1.0e-10
```

we get the expected  $O(h^2)$  convergence rate of the centered method on grids with  $\mathcal{P}^h < 1$  (Figure 11.19). Results from one of these runs generated the left part of Figure 11.13. Note that the vanleer limiter remains an option in `both.c`, an appropriate choice if the finest grid satisfies  $\mathcal{P}^h > 1$ . Concerning verification, `both.c` also includes a diffusion-only `nowind` problem which can be used to separately confirm the  $O(h^2)$  convergence of the diffusion discretization (not shown; Exercise 11.20).

Now we return to (iii) above, the principle that smoothers should respect the flow. Consider recirculating wind as in the `glaze` problem (Exercise 11.19). SSOR is a candidate smoother because each iteration traverses the grid in forward and backward orders, and thus at any location half the sweeps are (vaguely) in the wind direction. A significant improvement might be a 4-direction Gauss-Seidel iteration [49, 144], but this is not an existing PETSc PC. However, we are not limited to the classical iterations, and incomplete factorization smoothers are known to be effective on anisotropic diffusion and advection-diffusion problems [144, section 7.5, and references therein], so we propose to test these also.

To compare smoothers we do the following runs with the default `none` limiter:

```
$ ./both -bth_problem glaze -bth_eps 0.005 -snes_type ksponly -ksp_type bcgs \
-pc_type mg -mg_levels_ksp_type richardson -da_refine LEV
```

<sup>37</sup>We mention three details: 1. Computing  $\mathcal{P}^h$  for a given grid uses a representative wind scale  $a = \max |\mathbf{a}(\mathbf{x})|$ . 2. The default threshold 1.0 can be adjusted with `-bth_peclet_threshold`. 3. A warning appears if the test succeeds on all grids (and thus the user’s limiter was never applied).

We add one of three smoothers:

SSOR: `-mg_levels_pc_type sor`

ILU(0): `-mg_levels_pc_type ilu`

ILU(1): `-mg_levels_pc_type ilu -mg_levels_pc_factor_levels 1`

The last of these uses the sparsity pattern of the square of the system matrix to determine the allowed nonzero entries of the LU factors, and thus it allocates extra memory (in addition to the memory already used in out-of-place ILU). Note that, because first-order upwinding uses a star stencil, ILU(1) gives similar preconditioning to using ILU(0) and a width-two box stencil.

**Table 11.4.** Number of KSP iterations for different smoothers on the `glaze` problem with recirculating flow.

| smoother | $m_x = 65$ | $129$ | $257$ | $513$ | $1025$ | $2049$ |
|----------|------------|-------|-------|-------|--------|--------|
| SSOR     | 5          | 6     | 7     | 8     | 8      | 8      |
| ILU(0)   | 5          | 6     | 6     | 7     | 7      | 7      |
| ILU(1)   | 3          | 4     | 4     | 5     | 5      | 5      |

Table 11.4 shows the results for  $\text{LEV}=5, 6, 7, 8, 9, 10$  corresponding to  $m_x = m_y = 65, \dots, 2049$  grids. The ILU smoothers reduce the number of iterations, and an extra level of fill reduces further to about the level seen with the unidirectional wind and GS smoothing (Figure 11.18). On the other hand, the SSOR smoother actually uses the fewest flops and least time. We may expect any of these smoothers to generate an optimal solver, though the constant (in “ $O(N)$ ”) is larger than it was for unidirectional wind.

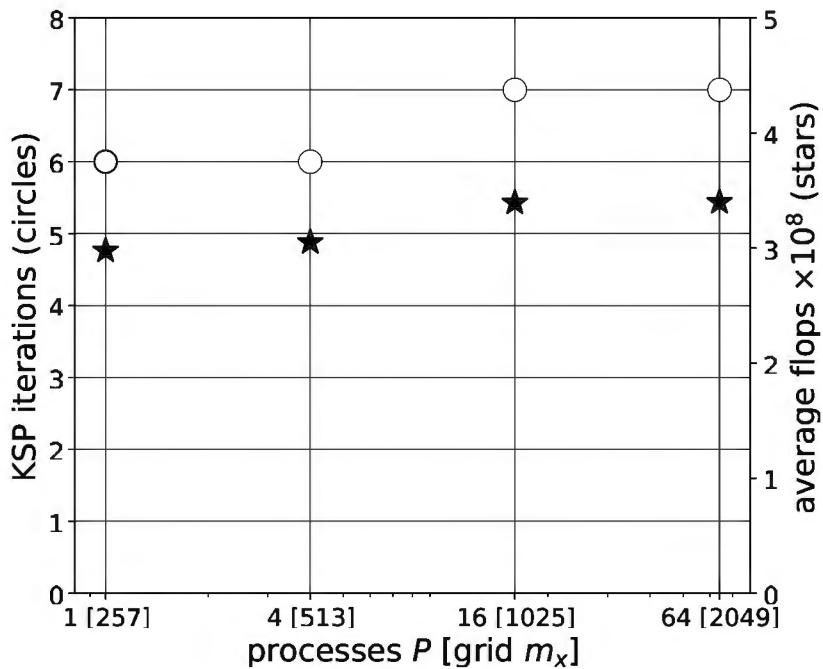
As our final example we solve the  $\epsilon = 1/100$  `glaze` problem in parallel using GMG preconditioning and a domain decomposition smoother. The  $O(h^2)$  centered discretization is used on fine grids, with none deeper in the V-cycles. The runs are of this form:

```
$ mpiexec -n P ./both -bth_eps 0.01 -bth_problem glaze \
    -da_grid_x 17 -da_grid_y 17 -da_refine LEV \
    -bth_limiter centered -bth_none_on_peclet -snes_type ksponly \
    -ksp_type bcgs -pc_type mg -mg_levels_ksp_type richardson \
    -mg_levels_pc_type asm -mg_levels_sub_pc_type sor
```

The smoother is ASM+SSOR and the  $17 \times 17$  coarse grid problem is solved with redundant LU. (The telescope PC would permit a much coarser coarse grid; see Chapter 7.)

Consider weak scaling with  $P = 1, 4, 16, 64$  processes and  $\text{LEV}=4, 5, 6, 7$ , respectively. Each process owns a  $257 \times 257$  subgrid, so when  $P = 64$  the grid has  $N = 2049^2 \approx 4 \times 10^6$  degrees of freedom. In the  $P = 1$  case this grid will barely resolve the boundary layers in the sense that  $\mathcal{P}^h = 0.78$  is close to one, while for larger  $P$  the solution features are well resolved.

To measure performance we add options `-ksp_converged_reason` and `-log_view` and read the “Flop:” line of the performance summary. The result (Figure 11.20) is that the number of KSP iterations only changes from 6 to 7 as  $P$  increases. The per-iteration average flops count on each process is essentially constant. Load balance is within 1.1% (not shown). This weak-scaling result confirms that our multigrid strategy for a 2D linear advection-diffusion problem scales as well as it did for the Poisson equation (Chapter 8).



**Figure 11.20.** In weak-scaling runs where each process owns a  $257 \times 257$  subgrid, KSP iterations (circles) and average flops per process (stars) are nearly constant from  $P = 1$  to  $P = 64$  processes.

## Exercises

- 11.1. Show that if  $u$  solves PDE (11.1) then it solves (11.2) for  $\tilde{g}$  constructed from  $g$ , and vice versa. Observe that if  $\nabla \cdot \mathbf{a} = 0$  then  $g = \tilde{g}$ .
  - 11.2. (This is a harmless reminder of solution by characteristics.)
    - (a) Consider the 1D version of (11.2) on  $\mathbb{R}^1$  with solution  $u(t, x)$ :
- $$u_t + a(x)u_x = g(x, u). \quad (11.37)$$
- If  $X(t) = X(t; \eta)$  is the solution of the scalar ODE  $\dot{X}(t) = a(X(t))$  with initial condition  $X(0) = \eta$ , i.e., a characteristic curve for (11.37), show that  $U(t) = u(t, X(t))$  solves  $\dot{U}(t) = g(X(t), U(t))$ .
- (b) Given an initial condition  $u(0, x) = q(x)$  for (11.37), describe how to use part (a) to solve the initial value problem.
  - (c) Solve the linear example  $u_t + (1 + x^2)u_x = 0$  with  $u(0, x) = e^{-x^2}$ .
- 11.3. (a) Show that combining (11.7) with (11.9) gives a second-order in space MOL scheme.
  - (b) After reducing to 1D, taking  $\mathbf{a} = \alpha$  constant, and setting  $g = 0$ , (11.1) becomes the simple advection equation  $u_t + \alpha u_x = 0$ . The system from part (a) becomes

$$U'_i = -\alpha \frac{U_{i+1} - U_{i-1}}{2h_x}. \quad (11.38)$$

With periodic boundary conditions on a grid of  $m$  points, equation (11.38) is  $U' = AU$  where  $A$  is an  $m \times m$  *circulant* matrix [143] with constant diagonals which wrap around ( $a_{ij} = a_{rs}$  if  $r - i = s - j \pmod{m}$ ). For such matrices an orthogonal basis of eigenvectors  $\{v_p(j) = \exp(i2\pi pj/m)\}_{p=0}^{m-1}$  is known, where  $i = \sqrt{-1}$ ,

independently of the matrix entries. Confirm this and then find  $\lambda_p$  so that  $Av_p = \lambda_p v_p$ . (A similar calculation gives the eigenvalues of matrix (11.15).)

- (c) Now consider applying the forward Euler scheme to (11.38), and recall the region of absolute stability from Figure 5.3. Conclude from the eigenvalues computed in (b) that this scheme, called *forward-time centered-space (FTCS)*, is not absolutely stable for any  $\Delta t > 0$ .
- (d) Applying the midpoint method for ODEs to (11.38) gives the classical *leapfrog* method

$$\frac{U_i^{l+1} - U_i^{l-1}}{2\Delta t} = -\alpha \frac{U_{i+1}^l - U_{i-1}^l}{2h_x}. \quad (11.39)$$

Show this is second order in time and space, and, by computing the closed absolute stability region  $\mathcal{S}$  for the midpoint method, that (11.39) is conditionally stable under CFL criterion (11.14).

- 11.4. By considering all cases in upwinding formula (11.10), and using forward Euler time-stepping, show that coefficients are positive if and only if CFL condition (11.14) holds.
- 11.5. Equation (11.11), namely  $u_t + \alpha u_x = 0$  for  $\alpha > 0$ , has characteristics which are straight lines, and the solution is constant along these lines. For a time semidiscretization with step  $\Delta t$ , the value of the solution  $(x, t_{n+1})$  can be found by following the characteristic back in time to  $(x - \alpha \Delta t, t_n)$ . On a grid  $\{x_i\}$  with spacing  $h$ , derive the first-order upwind (11.20), centered (11.21), and third-order upwind-biased (11.22) flux-discretization schemes by interpreting them as linear, quadratic, and cubic interpolation, respectively. In this case the CFL condition  $\Delta t |\alpha| \leq h$  avoids extrapolation. (For the third-order scheme the condition becomes  $\Delta t |\alpha| \leq 2h$ .)
- 11.6. Justify the following statement using the interpolation interpretation from the last exercise: *For (11.1) and a discontinuous initial condition we do not expect convergence in  $L^\infty$  norm for any of the schemes considered in this chapter.* Compare our choices of norms in the text of this chapter.
- 11.7. For 1D advection equation  $u_t + \alpha u_x = 0$ , show that the centered flux (11.9) yields an MOL scheme with local truncation error  $O(h^2)$ , but that the same scheme is  $O(h^3)$  for the modified equation

$$v_t + \alpha v_x + \frac{\alpha h^2}{6} v_{xxx} = 0. \quad (11.40)$$

Using Fourier series and periodic boundary conditions, show that solutions of (11.40) include dispersing waves. (Pure translation does not solve (11.40). *Dispersion relations* are discussed in [103].)

- 11.8. Show that if  $\psi(\theta) = 1/2$  then (11.23) reduces to (11.21), while if  $\psi(\theta) = 1/3 + (1/6)\theta$  then (11.23) reduces to (11.22).
- 11.9. Suppose  $v(x)$  is continuous on  $[0, L]$ , and extend it periodically to  $\mathbb{R}$ . Considering partitions  $\mathcal{P} = \{x_0 < x_1 < \dots < x_{n-1}\}$  of  $[0, L]$ , extended periodically, define the *total variation*

$$\text{TV}(v) = \sup_{\mathcal{P}} \sum_{i=0}^{n-1} |v(x_{i+1}) - v(x_i)|,$$

(Compare definition (11.27). Note  $\text{TV}(v) = \infty$  is possible even for continuous functions  $v(x)$ .) Suppose  $u(x, t)$  solves  $u_t + a(x)u_x = 0$  with periodic boundary conditions. By considering the flow generated by characteristic curves, show that  $\text{TV}(u(x, t)) = \text{TV}(u(x, 0))$ .

- 11.10. Confirm that equations (11.8) are true under the given hypotheses. Now use `TSMonitorSet()` to add a monitor of  $\sum_{ij} U_{ij}$  to `advec.c`. Confirm in serial and parallel that the value is constant in cases where  $g = 0$ .
- 11.11. By perhaps designing your own—see Figure 11.6 for ideas—or by using other flux-correction limiters from the literature, can you get smaller errors for `-adv_problem straight` than the results shown in Figures 11.10–11.12, i.e., generated with the Koren limiter? If so, are the numerical solutions visibly different than those shown in Figure 11.4? (*The author finds it hard to improve on Koren performance without greatly expanding the stencil. Compare [142].*)
- 11.12. Figures 11.11 and 11.12 show error norm results from `-adv_problem smooth`, an initial state which is in  $C^6$ . Confirm that initial condition `cone`, which is merely  $C^0$ , generates similar convergence rates over these grids.
- 11.13. Consider the following ODE boundary value problems, with 0, 1, or 2 boundary conditions as given, for  $u(x)$  on the interval  $x \in (0, 1)$ . Compute all solutions. Identify which problems are well-posed, i.e., for which there exists a unique solution.
- (i)  $u' = 0$ .
  - (ii)  $u' = 0, u(0) = a$ .
  - (iii)  $u' = 0, u(0) = a, u(1) = b$ .
  - (iv)  $-\epsilon u'' + u' = 0, u(0) = a, u(1) = b$ .
- 11.14. (*Continuing the theme of the above exercise.*) Let  $\mathbf{a} = (2, 1) \in \mathbb{R}^2$ . Consider the following PDE boundary value problems for solution  $u(x, y)$  on the square  $\Omega = (0, 1)^2 \subset \mathbb{R}^2$ . Find all solutions, if possible, and identify which problems are well posed. (State a space of functions on  $\Omega$  in which the solution is found.)
- (i)  $\mathbf{a} \cdot \nabla u = 0$ .
  - (ii)  $\mathbf{a} \cdot \nabla u = 0, u(0, y) = 1, u(x, 0) = 0$ .
  - (iii)  $\mathbf{a} \cdot \nabla u = 0, u(0, y) = 1, u(x, 0) = 0, u(1, y) = 0, u(x, 1) = 0$ .
  - (iv)  $-\epsilon \nabla^2 u + \mathbf{a} \cdot \nabla u = 0, u(0, y) = 1, u(x, 0) = 0, u(1, y) = 0, u(x, 1) = 0$ .
- 11.15. Construct `both1d.c` to solve 1D problem (11.36). Use FV/FD strategies analogous to (11.34). Implement the same limiters, and also an analytical Jacobian, though not necessarily for the van Leer limiter. Reproduce Figures 11.14, 11.15, and 11.16.
- 11.16. Do calculations by hand in the  $N = 5$  point case to do an iteration of Gauss-Seidel for the first-order upwind discretization of 1D problem (11.36). Specifically, start with a zero initial iterate and do one iteration each in the forward (flow-following) and backward (against-flow) variable orderings.
- 11.17. (*Do Exercise 11.15 first.*) Run the following visualization to see Newton iterations for the van Leer limiter:
- ```
$ ./both1d -b1_limiter vanleer -snes_grid_sequence 5 -snes_fd_color \
-snes_monitor_solution draw -ksp_view_mat draw -draw_pause 1
```
- Quantify the cost of the van Leer limiter compared to no limiter.
- 11.18. (*Do Exercise 11.15 first. This exercise suggests a minor variation on the van Leer scheme. The interested reader should see the literature on high-resolution schemes, including the rather different presentations in [84, 103, 114].*) A look at Figure 11.6 suggests that a smooth formula like the second-order van Leer scheme (11.26) can be built which comes closer to the third-order (i.e., for smooth solutions away from extrema) Koren scheme (11.25). In particular, note that  $\psi'(1) = 1/4$  for the van Leer scheme while  $\psi'(1) = 1/6$

for the third-order upwind-biased scheme (11.22) on which Koren is based. For  $c > 0$  define

$$\psi_a(\theta) = \frac{1}{2} \frac{\theta + c|\theta|}{1 + c|\theta|}.$$

Show that if  $c = 2$ , then  $\psi'_c(1) = 1/6$ , and that the scheme satisfies the symmetry property  $\psi_c(\theta)/\theta = \psi_c(1/\theta)$ . Implement  $\psi_2(\theta)$  as a limiter in `both1d.c`. By redoing Figure 11.15, show that the scheme has slightly improved order of convergence. Show, however, that the graph of  $\psi_2(\theta)$  does not quite stay in the Sweby region.

- 11.19. Show that the characteristics of the reduced problem (11.32) are level curves of a *stream function* [2]  $\Psi(\mathbf{x})$  satisfying

$$\mathbf{a}(\mathbf{x}) = \left( \frac{\partial \Psi}{\partial y}, -\frac{\partial \Psi}{\partial x} \right).$$

Compute a stream function for the wind in problem `glaze`. Thereby make a contour plot which shows the characteristic curves of the reduced problem. Then, perhaps also using a visualization like the following:

```
$ ./both -bth_problem glaze -da_refine 8 -pc_type mg \
-mg_levels_ksp_type richardson -snes_view_solution draw
```

and/or Figure 11.13, explain in a few sentences the balance of physical processes which generate this advection-diffusion solution.

- 11.20. Confirm that results for the  $\mathbf{a} = \mathbf{0}$  problem `nowind` in `both.c` are the same as from `ch6/fish.c` (which solves the same problem by default). Specifically, set  $\epsilon = 1$  and confirm that the system matrix is the same for corresponding grids (`-ksp_view_mat`), that the  $O(h^2)$  convergence rate is the same, and that the performance of CG+GMG solvers is the same. (A technicality: `-fsh_initial_gonboundary 0` is needed to get the same initial iterate as in `both.c`.)
- 11.21. Example 6.1.3 in [49] describes a boundary value problem for (11.30) with an *internal layer* of width  $O(\sqrt{\epsilon})$ ; the problem also exhibits exponential boundary layers. Add this problem to `both.c`; the data is  $\mathbf{a} = (-1/2, \sqrt{3}/2)$ ,  $g = 0$ , and  $b = 1$  where  $y > 2x - 1$  while otherwise zero. Test the multigrid solver techniques in the text which applied to problems `layer` and `glaze`. There should be few surprises.
- 11.22. The literature [144, section 7.7.5, and references therein] suggests that operator-dependent interpolation/restriction operators are needed to make a Galerkin coarse-grid strategy succeed. Show that naive use of Galerkin coarse-grid matrices does not work. For example, consider this visualization, a V-cycle run using first-order upwinding and GS smoothing on problem `layer`, with and without `-pc_mg_galerkin`:

```
$ ./both -da_refine X -snes_converged_reason -ksp_converged_reason \
-pc_type mg -mg_levels_ksp_type richardson \
-mg_levels_pc_type sor -mg_levels_pc_sor_forward \
-ksp_monitor_solution draw -draw_pause 1
```

On all but the coarsest grids, `-pc_mg_galerkin` causes a line-search failure.

- 11.23. Implement and test an analytical Jacobian for the `none` and `centered` limiters in `both.c`.
- 11.24. (*This exercise assumes you have already done Exercise 7.12.*) Write a code which solves the following advection-modified Liouville-Bratu equation with  $\epsilon > 0$ :

$$-\epsilon \nabla^2 u + \nabla \cdot (\mathbf{a}u) = \epsilon \lambda e^u, \quad u|_{\partial\Omega} = b, \quad (11.41)$$

First verify your discretizations (residual evaluation) using exact solutions:

- problem `layer` from `both.c` on  $\Omega = (-1, 1)^2$ , for a case where  $\lambda = 0$  but  $\mathbf{a}$  is nonzero, and
- Exercise 7.12 part (iv) on  $\Omega = (0, 1)^2$ , for a case when  $\epsilon = \lambda = 1$  and  $\mathbf{a} = 0$ .

(Note that the boundary conditions arise from the solutions.) Next, fix some nonzero wind field  $\mathbf{a}(\mathbf{x})$ , according to taste, and set zero Dirichlet boundary conditions on  $\Omega = (0, 1)^2$ . Solve (11.41) when  $\epsilon = \lambda = 1$  and demonstrate optimality of a multigrid solver in that case. Finally, recalling that the critical exponent is  $\lambda \approx 6.81$  when  $\mathbf{a} = 0$ , for your nonzero wind  $\mathbf{a}$  estimate the dependence of the critical exponent (Exercise 7.12 part (ii)) on the amount of diffusion:  $\lambda = \tilde{\lambda}(\epsilon)$ .

## Chapter 12

# Inequality constraints

The problem in this chapter is not literally a PDE because its solution is subject to inequality constraints. It must be posed in a global manner, by a *variational inequality* (weak) or *complementarity problem* (strong) formulation. These two new formulations are equivalent up to the same regularity concerns which distinguish weak and strong forms for PDEs (Chapters 9 and 10).

We solve the classical obstacle problem for a linearized elastic membrane [36, 51, 91, 130], a nonlinear problem because of the constraint. The solution does satisfy a PDE, the *interior condition*, in that portion of the domain where the inequality constraints are not active. The constraints are a source of boundary conditions which apply along a curve with *a priori* unknown location, the *free boundary*.

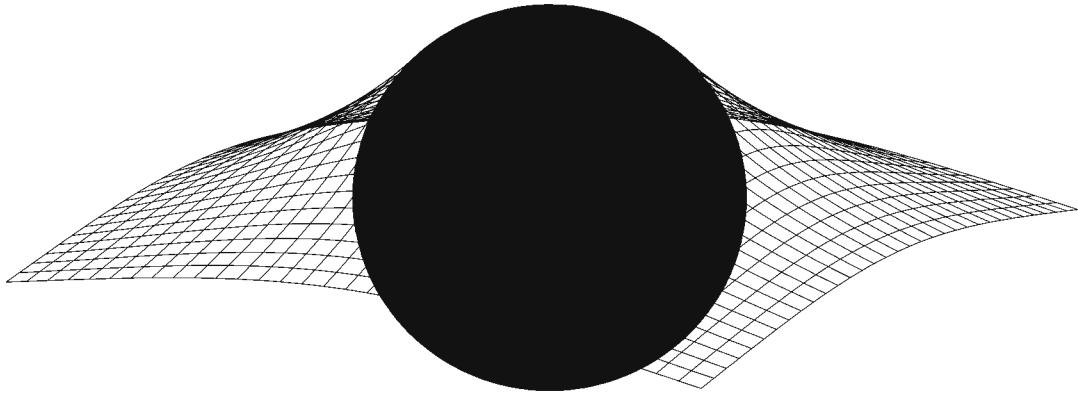
Because the interior condition is the linear Poisson equation, we can re-use the residual- and Jacobian-evaluation code from the FD Poisson equation solver of Chapter 6, but the problem is nonlinear so we use SNES. Only two “SNESVI” subtypes adapt the Newton method to allow inequality constraints, but otherwise we proceed as we have for nonlinear, elliptic PDE problems on a 2D structured DMAD grids, making solver choices like those for the minimal surface (Chapter 7) and  $p$ -Laplacian (Chapter 9) equations. Thus, after explaining SNESVI, we construct optimal solvers by applying grid-sequenced and multigrid-preconditioned (nonlinear full cycle) Newton-Krylov methods.

## The classical obstacle problem

Suppose that an elastic membrane at height  $z = u(x, y)$  is attached to a rigid wire frame at height  $z = g(x, y)$  along the boundary of a planar region  $\Omega$ , and suppose that the membrane is subject to a distributed load  $f(x, y)$ . Assume we also place a rigid, smooth obstacle, defined by a continuously differentiable function  $z = \psi(x, y)$ , underneath the membrane. For the problem to make sense, along the boundary the obstacle must be below the wire frame:  $\psi|_{\partial\Omega} \leq g$ . The classical obstacle problem asks, What is the minimum energy configuration of the membrane subject to the constraint that the membrane is on or above the obstacle, i.e.,  $u \geq \psi$ ?

Figure 12.1 shows an example where  $\Omega$  is a square,  $\psi$  is the upper unit hemisphere, and  $f = 0$ . (For aesthetic reasons we picture the obstacle as a closed sphere.) Also, the boundary condition  $g$  on  $\partial\Omega$  is, in this case, given by a formula for a radially symmetric exact solution (see below).

A more complete and physical membrane model would use the formulas of a minimal surface (Chapter 7; see also Exercise 12.11), but for this classical version of the problem [36, 51, 91, 130]



**Figure 12.1.** The solution  $u$  (wire frame) to an obstacle problem.

we accept the standard small-displacements linearization. The membrane therefore has a shape which is determined either by the Poisson equation or by contact with the obstacle:

$$\begin{aligned} -\nabla^2 u &= f && \text{where } u > \psi, \\ u &= \psi && \text{otherwise, and} \\ u &= g && \text{on } \partial\Omega. \end{aligned} \quad (12.1)$$

As the reader may already see, equations (12.1) are not an adequate way to pose our problem. In advance of solving the problem the *inactive set*

$$R_u = \{x, y \mid u(x, y) > \psi(x, y)\} \subseteq \Omega, \quad (12.2)$$

on which the Poisson equation applies, is unknown. (Regarding the terminology, the constraint  $u \geq \psi$  is not active, i.e., an equality, on  $R_u$ .) The complement  $A_u = \Omega \setminus R_u$  is called the *coincidence* [130] or *active set*. Finding the sets  $R_u$ ,  $A_u$ , and the free boundary

$$\Gamma_u = \partial R_u \cap \Omega, \quad (12.3)$$

on which a Dirichlet boundary condition  $u = \psi$  applies, are all part of the problem.

**Example.** Figure 12.2 shows these sets for the same case as in Figure 12.1. We use this case, on the square domain  $\Omega = (-2, 2)^2$ , both for illustration and code verification. The hemispherical obstacle has formula

$$\psi(r) = \begin{cases} \sqrt{1 - r^2}, & r \leq r_0, \\ \ell(r), & r > r_0, \end{cases}$$

where  $r = (x^2 + y^2)^{1/2}$ ,  $r_0 = 0.9$ , and  $\ell(r) = \psi(r_0) + \psi'(r_0)(r - r_0)$  is the unique linear function making  $\psi$  continuous and continuously differentiable. The exact solution  $u(x, y)$  satisfies Laplace's equation on  $R_u$ , namely the radial ODE

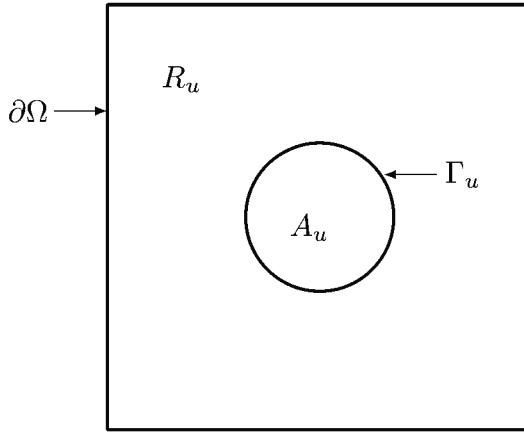
$$r u''(r) + u'(r) = 0. \quad (12.4)$$

We find the exact solution via free and fixed boundary conditions:

$$u(a) = \psi(a), \quad u'(a) = \psi'(a), \quad u(2) = 0; \quad (12.5)$$

the first two conditions apply at an unknown radius  $r = a$  with  $0 < a < 1$ . The solution to ODE (12.4) is  $u(r) = -A \log(r) + B$  on the interval  $a < r < 2$ . Conditions (12.5) can then be reduced to a root-finding problem for  $a$ ,

$$a^2(\log(2) - \log(a)) = 1 - a^2,$$



**Figure 12.2.** Sets for the solution to the obstacle problem in Figure 12.1.

with solution  $a = 0.697965148223374$ . It follows that  $A = a^2(1 - a^2)^{-1/2}$  and  $B = A \log(2)$ , giving an exact formula for  $u(r)$ .

The solution is now defined for all  $r \geq 0$ . Restricting this solution to  $\Omega$  supplies the boundary condition  $g(x, y) = u(r(x, y))$ , along the square  $\partial\Omega$ .

For any obstacle problem,  $u = \psi$  along the free boundary  $\Gamma_u$ . However, more is true because the membrane is also tangent to the obstacle. That is, along  $\Gamma_u$  two equations hold:  $u = \psi$  and  $\nabla u = \nabla \psi$ . Speaking heuristically, such simultaneous Dirichlet and Neumann conditions along  $\Gamma_u$  imply an overdetermined Poisson problem on  $R_u$ , but, because the location of  $\Gamma_u$  is also unknown, the problem can be well posed [130]. In fact, the following theory reformulates the problem as minimization in a closed, convex subset of a function space. After showing the well-posedness of this formulation, we can also show that a sufficiently-regular solution has a strong ‘‘complementarity’’ formulation.

Suppose  $\Omega \subset \mathbb{R}^d$  is a bounded region with Lipschitz boundary, and assume  $g$  is a continuous function defined on  $\partial\Omega$ . Recall from Chapters 9 and 10 that  $W^{1,2}(\Omega)$  is the space of functions  $v$  for which  $\int_{\Omega} |v|^2 < \infty$  and  $\int_{\Omega} |\nabla v|^2 < \infty$ . Suppose  $\psi \in W^{1,2}(\Omega)$  satisfies  $\psi \leq g$  along  $\partial\Omega$ . Let  $W_g^{1,2}(\Omega)$  be the affine subspace of functions with boundary value  $v = g$  (in a trace sense [51]). Define the nonempty *admissible set*

$$K_{\psi} = \{v \in W_g^{1,2}(\Omega) \mid v \geq \psi\}, \quad (12.6)$$

which is closed and convex (Exercise 12.1). Finally, supposing  $f \in L^2(\Omega)$ , define a quadratic functional over  $v \in W_g^{1,2}(\Omega)$ :

$$I[v] := \int_{\Omega} \frac{1}{2} |\nabla v|^2 - fv. \quad (12.7)$$

Functional  $I[v]$  is similar to the one in Chapter 9, namely for the  $p = 2$  Helmholtz problem. It is convex on  $K_{\psi}$ , so  $0 \leq \lambda \leq 1$  and  $v, w \in K_{\psi}$  imply  $I[\lambda v + (1 - \lambda)w] \leq \lambda I[v] + (1 - \lambda)I[w]$ . Furthermore it is strictly convex; if  $0 < \lambda < 1$  and  $v \neq w$  then the inequality is strict.

**Theorem 12.1.** [51, 91] *The following weak formulations of the obstacle problem are equivalent, and there exists a unique solution  $u \in K_{\psi}$ :*

$$I[u] \leq I[v] \quad \text{for all } v \in K_{\psi}, \quad (12.8)$$

or

$$\int_{\Omega} \nabla u \cdot \nabla(v - u) \geq \int_{\Omega} f(v - u) \quad \text{for all } v \in K_{\psi}. \quad (12.9)$$

Condition (12.8) is a constrained minimization, or calculus-of-variations [51], formulation while (12.9) is a *variational inequality* (VI). The equivalence of the two formulations follows by differentiating the functional  $I[v]$  and exploiting convexity, being careful to apply the functional only to elements of  $K_\psi$  (Exercise 12.2). Then, noting that the functional is *coercive* (Chapter 9), meaning that  $I[v]$  is large whenever  $v \in K_\psi$  has large  $W^{1,2}(\Omega)$  norm, general functional analysis arguments show that a solution to the minimization formulation exists (Exercise 12.1). The strict convexity of  $I[v]$  implies that the minimizer of  $I[v]$  over  $K_\psi$  is unique.

Even if equivalent problems (12.8) and (12.9) are well posed, do they carry the meaning we intended in problem (12.1)? While the space  $W^{1,2}(\Omega)$  includes functions for which classical second derivatives do not exist, strong statements like (12.1) require that the solution has second derivatives. In fact we can show that if we also assume that, for some  $p \geq 1$ ,  $u \in W^{2,p}(\Omega) \cap K_\psi$  solves the VI (12.9) then two strong statements hold, first the *interior condition* on  $R_u$ ,

$$-\nabla^2 u = f, \quad (12.10)$$

and second the following *complementarity problem* (CP) formulation *on all of*  $\Omega$ ,

$$\begin{aligned} -\nabla^2 u - f &\geq 0, \\ u - \psi &\geq 0, \\ (-\nabla^2 u - f)(u - \psi) &= 0. \end{aligned} \quad (12.11)$$

(In fact these hold only almost everywhere.) The third statement in (12.11), called “complementarity” in the optimization literature [118], says that either the Poisson equation or  $u = \psi$  holds in  $\Omega$ .

Both (12.10) and (12.11) follow from integration by parts on smooth test functions. First suppose  $\phi$  is a function with compact support in the open set  $R_u$ , and suppose it has continuous derivatives of all orders:  $\phi \in C_c^\infty(R_u)$ . Because  $u > \psi$  on  $R_u$ , and by continuity, there is  $\epsilon > 0$  so that  $v = u \pm \epsilon\phi > \psi$  on  $R_u$ . Note  $v \in K_\psi$  if we extend  $\phi$  by zero to all of  $\Omega$ . Then (12.9) implies  $\int_\Omega \nabla u \cdot \nabla(\pm\epsilon\phi) \geq \int_\Omega f(\pm\epsilon\phi)$  or, after integration by parts,

$$\pm\epsilon \int_\Omega (-\nabla^2 u - f)\phi \geq 0,$$

and thus  $\int_\Omega (-\nabla^2 u - f)\phi = 0$ . Because  $\phi$  was an arbitrary test function supported in  $R_u$ , (12.10) holds.

On the other hand, suppose the test function is nonnegative but has arbitrary compact support:  $\phi \in C_c^\infty(\Omega)$  and  $\phi \geq 0$ . Let  $v = u + \phi$  so again  $v \in K_\psi$ . Applying (12.9) and integration by parts,

$$\int_\Omega (-\nabla^2 u - f)\phi \geq 0.$$

This implies  $-\nabla^2 u - f \geq 0$  a.e. on  $\Omega$ , so we have shown the first part of (12.11). The second part simply says  $u \in K_\psi$ . Combining the interior condition (12.10) together with the fact  $u - \psi = 0$  on  $A_u$  shows the third part of (12.11).

Because the set on which it holds is unknown, equation (12.10) is just as inadequate as our first formulation (12.1). By contrast, though it requires more regularity than needed for the well-posedness of weak formulations (12.8) and (12.9), CP (12.11) is a reasonable way to state a free-boundary problem because each condition in (12.11) holds a.e. in  $\Omega$ .

Constrained minimization problems, VIs, and CPs are all closely related. In fact, CP (12.11) can be restated as the first-order *Karush-Kuhn-Tucker (KKT)* conditions [118] of minimization

problem (12.8) by using a slack variable or Lagrange multiplier  $\lambda \in L^2(\Omega)$ :

$$\begin{aligned} -\nabla^2 u - \lambda &= f, \\ u - \psi &\geq 0, \\ \lambda &\geq 0, \\ \lambda(u - \psi) &= 0. \end{aligned} \tag{12.12}$$

(This follows from (12.11) by the substitution  $\lambda = -\nabla^2 u - f$ .) Though strong form (12.12) has twice as many variables, it allows algorithms such as the primal-dual active set method [81].

On the other hand, the class of problems described by VIs and CPs is strictly larger than the class of constrained minimizations. For example, the VI

$$\int_{\Omega} F(u, |\nabla u|) \nabla u \cdot \nabla(v - u) \geq \int_{\Omega} f(v - u) \quad \text{for all } v \in K, \tag{12.13}$$

where  $K$  is some closed and convex subset of a Sobolev space, clearly generalizes (12.9). This VI is typically *not* the first-order condition of a constrained optimization. Indeed, in the context of constrained optimization a VI is a statement about the first derivative of the functional, so a VI derived from optimization must have the symmetry which arises from commutativity of second derivatives, namely a symmetric Hessian matrix. Generally (12.13) does not have this symmetry. The geometry of an ice sheet flow is an example of a VI/CP which is not the constrained minimization of a functional [30, 86].

We conclude the well-posedness theory with a few observations:

- (i) If  $u$  solves (12.8) and/or (12.9) then its second derivatives are generally not continuous on  $\Omega$  because the Laplacian  $\nabla^2 u$  will jump along the free boundary  $\Gamma_u$ . For example, in the problem shown in Figures 12.1 and 12.2 the active set  $A_u$  is part of the upper hemisphere of the obstacle, thus  $\nabla^2 u = \nabla^2 \psi$  is bounded above by a negative constant on  $A_u$ . Because  $u$  satisfies  $\nabla^2 u = 0$  on  $R_u$ , the Laplacian has a jump discontinuity along  $\Gamma_u$ .
- (ii) The data of the problem shown in Figures 12.1 and 12.2 could be made arbitrarily smooth without changing the solution behavior near the free boundary. Thus, in contrast to the Poisson equation, an obstacle problem can have arbitrarily smooth data and yet not have a solution in  $C^2$ .
- (iii) One defines a solution  $u$  of obstacle problem (12.9) to be *degenerate* if it just happens to solve the Poisson equation on part of its active set  $A_u$ . More precisely, the solution is *nondegenerate* if  $-\nabla^2 u - f > 0$  everywhere in the interior of  $A_u$ . If an obstacle problem is nondegenerate then the free boundary  $\Gamma_u$  is stable with respect to sufficiently small perturbations of the data [130, section 6:5]. When it comes to the numerical determination of the coincidence set we will assume nondegeneracy.

## Newton solvers for bound-constrained problems

Our code in this Chapter, `obstacle.c`, will solve the classical obstacle problem using either of two bound-constrained Newton solvers suited to VI/CP problems. Otherwise the code acts as expected; it sets up DMDA and SNES objects, calls `SNESSolve()`, and measures numerical error against an exact solution. There is substantial code reuse because the residual- and Jacobian-evaluation call-backs are unchanged from the Poisson solver in Chapter 6, so the new code is quite short. We only write new functions for the exact solution and for a SNES call-back supplying the inequality bounds.

The two SNESVI types are variants of line-search Newton methods:

- `vinewtonrsls`, which stands for “VI-adapted Newton solver with reduced-space line search,” abbreviated RS below, and
- `vinewtonssls`, a “semismooth” version, abbreviated SS.

These types allow *bound (box) constraints* given by vectors  $\mathbf{X}_l, \mathbf{X}_u \in \hat{\mathbb{R}}^N$ ,

$$\mathbf{X}_l \leq \mathbf{u} \leq \mathbf{X}_u, \quad (12.14)$$

where  $\hat{\mathbb{R}} = [-\infty, +\infty]$  denotes the extended real line.

Both SNESVI types were originally designed to solve finite-dimensional *nonlinear complementarity problems* (NCPs), namely

$$F(\mathbf{w}) \geq 0 \quad \text{and} \quad \mathbf{w} \geq 0 \quad \text{and} \quad \mathbf{w} F(\mathbf{w}) = 0, \quad (12.15)$$

where  $\mathbf{w} \in \mathbb{R}^N$  and  $F : \mathbb{R}^N \rightarrow \mathbb{R}^N$  (the *residual*), but they have been enhanced to allow two-sided bounds (12.14) [15]. Note that discretization is, of course, needed to put a continuum obstacle problem like (12.11) in finite-dimensional NCP form (12.15). Furthermore, a translation  $w = u - \psi$  and  $F(w) = -\nabla^2(w + \psi) - f$  would be needed to put (12.11) into literal form (12.15), but the SNESVI types use arbitrary bound constraints (12.14) directly, so this is unnecessary.

The last condition in (12.15), complementarity, can be interpreted either as a pointwise product  $\mathbf{w} F(\mathbf{w}) = 0$  or as an inner product  $\mathbf{w}^\top F(\mathbf{w}) = 0$ ; the forms are equivalent because of the first two conditions. The condition says that either an entry  $w_i$  is zero or the corresponding residual entry  $F_i(\mathbf{w})$  is zero. The problem is nondegenerate if, once the problem is solved, for each  $i$  either  $w_i > 0$  or  $F_i(\mathbf{w}) > 0$ .

We supply exactly the same residual which was used in Chapter 6 for the Poisson equation, namely the FD discretization of  $F(u) = -\nabla^2 u - f$ , and its Jacobian. There is, however, one important detail regarding the user-supplied functions, namely that for SNESVI the Jacobian must be *positive-definite*. That is, the sign of the residual is important; the residual written for Chapter 6 already has the correct sign.

We set the SNES type and a new kind of call-back:

```
SNESSetType(snes, SNESVINEWTONRSL);
SNESVISetComputeVariableBounds(snes, &FormBounds);
```

Function `FormBounds()` is shown below in Code 12.1. In our problem there is no upper bound so  $\mathbf{X}_u = +\infty = \text{PETSC\_INFINITY}$ ; compare Exercise 12.10.

```
// for call-back: tell SNESVI we want psi <= u < +infinity
PetscErrorCode FormBounds(SNES snes, Vec XI, Vec Xu) {
    DM da;
    DMDALocalInfo info;
    PetscInt i, j;
    PetscReal **aXI, dx, dy, x, y;
    SNESGetDM(snes, &da);
    DMDAGetLocalInfo(da, &info);
    dx = 4.0 / (PetscReal)(info.mx-1);
    dy = 4.0 / (PetscReal)(info.my-1);
    DMDAVecGetArray(da, XI, &aXI);
    for (j=info.ys; j<info.ys+info.ym; j++) {
        y = -2.0 + j * dy;
        for (i=info.xs; i<info.xs+info.xm; i++) {
            x = -2.0 + i * dx;
            aXI[i][j] = x * y;
```

```

    aXI[j][i] = psi(x,y);
}
DMDAVecRestoreArray(da, XI, &aXI);
VecSet(Xu,PETSC_INFINITY);
return 0;
}

```

**Code 12.1.** *c/ch12/obstacle.c, part I.* A call-back gives the VI/CP bounds.

We now sketch how the RS and SS algorithms work. Both are documented by [15], as solvers for problem (12.15), and they are similar in outline. Each one uses the current iterate to define an active set and it constructs a linear system based on this set and the supplied functions (call-backs). It then solves a linear (Newton) system to calculate a search direction, and finally performs a constrained line search to compute a new iterate which sufficiently decreases a merit function. However, in other details the algorithms differ:

RS. For an iterate  $\mathbf{w}^k \in \mathbb{R}^N$  the disjoint active and inactive sets in this algorithm are defined as

$$\begin{aligned}\mathcal{A}(\mathbf{w}^k) &= \{i \in \{1, \dots, N\} \mid w_i^k = 0 \text{ and } F_i(\mathbf{w}^k) > 0\}, \\ \mathcal{I}(\mathbf{w}^k) &= \{i \in \{1, \dots, N\} \mid w_i^k > 0 \text{ or } F_i(\mathbf{w}^k) \leq 0\}.\end{aligned}\quad (12.16)$$

The set  $\mathcal{A}(\mathbf{w})$  identifies the variables where the lower bound is active *and* the function value is positive (thus ignorable). Said the other way, an index  $i$  is in  $\mathcal{I}(\mathbf{w})$  if we should try to adjust variable  $w_i$  to either help solve that equation (make  $F_i(\mathbf{w})$  zero) or activate that constraint (make  $w_i$  zero). These sets are implemented using an index set (IS) type.

Prior to convergence there may be  $i \in \mathcal{I}(\mathbf{w}^k)$  such that  $w_i^k > 0$  and  $F_i(\mathbf{w}^k) < 0$ , violating complementarity (12.15) for that index. However, once the iteration converges to limit  $\hat{\mathbf{w}}$ , then, within a tolerance,  $F_i(\hat{\mathbf{w}}) = 0$  for each  $i \in \mathcal{I}(\hat{\mathbf{w}})$ , and thus the intended equations  $F = 0$  have been solved on the inactive set.

At each iteration a search direction is calculated by (approximately) solving a modified Newton step linear system. Only the inactive variables  $\mathcal{I}^k = \mathcal{I}(\mathbf{w}^k)$  are involved:

$$J(\mathbf{w}^k)_{\mathcal{I}^k, \mathcal{I}^k} \mathbf{d}_{\mathcal{I}^k}^k = -F(\mathbf{w}^k)_{\mathcal{I}^k}. \quad (12.17)$$

Here  $J(\mathbf{w})$  denotes the user-supplied, or finite differenced, Jacobian. The subscripts indicate that only rows and columns corresponding to inactive degrees of freedom are included. Once (12.17) is solved then the method also sets  $\mathbf{d}_i^k = 0$  for all  $i \in \mathcal{A}(\mathbf{w}^k)$ , thus defining a search direction  $\mathbf{d}^k \in \mathbb{R}^N$ .

Let  $K = \{\mathbf{w} \geq 0\}$  be the admissible set and assume  $\mathbf{w}^k \in K$ . The search direction  $\mathbf{d}^k$  may lead outside the admissible set, i.e.  $\mathbf{w}^k + \beta \mathbf{d}^k \notin K$  may occur for some  $\beta > 0$ . However, in RS the line search (Chapter 4) only evaluates the residual and Jacobian on admissible vectors. The simplest projection is used to achieve this, namely  $\pi : \mathbb{R}^N \rightarrow K$  defined by  $\pi[\mathbf{w}]_i = w_i$  if  $w_i > 0$  and  $\pi[\mathbf{w}]_i = 0$  if  $w_i \leq 0$ . Also, because an NCP solution  $\hat{\mathbf{w}}$  may yield positive residual values  $F_i(\hat{\mathbf{w}})$  for  $i \in \mathcal{A}(\hat{\mathbf{w}})$ , RS redefines the residual in the line search as follows:

$$\tilde{F}_i(\mathbf{w}) = \begin{cases} F_i(\mathbf{w}) & \text{if } w_i > 0, \\ \min\{F_i(\mathbf{w}), 0\} & \text{if } w_i \leq 0. \end{cases} \quad (12.18)$$

The line search then seeks a new iterate  $\mathbf{w}^{k+1} = \pi[\mathbf{w}^k + \beta \mathbf{d}^k]$ ,  $\beta > 0$  satisfying a

sufficient-decrease condition using the modified residual (12.18) and a 2-norm merit function, namely

$$\|\tilde{F}(\pi[\mathbf{w}^k + \beta\mathbf{d}^k])\|_2 \leq (1 - \sigma\beta) \|\tilde{F}(\mathbf{w}^k)\|_2, \quad (12.19)$$

where  $\sigma = 10^{-4}$  [15].

**SS.** The formulas of this method, which in contrast to RS may ask the user's code to evaluate *nonadmissible* iterates, are more complicated. We only give a brief summary.

A function  $\varphi : \mathbb{R}^2 \rightarrow \mathbb{R}$  is an *NCP function* when  $\varphi(a, b) = 0$  if and only if  $a \geq 0, b \geq 0$ , and  $ab = 0$ . The *Fischer-Burmeister* (FB) function,

$$\varphi_{FB}(a, b) = a + b - \sqrt{a^2 + b^2}, \quad (12.20)$$

is such an NCP function (Exercise 12.5), and SS redefines the residual  $\hat{F} : \mathbb{R}^N \rightarrow \mathbb{R}^N$  using it:

$$\hat{F}_i(\mathbf{w}) = \varphi_{FB}(w_i, F_i(\mathbf{w})). \quad (12.21)$$

The main idea is that  $\hat{F}(\mathbf{w}) = 0$  if and only if  $\mathbf{w}$  solves (12.15).

In rough outline the SS method solves the NCP by applying an unconstrained Newton method to the equation  $\hat{F}(\mathbf{w}) = 0$ . However, because an NCP function may not be smooth (continuously differentiable)—indeed  $\varphi_{FB}$  is not—the method cannot simply compute the Jacobian derivative of  $\hat{F}$ . Instead it computes a “subdifferential”  $H^k(\mathbf{w}) \in \mathbb{R}^{N \times N}$  [15]. This matrix is constructed using certain scale-and-translate formulas applied to the user-supplied Jacobian, based upon a different definition of the active and inactive sets compared to (12.16). The new formulas preserve the symmetry and positive-definiteness of the user-supplied Jacobian (if it is SPD). Then the method solves the linear (Newton) system in all variables,

$$H^k(\mathbf{w}^k) \mathbf{d}^k = -\hat{F}(\mathbf{w}^k) \quad (12.22)$$

to compute a search direction  $\mathbf{d}^k \in \mathbb{R}^N$ . The line-search sufficient-descent criterion uses  $\frac{1}{2}\|\hat{F}(\mathbf{w})\|_2^2$  as a merit function. Projection is *not* applied before evaluating  $\hat{F}$  and thus the user's residual code must handle vectors outside the admissible set. For remaining details see [15].

## Newton-multigrid and grid sequencing

We now solve the obstacle problem using constraint-adapted, preconditioned Newton-Krylov methods. Of the two SNESVI types we choose RS as the default (for reasons given below). The linear systems (12.17) and (12.22) are SPD in this case so the CG iteration is set as the default KSP type.

Recall that optimal  $O(N)$  performance can only occur when the number of SNES and KSP iterations is independent of grid resolution. This, of course, depends on high-quality preconditioning, and only multigrid approaches exhibit the necessary spectral equivalence for elliptic PDEs (Chapters 6–11). We try both DMDA-based geometric (GMG) and algebraic (AMG; Chapter 10) multigrid preconditioning.

After building the code `obstacle.c` in the usual way (`cd ch12/ && make obstacle`), consider runs of the form

```
| $ ./obstacle -snes_type VI PC -da_refine LEV
```

for VI equal to `vinewtonrsls` or `vinewtonssls`. After a bit of experimentation we see that the following solvers converge at levels  $LEV = 3, 4, 5, 6, 7$ :

- GMG:  $\text{PC} = -\text{pc\_type mg -mg\_levels\_ksp\_max\_it } 3$
- AMG:  $\text{PC} = -\text{pc\_type gamg -pc\_gamg\_type classical}$

Regarding these choices, note the SS+GMG run with  $\text{LEV} = 6$  diverges when using two smoother sweeps, but all levels converge with the stronger smoother. Also, classical AMG reduces the number of KSP iterations compared to smoothed aggregation (Chapter 10) in all of these cases.

Table 12.1 shows the result. The major point is that the SNES iterations increase substantially with grid resolution. We also observe that the number of KSP iterations is generally steady over grid resolutions except with the SS+GMG combination.

**Table 12.1.** Number of SNES and KSP iterations to solve the obstacle problem using the two SNESVI types (RS,SS) and two types of multigrid preconditioning. KSP iterations are for the last Newton step.

grid	RS			SS		
	SNES	GMG KSP	AMG KSP	SNES	GMG KSP	AMG KSP
$17 \times 17$	3	4	4	7	5	4
$33 \times 33$	6	3	4	11	7	4
$65 \times 65$	7	4	4	11	11	4
$129 \times 129$	12	4	4	16	13	4
$257 \times 257$	21	4	5	25	23	5

Why is the SNES count is increasing? The essential reason is that quadratic convergence of the Newton iteration is blocked until the (discrete) active and inactive sets stabilize. That is, the SNES first has to find the free boundary, and only then can the interior condition, the Poisson equation, be precisely solved. On fine grids almost all the work is in finding the active/inactive sets.

To see this concretely and visually, first note that `obstacle.c` sets  $\mathbf{w}^0$  to zero for simplicity. As the reader can check by visualization (below), for both SNESVI types the second SNES iterate  $\mathbf{w}^1$  is admissible and nearly equal to  $\max\{0, \psi\}$ , so  $\mathbf{w}^1$  has a too-large active set. The iteration must decrease the active set to its final configuration before it enters the domain of quadratic convergence. With the default RS type we can see the behavior using `-snes_vi_monitor`:

```
$ ./obstacle -pc_type mg -da_refine 5 -snes_vi_monitor
0 SNES VI Function norm 8.02729 Active lower constraints 729/885 ...
1 SNES VI Function norm 1.4556 Active lower constraints 629/729 ...
2 SNES VI Function norm 0.463479 Active lower constraints 553/629 ...
...
6 SNES VI Function norm 2.44749e-08 Active lower constraints 421/421 ...
done on 65 x 65 grid ... CONVERGED_FNORM_RELATIVE, SNES iter = 6, ...
errors: av |u-uexact| = 9.818e-05, |u-uexact|_inf = 5.991e-04, ...
```

The pair of numbers following “Active lower constraints” is in the form *active/at-bound*. The *active* variables satisfy both  $w_i^k = \psi_i$  (actually the condition is  $w_i^k \leq \psi_i + \delta$  for a tolerance  $\delta > 0$ ) and  $F_i(\mathbf{w}^k) > 0$ . That is, these variables are in the active set as defined by (12.16). The larger *at-bound* number gives the variables satisfying only the first condition. In this run the *at-bound* variables  $w_i$  which have a negative  $F$  value are moving away from the constraint, so the estimated active set gets smaller.

In the above run the default RS type monotonically decreases the number of active variables. While SS also starts with an active set which is too large, the number of active variables does not decrease monotonically. At convergence, however, the two types agree on the number of active

variables. On this problem the interior condition is the linear Poisson equation, but note the SS type introduces a nonlinearity into the residual (12.21). It then needs a few more iterations to converge once the active/inactive set identification is completed.

The following X-windows-based visualizations help to understand the iterations of SNESVI solvers:

- `-snes_monitor_solution draw` shows Newton iterates  $\mathbf{w}^k$
- `-snes_monitor_solution_update draw` shows Newton steps  $\mathbf{d}^k$
- `-snes_vi_monitor_residual` shows residual values  $\tilde{F}_i(\mathbf{w}^k)$ ; see (12.18)

The last of these viewers displays zero residual at each point in  $\mathcal{A}(\mathbf{w}^k)$ —see (12.16)—so the inactive variables, corresponding to nonzero residual values where more work remains to be done, are visible (Exercise 12.6).

Many elementary numerical methods for obstacle problems can only move the approximated free boundary by one grid cell per iteration [65], and this is true of the above runs. This fact implies that, under grid refinement, the number of Newton iterations increases with the number of grid points in each direction, exactly as seen in Table 12.1. The number of SNES iterations is roughly proportional to the number of grid spaces between the free-boundary positions for the initial iterate and the converged solution. We must fix this behavior if we want to build an optimal solver.

So, with these preliminary results and visualizations in mind, how do we reduce the number of SNES iterations on fine grids? One can show that on a coarse grid almost any admissible iterate is comparably effective (Exercise 12.7), and so for finer grids we propose to exploit grid-sequencing (Chapter 7). Combining this with multigrid preconditioning, the resulting nonlinear full multigrid cycles will transcend the limitations on moving the free-boundary.

To do so we simply replace `-da_refine` with `-snes_grid_sequence` in the above runs. The result in Table 12.2 shows nongrowing, or slowly growing, SNES iteration counts on the finest grid. The SS+GMG combination again shows growing KSP iterations; this solver combination is not recommended.

**Table 12.2.** Using `-snes_grid_sequence` to redo the runs in Table 12.1 produces nongrowing SNES iterations for RS, and slowly increasing for SS. SNES iterations are on the last (finest) grid and KSP iterations on the last Newton step.

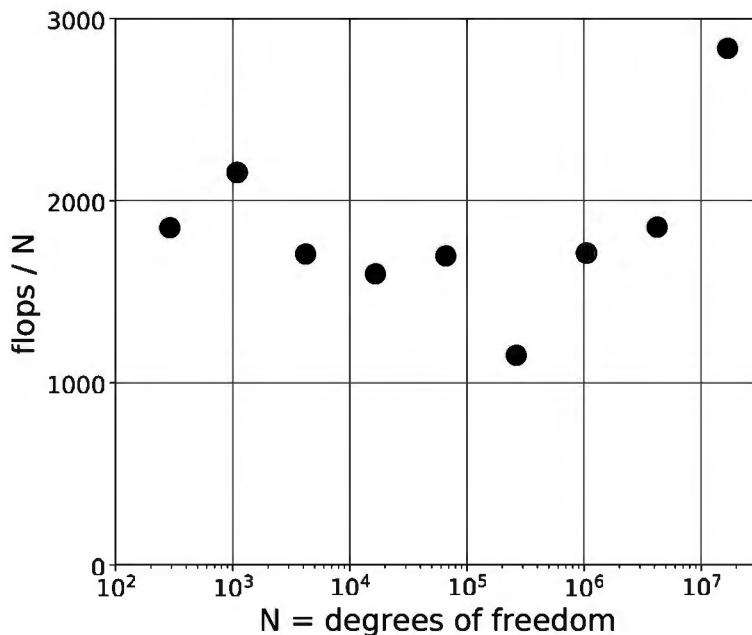
grid	RS			SS		
	SNES	GMG KSP	AMG KSP	SNES	GMG KSP	AMG KSP
$17 \times 17$	3	4	4	5	5	4
$33 \times 33$	4	4	4	8	7	4
$65 \times 65$	2	4	4	6	11	4
$129 \times 129$	2	4	4	6	14	4
$257 \times 257$	3	4	4	7	21	4

## Optimal and scalable solver combinations

From the above runs there are three candidate solvers with apparently grid-independent iterations, namely RS+GMG, RS+AMG, and SS+AMG. However, a look at `-log_view` output shows that the first is the most efficient. For example, on the finest ( $257 \times 257$ ) grid, in the runs in Table 12.2, RS+AMG did 2.4 times, and SS+AMG 8.4 times, more flops than RS+GMG. The big difference is in the PCApply event.

We can make the fastest solver RS+GMG even more efficient by using the default two smoother sweeps (i.e., `-mg_levels_ksp_max_it 2`) and by switching the smoother to using `-mg_levels_ksp_type richardson` instead of the default `chebyshev`. We now test this combination for algorithmic complexity, convergence, and parallel performance.

First, a graph of flops/ $N$  as a function of  $N$  is nearly level, other than a wobble of one SNES iteration on two grids, up to a  $4097 \times 4097$  grid with  $N > 10^7$  degrees of freedom (Figure 12.3). Thus, despite greatly reduced solution regularity relative to the corresponding Poisson problem, we have good evidence of solver optimality (Chapter 7) on this nonlinear and inequality-constrained problem.

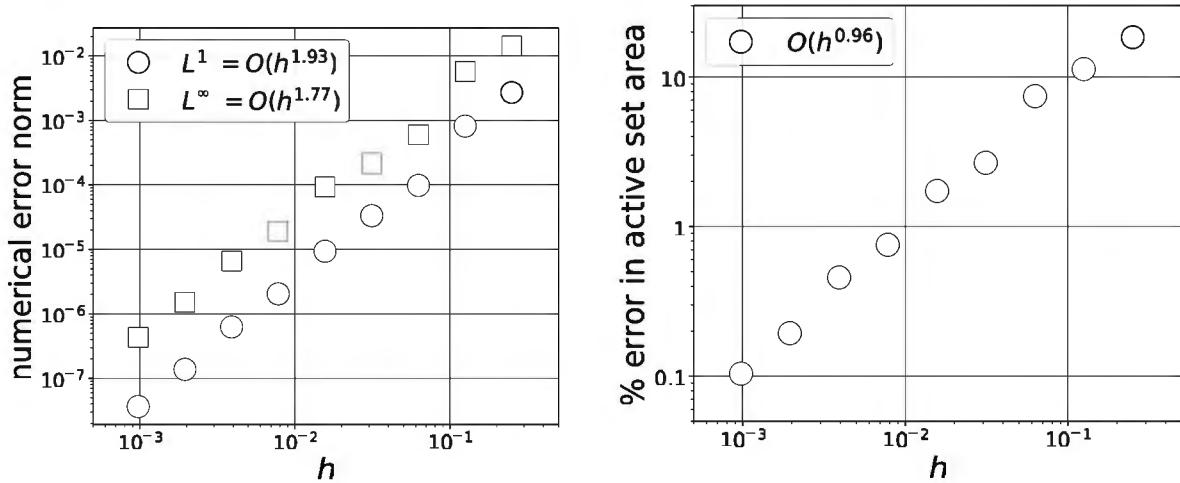


**Figure 12.3.** A grid-sequenced, GMG-preconditioned RS solver shows nearly constant flops/ $N$  as a function of  $N$ .

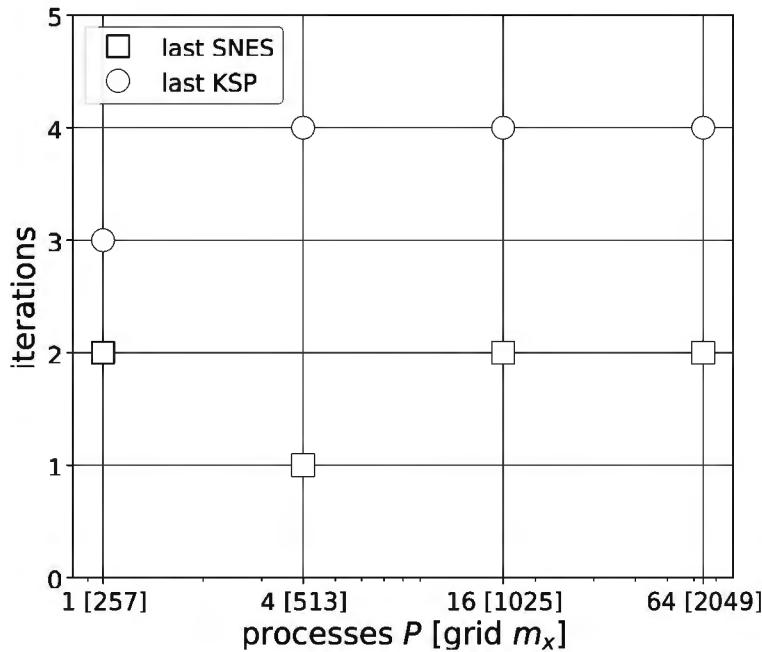
Regarding convergence, the left part of Figure 12.4 shows how the average ( $|\Omega|^{-1}\|u - u_{\text{exact}}\|_{L^1}$ ) and maximum ( $\|u - u_{\text{exact}}\|_{L^\infty}$ ) error norms depend on the grid spacing  $h$ . By fitting results from the seven finest grids, we see convergence rates relatively close to  $O(h^2)$ . Given that our centered FD discretization from Chapter 6 has  $O(h^2)$  local truncation error for smooth solutions, and given that the solution of this obstacle problem has only bounded, but not continuous, second derivatives, i.e.,  $u \in W^{2,\infty}(\Omega) \setminus C^2(\Omega)$ , such rates are all that could be desired.

The right side of Figure 12.4 shows a measure of convergence which is specifically relevant to inequality constrained problems. For this particular problem the coincidence (active) set is a disc centered at the origin with known radius and area (Figure 12.2). By assigning area  $h^2$  to each grid point in the numerically computed active set we can compute the percentage error in active-set area. This quantity converges at a rate just slower than  $O(h^1)$  (Figure 12.4), which is the best possible because the numerical solution cannot determine the location of the free boundary to less than one grid space  $h$ . (Note the interpolant of the exact solution also has  $O(h^1)$  active-set area error.)

We end with a quick weak-scaling study (Chapter 8), only looking at flops performance (and not run time). For parallel multigrid we choose overlapping ASM and SSOR as the smoother and a large coarse grid to allow redundant (Chapter 7) treatment, but otherwise the solver is the same as our preferred serial solver:



**Figure 12.4.** The numerical error norms  $\|u - u_{\text{exact}}\|$  (left) and active-set area errors (right) converge at close to the desired rates.



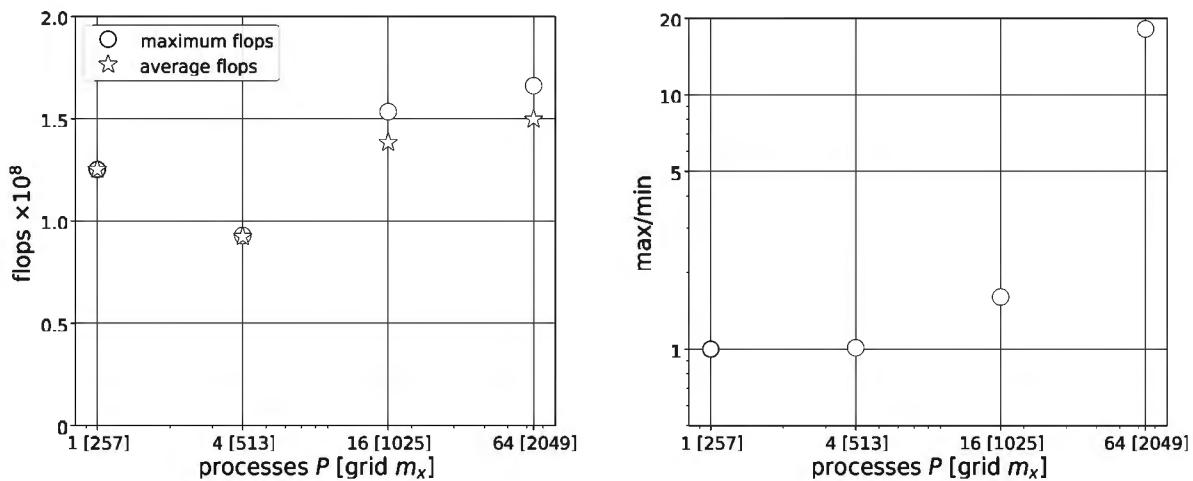
**Figure 12.5.** SNES iterations on the finest grid, and KSP iterations during the last SNES iteration, in weak-scaling runs.

```
$ mpiexec -n P ./obstacle -pc_type mg -mg_levels_ksp_type richardson \
-mg_levels_pc_type asm -mg_levels_sub_pc_type sor \
-da_grid_x 17 -da_grid_y 17 -snes_grid_sequence LEV -log_view
```

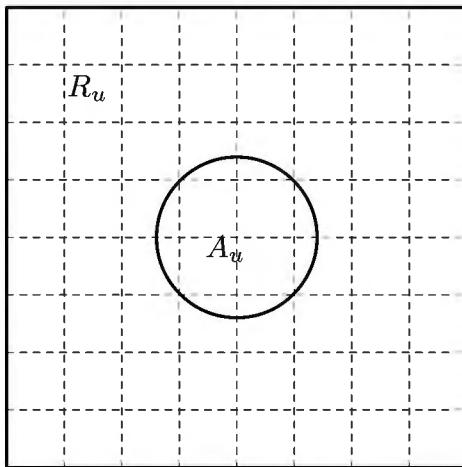
For \$P = 1, 4, 16, 64\$ we choose \$\text{LEV} = 4, 5, 6, 7\$, respectively; each process owns a \$257 \times 257\$ grid with roughly \$7 \times 10^4\$ degrees of freedom.

In a weak scaling solver we expect both the maximum and average flops over all processes to be nearly constant, and that is what we see. The SNES and KSP counts (Figure 12.5) are small and nearly constant; this parallel smoother is nearly as good as in serial. In maximum and average flops (Figure 12.6; left) we also see what we want.

However, the Max/Min values, the imbalance ratio, which would be close to 1 for good load balance, suddenly jumps above 18 at \$P = 64\$ (Figure 12.6; right). The reason, which has to do



**Figure 12.6.** Maximum and average flops per process are nearly constant (left), but the flops imbalance ratio (max / min) jumps upward at  $P = 64$  (right).



**Figure 12.7.** With  $P = 64$  processes, laid-out by the DMDA as shown (dashed grid), four processes own only active-set grid points. (Compare Figure 12.2.)

with our particular obstacle problem, is shown in Figure 12.7. There is a big difference in solver effort between the grid points in the active set of the converged solution (i.e.,  $A_u$ ), i.e., which correspond to trivial equations, and the grid points in the complementary inactive set ( $R_u$ ) where a PDE is being solved. This extreme level of imbalance is special to the RS solver; in RS the active-set degrees of freedom are not included in the linear system. This seems unfair—some processes do nothing while the others labor away—but it is not a serious barrier to weak scaling.

---

## Exercises

- 12.1. Show that  $K_\psi$  defined in (12.6) is closed and convex. Use the Poincaré inequality on  $W_0^{1,2}(\Omega)$  [51] to show that  $I[u]$  defined by (12.7) is strictly convex and coercive on  $W_g^{1,2}(\Omega)$ . Then show (by contradiction) that any minimizer (12.8) is unique. Finally show that a solution  $u \in K_\psi$  exists for formulation (12.8). (See section 8.4.2 of [51].)

- 12.2. Show the equivalence of formulations in Theorem 12.1. (*Consider the function  $f(\epsilon) = I[u + \epsilon(v - u)]$  for  $0 \leq \epsilon \leq 1$ . In one direction we know  $f(\epsilon) \geq f(0)$ , while in the other we can show  $f(\epsilon)$  is differentiable and nondecreasing.*)
- 12.3. (*The next two exercises may clarify the ideas behind, and the name of, the RS method.*) Suppose  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  is smooth and consider the equality-constrained optimization problem

$$\min_{A\mathbf{x}=\mathbf{b}} f(\mathbf{x}),$$

where  $A \in \mathbb{R}^{m \times n}$  has full row rank,  $\mathbf{b} \in \mathbb{R}^m$ , and  $0 \leq m \leq n$ . A *reduced-space Newton method* for this problem modifies the line-search Newton equations (Chapter 4) to ensure that the iterates remain in the feasible set  $\{\mathbf{x} \mid A\mathbf{x} = \mathbf{b}\}$ . One way to do this uses a *null-space matrix* [71, 118] for  $A$ , namely a full-rank matrix  $Z \in \mathbb{R}^{n \times (n-m)}$  whose columns form a basis of the null space of  $A$ . The method then solves step equations

$$(Z^\top H_f(\mathbf{x}^k) Z) \mathbf{v} = -Z^\top \nabla f(\mathbf{x}^k), \quad (12.23)$$

where  $H_f$  is the Hessian of  $f$ . Note that the  $(n-m) \times (n-m)$  matrix  $\hat{H} = Z^\top H_f(\mathbf{x}^k) Z$  is the *reduced Hessian* while  $Z^\top \nabla f(\mathbf{x}^k) \in \mathbb{R}^{n-m}$  is the *reduced gradient*. The Newton update is  $\mathbf{x}^{k+1} = \mathbf{x}^k + Z\mathbf{v}$ .

Show that if  $\mathbf{x}^k$  is feasible then  $\mathbf{x}^{k+1}$  is also feasible. Also show that if  $H_f(\mathbf{x}^k)$  is positive-definite then  $\hat{H}$  is also positive-definite, thus that the linear system (12.23) has a unique solution.

- 12.4. An *active set method* for the inequality constrained problem

$$\min_{A\mathbf{x} \geq \mathbf{b}} f(\mathbf{x}) \quad (12.24)$$

generates feasible iterates  $\mathbf{x}^k \in \{\mathbf{x} \mid A\mathbf{x} \geq \mathbf{b}\}$  and maintains a *working set*  $\mathcal{W}^k$  of the indices of the constraint equations  $i$  which are active at  $\mathbf{x}^k$ , i.e., for which the scalar equation  $\mathbf{a}_i^\top \mathbf{x} = b_i$  holds, where  $\mathbf{a}_i^\top$  is a row of  $A$ . At each iteration the method solves the *equality*-constrained problem in which the active constraints are treated as equalities. That is, it solves

$$\min_{\tilde{A}\mathbf{x} = \tilde{\mathbf{b}}} f(x),$$

where  $\tilde{A}$ ,  $\tilde{\mathbf{b}}$  are the rows of  $A$ ,  $\mathbf{b}$  corresponding to the indices in  $\mathcal{W}^k$ . Depending on the implementation, indices may enter and leave the working set based on a ratio test [71] or on the values of Lagrange multipliers. (The RS method from [15] uses the projected line search (12.19) and a modified residual (12.18) for these purposes.)

If  $A = I$  then (12.24) becomes a lower-bound constrained problem

$$\min_{\mathbf{x} \geq \mathbf{b}} f(\mathbf{x}). \quad (12.25)$$

Show that in this case null space matrices for the active constraints have entries of zero and one only, with actions that can be described using index sets. Then show that the reduced-space Newton equation (12.23) can be written

$$H_f(\mathbf{x}^k)_{\mathcal{I}^k, \mathcal{I}^k} \mathbf{s}_{\mathcal{I}^k} = -\nabla f(\mathbf{x}^k)_{\mathcal{I}^k}, \quad (12.26)$$

where  $\mathcal{I}^k$  is the index-set complement of  $\mathcal{W}^k$ . A direction  $\mathbf{s} \in \mathbb{R}^n$  is defined by adding  $\mathbf{s}_{\mathcal{W}^k} = 0$ . Solving linear system (12.26), which is the same as (12.17), is then followed by the usual update  $\mathbf{x}^{k+1} = \mathbf{x}^k + \lambda \mathbf{s}$  where  $\lambda$  comes from a line search. Finally note that problem (12.25) corresponds to NCP (12.15) under the substitutions  $\mathbf{w} = \mathbf{x} - \mathbf{b}$  and  $F(\mathbf{w}) = \nabla f(\mathbf{w} + \mathbf{b})$ .

- 12.5. Show that the Fischer-Burmeister function (12.20) is an NCP function. Next show that if  $\hat{F}$  defined by (12.21) then  $\hat{F}(\mathbf{w}) = 0$  if and only if  $\mathbf{w}$  solves NCP (12.15).
- 12.6. In an `obstacle.c` run with `-pc_type mg -da_refine 4`, and testing each SNESVI solver, try the visualization `-snes_vi_monitor_residual`. Compare `-snes_monitor_residual draw`. The latter view is less useful for a VI/CP problem; explain.
- 12.7. Modify `obstacle.c` to generate random but admissible initial iterates, that is, satisfying  $u^0 \geq \psi$  and  $u^0|_{\partial\Omega} = g$ . Show that on coarse grids the SNES iterations are insensitive to the differences among these initial iterates but on fine grids one generally sees large SNES iteration counts proportional to the movement, in grid cells, of the free boundary.
- 12.8. Currently the only way in PETSc to handle optimization problems with general linear inequality constraints is to add variables to convert to box constraints. For example, for constraints  $\mathbf{c} \leq A\mathbf{x} \leq \mathbf{d}$  one introduces new variables  $\mathbf{y} = A\mathbf{x}$  to give a larger problem in unknowns  $(\mathbf{x}, \mathbf{y})$ . The problem has box inequality constraints on  $\mathbf{y}$ , trivial inequality constraints on  $\mathbf{x}$  (i.e.,  $-\infty \leq \mathbf{x} \leq +\infty$ ), and new equality constraints  $A\mathbf{x} - \mathbf{y} = 0$ . In a fixed, low-dimensional vector space, construct such an example problem  $\min_{\mathbf{c} \leq A\mathbf{x} \leq \mathbf{d}} f(\mathbf{x})$ . Generate an equivalent problem with box constraints, decide on an optimization strategy subject to the new homogeneous, linear equality constraints, and demonstrate a SNESVI solver.
- 12.9. Consider the following porous-dam free-boundary problem in complementarity (i.e., NCP) form [9, 20]:

$$-\nabla^2 u + 1 \geq 0, \quad u \geq 0, \quad u(-\nabla^2 u + 1) = 0. \quad (12.27)$$

Here  $u(x, y)$  is the pressure, and the dam is saturated where  $u > 0$ . In [20] the domain is the rectangle  $\Omega = [0, a] \times [0, y_1]$  (Figure 12.8) with dimensions  $a = 16$  m and  $y_1 = 24$  m. Face CD has height  $y_2 = 4$  m. The boundary conditions are Dirichlet,

$$g(x, y) = \begin{cases} (y_1 - y)^2/2 & \text{on AB,} \\ ((a - x)y_1^2 + x y_2^2)/(2a) & \text{on BC,} \\ (y_2 - y)^2/2 & \text{on CD,} \\ 0 & \text{on DFA.} \end{cases} \quad (12.28)$$

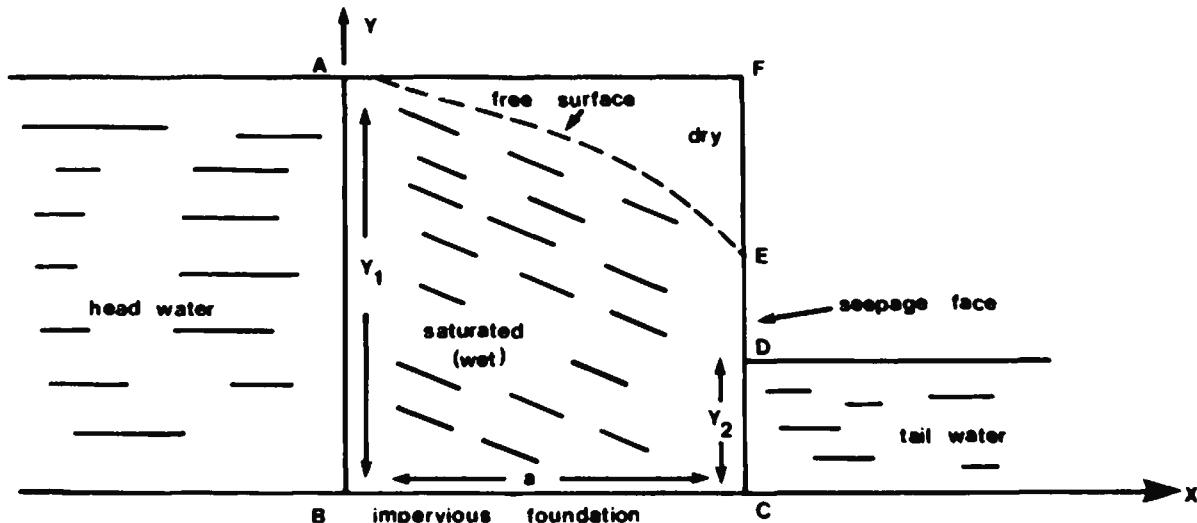
Modify `obstacle.c` to a code `dam.c` which will solve the above problem. Write a function for  $g(x, y)$ , but reuse Poisson equation tools from Chapter 6. A recommended coarse grid is  $3 \times 4$ , with equal spacing  $h_x = h_y = 8$  m. As no exact solution is known, a minimal verification is to compare results on a  $5 \times 7$  grid (`-da_refine 1`) to Table 4.1 in [20].

A modeling goal might be to compute the height of the seepage face ED in Figure 12.8. Write a function `GetSeepageFaceHeight()` which extracts this quantity from the solution; a threshold for “wet” will be needed. Note that the seepage face may be distributed across multiple processors so use `MPI_Allreduce()` with `MPIU_MAX`. Choose an optimal solver from this chapter to generate high-resolution results. (On centimeter-scale grids the author computes  $|ED| \approx 8.7$  m.)

- 12.10. Elastoplastic torsion failure [98, and references therein] involves an upper bound on the unknowns. In complementarity form on  $\Omega = (0, 1)^2$  the problem is

$$-\nabla^2 u \geq 2C, \quad u \leq \psi, \quad (\nabla^2 u + 2C)(\psi - u) = 0, \quad (12.29)$$

where  $u = 0$  along  $\partial\Omega$ . The upper obstacle is  $\psi(\mathbf{x}) = \text{dist}(\mathbf{x}, \partial\Omega)$ . In 2D this problem models the stress potential  $u$  of a bar with cross section  $\Omega$ . The applied torsion and other



**Figure 12.8.** The problem of partial saturation (seepage) of water through a porous, rectangular dam. Figure taken from [20].

physical properties are parameterized by  $C$ . The bar responds elastically at locations where the stress is below the threshold ( $u < \psi$ ), but plastic failure occurs at that threshold. The active set where  $u = \psi$  is thus the plastic-failure part, while on the inactive set an elastic model equation  $-\nabla^2 u = 2C$  applies.

Modify `obstacle.c` to `elasto.c` to solve the above problem. Again one may reuse Poisson equation tools from Chapter 6. Minimal verification is to compare to Figure 4.2 in [98], which shows results for  $C = 2.5, 10$ . Demonstrate an optimal solver.

- 12.11. Solve a minimal-surface equation (MSE; Chapter 7) obstacle problem. For example, make small modifications to `ch7/minimal.c` so that it solves the MSE version of the Example at the beginning of the current chapter. (Assume the same domain, obstacle, and Dirichlet boundary values.) Demonstrate an optimal solver, and then use high-resolution grids to compare the coincidence set and the free boundary to those computed by `obstacle.c`.

## Chapter 13

# Finite element method III: Firedrake and DMFlex

The next two chapters break the mould. Their Python [149] example codes use the Firedrake [126] finite element (FE) library, so direct calls to PETSC are avoided. Why? Because too much of our coding effort has been in *discretizing* PDEs using relatively standard methods. One way to explore solvers for more interesting PDEs is to make discretization someone else's responsibility. Firedrake is one way for the power of PETSc—its run-time-controllable solver stack—to break free of the discretization clutter.

In Firedrake application codes a PDE weak form is stated using only a few lines of the Unified Form Language (UFL; [5]). The UFL is a Python component of both the Firedrake and FEnICs [107] libraries, and these libraries allow users to apply the FE method without knowing how it is implemented. However, Firedrake adds an abstraction layer (PyOP2) which separates the local discretization from its parallel execution over the mesh [126]. Furthermore, key Firedrake data structures are actually PETSC objects, especially `DM`, `IS`, `Vec`, and `Mat` types, accessed through the `petsc4py` interface [39].

In the examples of the next two chapters, the discrete equations, linear or nonlinear, are solved using PETSC, with full command-line control of the solvers though familiar options. All of the Newton iteration, Krylov space, and preconditioning principles and options discussed so far remain available. Thus with Firedrake one can quickly state a PDE boundary-value problem, choose among a rich collection of FE methods and function spaces, use unstructured meshes in parallel, and have the full power of PETSC solvers, all with minimal programmer effort at the discretization stage. The contrast to our limited FE examples in Chapters 9 and 10, with their long C codes and restrictive discretization choices, is substantial.

In this short chapter we solve the Poisson equation yet again. However, Firedrake's discretization flexibility allows us to immediately play with higher-degree polynomial elements, and we demonstrate an elementary h/p FE method [87], essentially a spectral method [142]. We also look under the hood at how an unstructured mesh is managed by PETSC `DMFlex` objects [85, 95]. `DMFlex` is used by Firedrake [100] for the topology and geometry of unstructured meshes.

We omit any discussion of the Firedrake installation process, for which the reader should see the “Download” link at

[www.firedrakeproject.org](http://www.firedrakeproject.org).

Furthermore no Python programming introduction is attempted here. (However, certain advice specific to this Chapter's examples is in `p4pdः/python/README.md`.)

The Firedrake and FEnICs libraries share both FE ideas and the UFL. Thus the extensive FEnICs tutorial literature (e.g., [101, 107]) may be a useful supplement to the Firedrake documentation, modulo many details of syntax and solver control.

## The Poisson equation (one last time)

Recall the Poisson problem from Chapters 3, 6, and 10. Considering only Dirichlet boundary conditions,  $u \in W_g^{1,2}(\Omega)$  satisfies the weak form

$$\int_{\Omega} \nabla u \cdot \nabla v - fv = 0 \quad (13.1)$$

for all test functions  $v \in W_0^{1,2}(\Omega)$ . The corresponding strong form is  $-\nabla^2 u = f$  on  $\Omega$  and  $u = g$  on  $\partial\Omega$ .

A UFL statement of (13.1) is similar to the mathematical symbols themselves. In fact, the core of our first Firedrake code is in the following few lines which define a FE function space, state a weak form, and solve a discrete system  $\mathbf{F}(\mathbf{u}) = \mathbf{0}$ :

```
W = FunctionSpace(mesh, 'Lagrange', degree=k)
u, v = Function(W), TestFunction(W)
F = (dot(grad(u), grad(v)) - f_rhs * v) * dx
bc = DirichletBC(W, g_bdry, bdry_ids)
solve(F == 0, u, bcs=[bc], ...)
```

(The full code will be shown momentarily.) Note that additional formulas for  $f$  and  $g$  must define `Functions` `f_rhs` and `g_bdry`.

In order for the above lines of code to work correctly we must have defined either a triangular or quadrilateral mesh, including identification of the boundary nodes in an integer array `bdry_ids`. Information about the domain  $\Omega$  itself, in the `mesh` variable, is passed into the `FunctionSpace()` definition for `W` as above. While we only allow  $P_k$  or  $Q_k$  elements [36, 49] here, i.e., Lagrange elements, the degree  $k$  may be chosen at run time.

Note that we obtain both a `Function` `u` and a `TestFunction` `v` from the `FunctionSpace` `W`. The former (`u`) is a partly symbolic (i.e., partly UFL) object but it also has allocated space for the values at the nodes of the `mesh`, one real value for each degree of freedom associated to `W`. (The degrees of freedom for each element are actually basis functions for the dual space of polynomials of a given degree [36], as generated by the FIAT [107] and FInAT [83] components of Firedrake.) By contrast, a `TestFunction` is a purely symbolic object used in defining weak forms.

In defining the residual, integration over  $\Omega$  in (13.1) is implied by formally multiplying by `dx` in the UFL expression which defines the residual `F`. Then the `solve()` function sees the residual function and the Dirichlet boundary conditions, and FE assembly occurs at this point. That is, the above lines of code defining the weak form and boundary conditions actually store symbolic information for use inside the `solve()` function, and calling `solve()` generates a SNES callback (Chapter 4) under the hood.

Our entire Python program `fish.py` is shown in Code 13.1. It solves the same problem as the default case of `fish.c` in Chapter 6. For this particular problem the domain is the unit square  $\Omega = (0, 1)^2$  and the right-hand side  $f$  is manufactured using the exact solution  $u(x, y) = -xe^y$ .

```
#!/usr/bin/env python

from argparse import ArgumentParser, RawTextHelpFormatter
from firedrake import *
from firedrake.petsc import PETSc

# Read command-line options (in addition to PETSc solver options
# which use -s_ prefix; see below)
parser = ArgumentParser(description="")
```

```

Use Firedrake's nonlinear solver for the Poisson problem
-Laplace(u) = f      in the unit square
              u = g      on the boundary
Compare c/ch6/fish.c. The prefix for PETSc solver options is 's_'.
Use -help for PETSc options and -fishhelp for options to fish.py."",
    formatter_class=RawTextHelpFormatter,add_help=False)
parser.add_argument('--fishhelp', action='store_true', default=False,
                    help='help for fish.py options')
parser.add_argument('--mx', type=int, default=3, metavar='MX',
                    help='number of grid points in x-direction')
parser.add_argument('--my', type=int, default=3, metavar='MY',
                    help='number of grid points in y-direction')
parser.add_argument('--o', metavar='NAME', type=str, default='',
                    help='output file name ending with .pvf')
parser.add_argument('--k', type=int, default=1, metavar='K',
                    help='polynomial degree for elements')
parser.add_argument('--quad', action='store_true', default=False,
                    help='use quadrilateral finite elements')
parser.add_argument('--refine', type=int, default=-1, metavar='X',
                    help='number of refinement levels (e.g. for GMG)')
args, unknown = parser.parse_known_args()
if args.fishhelp: # -fishhelp is for help with fish.py
    parser.print_help()

# Create mesh, enabling GMG via refinement using hierarchy
mx, my = args.mx, args.my
mesh = UnitSquareMesh(mx-1, my-1, quadrilateral=args.quad)
if args.refine > 0:
    hierarchy = MeshHierarchy(mesh, args.refine)
    mesh = hierarchy[-1] # the fine mesh
    mx, my = (mx-1) * 2**args.refine + 1, (my-1) * 2**args.refine + 1
x,y = SpatialCoordinate(mesh)
mesh._topology_dm.viewFromOptions('--dm_view')
# to print coordinates: print(mesh.coordinates.dat.data)

# Define function space, right-hand side, and weak form.
W = FunctionSpace(mesh, 'Lagrange', degree=args.k)
f_rhs = Function(W).interpolate(x * exp(y)) # manufactured
u = Function(W) # initialized to zero here
v = TestFunction(W)
F = (dot(grad(u), grad(v)) - f_rhs * v) * dx

# Define Dirichlet boundary conditions
g_bdry = Function(W).interpolate(- x * exp(y)) # = exact solution
bdry_ids = (1, 2, 3, 4) # all four sides of boundary
bc = DirichletBC(W, g_bdry, bdry_ids)

# Solve system as though it is nonlinear: F(u) = 0
solve(F == 0, u, bcs = [bc], options_prefix = 's',
       solver_parameters = {'snes_type': 'ksponly',
                            'ksp_type': 'cg'})

# Print numerical error in L_infty and L_2 norm
elementstr = '%d' % ([P, Q][args.quad], args.k)
udiff = Function(W).interpolate(u - g_bdry)
with udiff.dat.vec_ro as vudiff:
    error_Linf = abs(vudiff).max()[1]
error_L2 = sqrt(assemble(dot(udiff, udiff) * dx))
PETSc.Sys.Print('done on %d x %d grid with %s elements:\n' %
               (mx, my, elementstr))
PETSc.Sys.Print('  error |u-uexact|_inf = %.3e, |u-uexact|_h = %.3e\n' %
               (error_Linf, error_L2))

```

```
# Optionally save to a .pvf file viewable with Paraview
if len(args.o) > 0:
    PETSc.Sys.Print('saving solution to %s ...' % args.o)
    u.rename('u')
    File(args.o).write(u)
```

**Code 13.1.** `python/ch13/fish.py`. A brief, powerful Poisson code.

The code is remarkably short given its flexibility and power. Using the prefix `-s_`, the solver can be controlled by hundreds of PETSC solver options, with possibilities listed via `-help` as usual. The code also reads a few options using the `argparse` library; these are listed by `-fishhelp`.

Looking into the code, the utility method `UnitSquareMesh()` generates a regular grid of triangles, laid out like the mesh in Figure 10.19, or a grid of rectangles (quadrilaterals) using option `-quad`. Via `DMFlex` (below), Firedrake manages these meshes as unstructured. The code in Chapter 14 will demonstrate how to read a mesh from a Gmsh-format file.

As with most examples in the book we solve  $\mathbf{F}(\mathbf{u}) = \mathbf{0}$  using PETSC SNES, but here we set the SNES type to `ksponly` because the problem is linear. The Jacobian matrix, generated by Firedrake using symbolic differentiation, is SPD so the CG method (Chapter 2) is a good KSP choice, and it is set as the default.

Note that the code could have been structured to directly solve a linear system using KSP. This requires different syntax in defining the weak form and invoking the solver:

```
W = FunctionSpace(mesh, 'Lagrange', degree=k)
u, v = TrialFunction(W), TestFunction(W)
a = dot(grad(u), grad(v)) * dx
L = f_rhs * v * dx
bc = DirichletBC(W, g_bdry, bdry_ids)
u = Function(W)
solve(a == L, u, bcs=[bc], ...)
```

Both arguments of the UFL version of the bilinear form  $a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v$  are symbolic, a `TrialFunction` and `TestFunction`, respectively. Later the solver needs a solution vector with values on the mesh, the discrete unknowns, so  $u$  is redefined as a `Function`. The code is otherwise the same (Exercise 13.1).

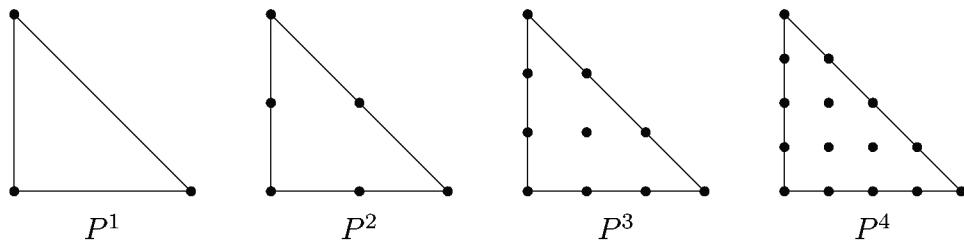
Running `fish.py` requires activation of the Python virtual environment in which Firedrake was built. Thus a first run looks something like this:

```
$ cd p4pdः/python/ch13
$ unset PETSC_DIR; unset PETSC_ARCH
$ source ~/firedrake/bin/activate
(firedrake) $ ./fish.py
done on 3 x 3 grid with P_1 elements:
error |u-uexact|_inf = 3.365e-03, |u-uexact|_h = 1.190e-03
```

Note that the uniform mesh (grid) is described in `fish.py` output by the number of vertices in each direction; this 9-node grid has 8 triangular elements.

We may immediately try higher-order finite elements and finer grids. For example we can try  $P_2$  elements (Figure 13.1) on a  $17 \times 17$  grid:

```
| (firedrake) $ ./fish.py -mx 17 -my 17 -k 2
```



**Figure 13.1.** Triangular Lagrange finite elements and their degrees of freedom.

Using  $Q_3$  quadrilateral elements (Chapter 9) is easy too:

```
| (firedrake) $ ./fish.py -refine 3 -quad -k 3
```

The next section expands on how numerical error depends on element degree  $k$ .

Because we are actually running PETSc under the hood, we can easily experiment with preconditioners, and indeed full control of the SNES/KSP/PC stack is available at the command line. For example, algebraic multigrid (AMG; Chapter 10), which depends only on the matrix entries, is available. In addition, because `MeshHierarchy()` generates refined grids (Code 13.1), Firedrake can set up interpolation and restriction operators so that geometric multigrid (GMG; Chapter 6) works too. For example, running the following with either `gamg` or `mg` for the PC type uses  $P_2$  elements and a  $129 \times 129$  fine grid to yield a solution with more than 10-digit accuracy:

```
| (firedrake) $ ./fish.py -refine 6 -k 2 -s_ksp_rtol 1.0e-12 -s_pc_type X
```

Add option `-s_ksp_converged_reason` to see that AMG takes 28 iterations while GMG takes 9. Using GMG preconditioning there are exactly 9 iterations for `-refine 3` through `-refine 7`, so the method is optimal as expected, while untuned AMG gives slowly growing iterations.

Regarding visualization of results we note that the graphical X viewers used with DMDA examples in previous chapters are not suited to DMplex unstructured meshes. However, Firedrake can save the solution to a `.pvf` file for viewing with Paraview:

```
| (firedrake) $ ./fish.py -refine 3 -o solution.pvf
| (firedrake) $ paraview solution.pvf
```

Use of Paraview can be reasonably straightforward, but we make no attempt to document it here; see [www.paraview.org](http://www.paraview.org).

## Polynomial-degree refinement

Until now we have regarded “refinement” as the process of making the mesh spacing small, and the number of unknowns corresponding large, while holding the discretization scheme fixed. However, with Firedrake we are also free to change the polynomial degree of the elements while holding the mesh fixed. These two modes of numerical improvement are called *h-refinement* and *p-refinement*, respectively [49]. *Spectral methods* [142] use pure *p*-refinement and *h/p finite element* methods [87] use both modes. However, the unqualified word “refinement” for FD, FE, and FV methods is understood to be *h*-refinement.

To illustrate *p*-refinement we fix a  $5 \times 5$  grid and increase the degree  $k$ :

```
| (firedrake) $ for K in 1 2 3 4 5 6 7 8; do \
|   ./fish.py -refine 1 -s_ksp_type preonly \
|   -s_pc_type cholesky -k $K; done
```

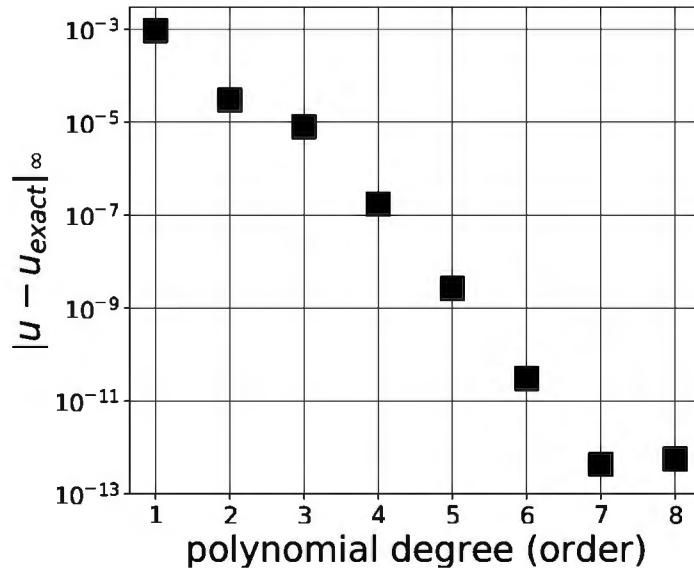


Figure 13.2. *p*-refinement rapidly reduces the numerical error.

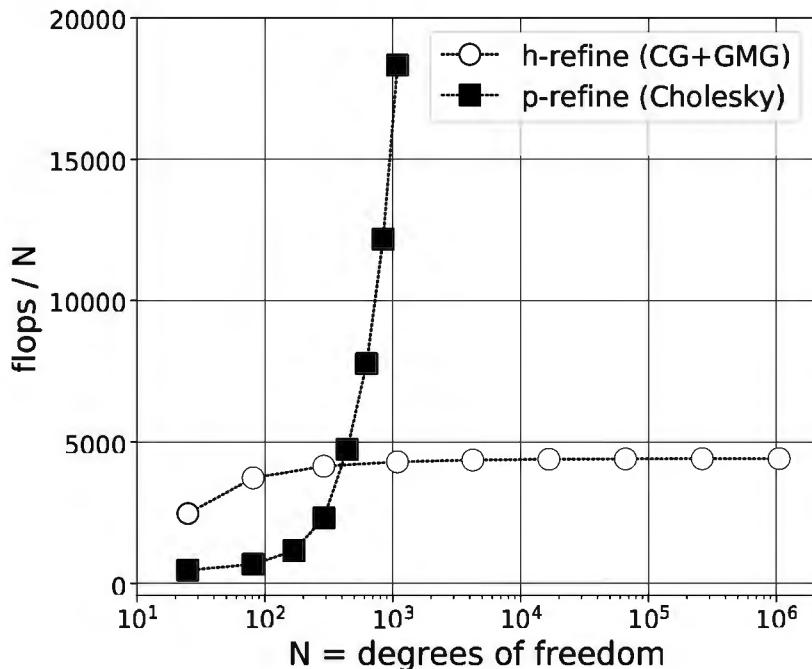
The numerical errors generated by these runs decrease rapidly because the solution is smooth and well-approximated by polynomials (Figure 13.2). It can be shown that, as the polynomial degree  $k \rightarrow \infty$ , and assuming exact arithmetic, the error would go to zero faster than any negative power of the  $k$  [87]. For  $P_7$  and  $P_8$  elements our solution is accurate to 12 digits. Given the conditioning of the matrices (not shown), this level of accuracy cannot be significantly improved when using double precision [143]. This level of accuracy is not achievable by *h*-refinement on the  $P_1$  method because we run out of memory (Exercise 13.2).

Matrices in the above loop increase in size from 25 to 1089 rows. For such small matrices it makes sense to use a direct solver, so here we have chosen Cholesky with nested-dissection ordering (Chapter 2). The matrices have a block pattern from the  $5 \times 5$  grid, but the blocks are relatively dense and overall sparsity is in the 4–8% range (Exercise 13.5).

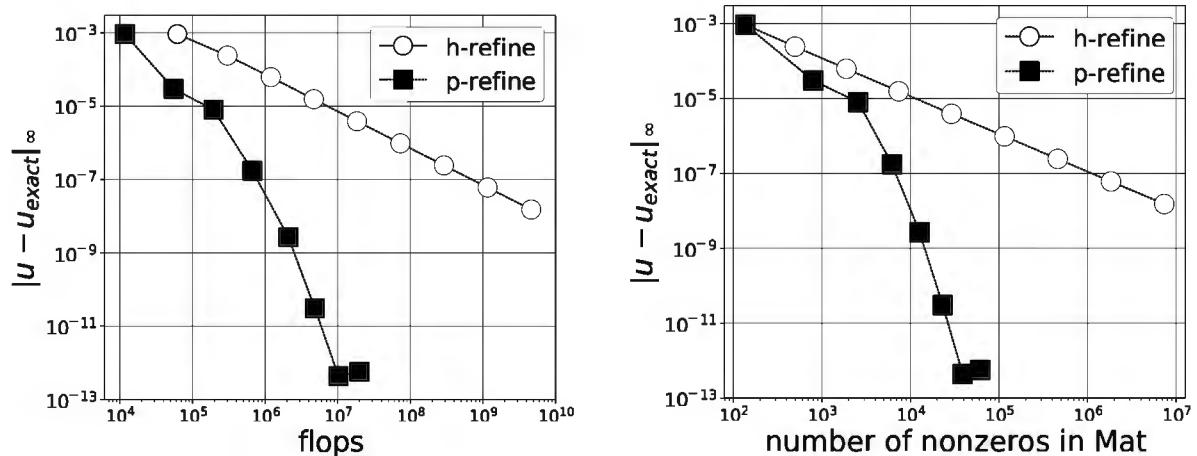
These *p*-refinement results make a mockery of the Chapter 7 definition of “optimal.” To illustrate this we let  $N$  be the total number of degrees of freedom and we graph flops/ $N$  versus  $N$  for both the *p*-refinement path above and an *h*-refinement path (Figure 13.3); the latter use  $P_1$  elements and CG+GMG preconditioning (Exercise 13.2). The *h*-refinement runs show optimality in our usual sense; the amount of work per degree of freedom becomes nearly constant. By contrast, in *p*-refinement the work per degree of freedom grows drastically. However, we can generate a highly accurate solution to the PDE using relatively few total degrees of freedom. Thus each degree of freedom in *p*-refinement *deserves* more effort than a degree of freedom for a fixed-polynomial-degree method, assuming the PDE solution is smooth [142]. It is thus a fact of life that the Chapter 7 definition has limitations.

**Fact 20.** Defining “optimal” as  $O(N)$  work for  $N$  discrete degrees of freedom makes no sense for spectral methods. *In a successful spectral method the degrees of freedom are worth much more than they are in a fixed-polynomial-degree FE/FD/FV method.*

To make this assertion quantitative we regard the numerical error as a function of the number of flops, i.e., accuracy as a function of effort. Now *p*-refinement is a clear winner (Figure 13.4, left). For example, we get numerical error norms of size  $10^{-7}$  with a thousand times fewer flops than the  $P_1$  method. The right side of Figure 13.4 shows the same conclusion relative to the number of nonzero entries in the matrix, i.e., the storage size. Furthermore, the same conclusions apply for quadrilateral  $Q_k$  elements (Exercise 13.4).



**Figure 13.3.** Measuring flops/ $N$  versus  $N$  suggests we should reevaluate our use of “optimal” when describing solvers.

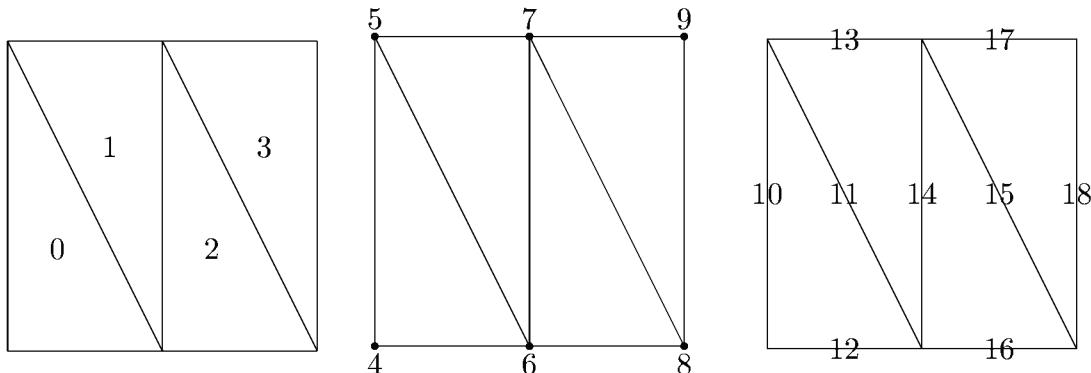


**Figure 13.4.** The  $p$ -refinement approach is the clear winner by an accuracy-versus-flops (left) or an accuracy-versus-nonzeros (right) standard.

## The underlying DMFlex object

We now exploit our simple Poisson code `fish.py` in a different way, by looking under the hood to understand the `DMFlex` type for managing unstructured meshes. This keeps a promise made at the end of Chapter 10, where we sketched how unstructured meshes are handled in parallel.

Generally `DM` objects describe both the topology (e.g., which vertices are connected by edges) and geometry (coordinates of vertices) of a mesh or grid. They can also describe how solutions and other fields are laid out on the mesh [85]. Firedrake, however, uses separate `DM` objects for the topology/geometry of the mesh, on the one hand, and for discretizations associated to particular problems on the other [100]. That is, there is one `DMFlex` for the mesh and one `DMShell` for



**Figure 13.5.** DMplex mesh index order: elements then vertices then edges.

each function space over that mesh. In the case of `fish.py`:

- `mesh._topology_dm` is a DMplex describing the topology and geometry of `mesh`, and the way it is distributed in parallel, while
- `W.dm` is a DMShell describing the data layout for FunctionSpace `W`.

Regarding the layout of data, the Stokes example in Chapter 14 illustrates how different FunctionSpaces in the same program, using the same mesh, may have different degrees of freedom associated with the mesh points.

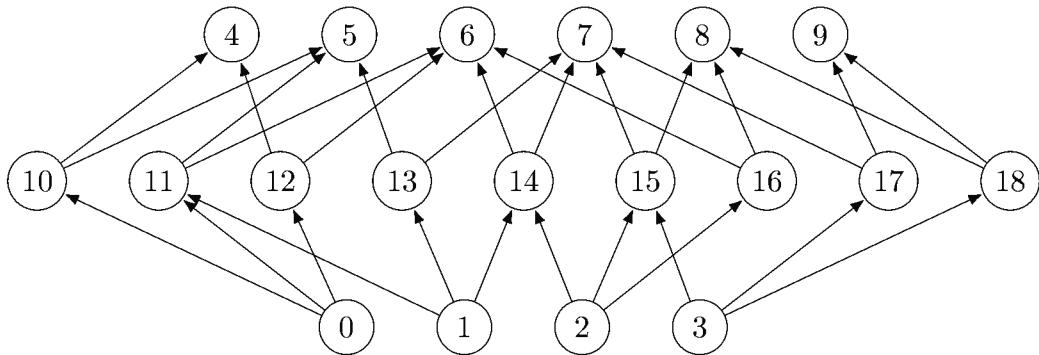
All pieces of a DMplex mesh, i.e., all vertices, edges, and elements, are called *points* [10], or *n*-cells, and every point has a unique index. Consider the example of the triangulation shown in Figure 13.5, which is a view of `mesh._topology_dm`:

```
(firedrake) $ ./fish.py -mx 3 -my 2 -dm_view
DM Object: DM... 1 MPI processes
  type: plex
DM... in 2 dimensions:
  0-cells: 6
  1-cells: 9
  2-cells: 4
...
```

Here the 0,1,2-cells are vertices, edges, and triangular elements, respectively. The index order shown in the figure is standard for DMplex [100]: elements first, vertices second, and edges third. In parallel the elements are strictly partitioned over processes (below), before the distribution of the vertices and edges, which is one reason why elements come first in the index order.

DMplex regards a mesh as a relation among points, namely via certain covering relationships [10]. For example, an edge is covered by two vertices, and a triangle by three edges. (Equivalently one may say that the two vertices are “incident to” the edge, and so on.) Thus, as shown in Figure 13.6, the topology information in a DMplex object may be regarded as a directed acyclic graph (DAG; [34, 100]) with an edge from a point to each of the points which cover it. As shown in the figure, this DAG is stratified by the topological dimension of the points, also called the *depth*.

Only the vertices have geometrical coordinates, i.e.,  $(x_4, y_4), \dots, (x_9, y_9)$ . (They are stored in a VectorFunctionSpace object `mesh.coordinates`. To reveal these coordinates do `print(mesh.coordinates.dat.data)`.) Assuming that the edges are linear and the elements



**Figure 13.6.** The same DM`Plex` mesh as in Figure 13.5, but as a directed acyclic graph (DAG) with three strata (horizontal levels).

are planar, the geometry of the mesh is fully defined by these vertex coordinates combined with the topological/combinatorial information in the DM`Plex`.

Finite element assembly requires calling DM`Plex` methods to get the indices of the points which form the *cone* [10, 85, 100] of a given point, namely all the points which cover it. One must also call a method to get the coordinates of the vertices (Exercise 13.6). In fact, recall that the residual function for a given FE method is assembled element by element by doing the integrals in the weak form (Chapter 10). For each such integral we need certain data on the edges which cover the element and on the vertices which cover those edges. Thus we need the DM`Plex` transitive *closure* operation on the element. For example, following edges upward from element 1 gives `closure(1)`, the edges and vertices needed to compute an integral over element 1. The reader should confirm that in Chapter 10 we (laboriously) implemented integrals using the element closure in exactly this way. Now we are happy to let Firedrake and DM`Plex` handle all the details.

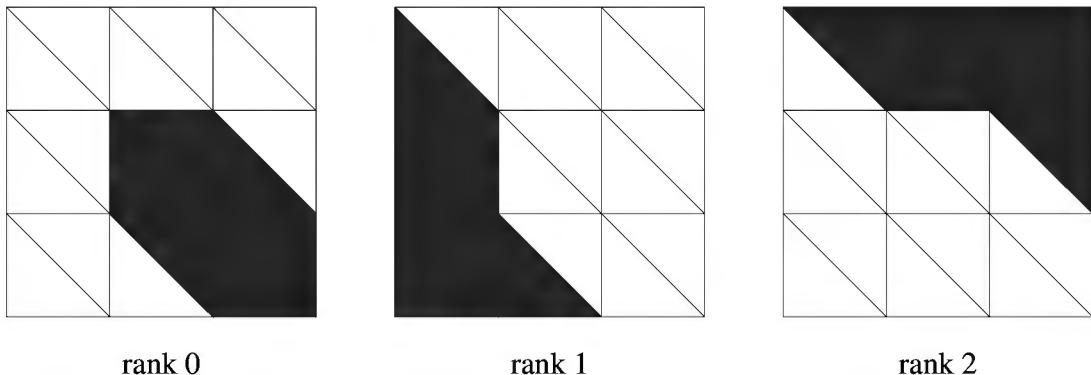
In parallel the mesh must be distributed so that each MPI process holds sufficient information for the residual and Jacobian functions to be assembled and for the solver to be applied on the process' part of the mesh. To give only a bit more detail, the elements are first strictly partitioned across the processes so that each rank owns a unique set of elements. However, all edges and vertices which cover an owned element must also be accessible on that process. This implies redundant storage of certain lower-dimensional points of the mesh, that is, the storage of *ghost* points for the closure of each element. It also follows that there is a distinction between local (with ghosts) and global (uniquely owned) Vecs, a familiar idea from DMDA structured grids (Chapter 3).

For example, consider how a mesh with 18 triangular elements is partitioned over 3 MPI processes:

```
(firedrake) $ mpiexec -n 3 ./fish.py -mx 4 -my 4 -dm_view
DM Object: Parallel Mesh 3 MPI processes
  type: plex
Parallel Mesh in 2 dimensions:
  0-cells: 7 8 8
  1-cells: 12 13 13
  2-cells: 6 6 6
  ...

```

The owned elements (2-cells) are strictly partitioned and equally distributed, with 6 on each MPI rank. However, a relatively obvious mesh decomposition into similar rectangular domains is *not* used here. The actual decomposition shown in Figure 13.7 is slightly more efficient; see Exercise 13.7.



**Figure 13.7.** Owned (shaded) elements in a mesh partitioned across three MPI processes.

## Exercises

- 13.1. Convert `fish.py` into a KSP-only code `kspfish.py` and check that it performs the same way.
- 13.2. Generate convergence results for `fish.py` using  $P_k$  and  $Q_k$  elements,  $k = 1, 2, 3$ , and  $h$ -refinement. Recommended runs use options

```
-s_pc_type mg -s_ksp_rtol 1.0e-14 -refine LEV -k K [-quad]
```

In particular, confirm  $O(h^2)$  convergence with  $P_1$  elements, but see [49] for the expected convergence rates with other elements.

- 13.3. Run in `c/ch6/` and `python/ch13/`, respectively:

```
| $ ./fish -da_refine 1 -ksp_view_mat :A6.m:ascii_matlab
| (firedrake) $ ./fish.py -refine 1 -s_ksp_view_mat :A13.m:ascii_matlab
```

and compare the matrices. (Consider sparsity patterns, diagonal entries, norms, and condition numbers.) Evaluate the choice of diagonal scale used in `ch6/fish.c`.

- 13.4. Add `-quad` to the runs which generated Figure 13.2 to create a figure for  $Q_k$  elements; our conclusions regarding  $p$ -refinement will not change. Compare  $h$ -refinement with  $Q_1$  elements and generate new versions of Figures 13.3 and 13.4.
- 13.5. Consider the  $p$ -refinement runs in Figure 13.2. By looking only at larger  $p$  values (e.g., 6, 7, 8), grids of different sizes (e.g.  $3 \times 3$  versus  $5 \times 5$ ), and either direct solvers (e.g., `-s_ksp_type preonly -s_pc_type cholesky`) or preconditioned Krylov methods (e.g., `-s_ksp_type cg -s_pc_type mg -s_ksp_rtol 1.0e-14`), and by counting flops versus numerical error norms, find a preferred high-accuracy solver for this Poisson problem when using substantial  $p$ -refinement.

- 13.6. Consider the serial run

```
| (firedrake) $ ./fish.py -mx 3 -my 2 -dm_view
```

Reproduce Figures 13.5 and 13.6 by adding code to `fish.py` which calls these methods of `mesh._topology_dm`, a DMFlex object:

- `getDepth()` to get the dimension, namely 2,
- `getChart()` to get the range of indices for the mesh, namely  $0, \dots, 18$ ,
- `getDepthStratum(x)` for  $x = 0, 1, 2$  to get the ranges of indices for vertices, edges, and elements, respectively,

- `getCoordinates().array` to get the coordinates of the vertices,
  - `getCone(x)` for  $x = 0, \dots, 3$  to get the indices of the edges which are the (topological) boundaries of the four elements, and
  - `getCone(x)` again for  $x = 10, \dots, 18$  to get the indices of the vertices which are the boundaries of the nine edges.
- 13.7. Modify `fish.py` so that you can identify the global and local element, edge, and vertex decompositions in the parallel run
- ```
| (fiaddrake) $ mpexec -n 3 ./fish.py -mx 4 -my 4 -dm_view
```
- (In addition to the methods in the previous exercise, `getCoordinatesLocal()` is useful too.) Reproduce Figure 13.7.

## Chapter 14

# Stokes equations (with Firedrake)

The Python code in this chapter uses Firedrake to solve the Stokes model of a viscous fluid. After glossing the physics of fluids we introduce the strong and weak forms of this model, and then we apply mixed finite element (FE) methods [49] which approximate the velocity and pressure variables from different spaces. The resulting discrete Stokes equations are symmetric but indefinite, with a natural block structure. The stability of mixed FE methods relates to the spectrum of this block matrix, which we will make an effort to understand, with important consequences for preconditioning.

Achieving high performance requires effective preconditioning, of course, but this is non-trivial. We recall the `fieldsplit` PC type, introduced in Chapter 7, which composes a preconditioner from chosen PCs on the blocks. For the velocity-velocity block of the Stokes system, essentially a Laplacian, as in previous chapters we precondition with geometric multigrid. The key additional step is *Schur-complement preconditioning* for the pressure variables, as supported by `fieldsplit`. With these tools we can demonstrate convergent, optimal, and weak-scaling Stokes solvers for a lid-driven cavity problem, on 2D unstructured meshes, based on composed `fieldsplit`-Schur-multigrid preconditioners. We end with a high-resolution solution for corner eddies.

Readers unfamiliar with Firedrake should at least read Chapter 13 before this one.

## Incompressible viscous fluids

A *fluid* is a mathematical abstraction in which the density, velocity, and stresses in a large collection of interacting particles are described by continuous functions. Of course, this useful abstraction underlies many practical scientific and engineering models, and for background on fluid equations see [2, 153, 156]. Here we present only enough of the theory to state the Stokes model for very viscous incompressible fluids.

Let  $\Omega \subset \mathbb{R}^d$  be a domain with well-behaved boundary; only dimensions  $d = 2, 3$  are considered. For  $t > 0$  and  $\mathbf{x} \in \Omega$ , let  $\mathbf{u}(t, \mathbf{x}) \in \mathbb{R}^d$  be the (vector) velocity and  $p(t, \mathbf{x})$  the (scalar) pressure. Let  $\rho > 0$  be the density which, for simplicity, we take to be constant. An incompressible, isotropic, viscous fluid is described by the following equations [2]:

$$\rho \left( \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = \nabla \cdot \boldsymbol{\sigma} + \mathbf{f}, \quad (14.1)$$

$$\nabla \cdot \mathbf{u} = 0, \quad (14.2)$$

$$\boldsymbol{\sigma} = 2\mu D\mathbf{u} - pI. \quad (14.3)$$

Here  $\sigma = \sigma_{ij}$  is the (Cauchy) *stress tensor*,  $\mathbf{f}$  is a *body force*,  $\mu$  is the (*dynamic*) *viscosity*, and  $D\mathbf{u} = (D\mathbf{u})_{ij}$  is the *strain rate tensor*, namely the symmetrized gradient

$$D\mathbf{u} = \frac{1}{2} (\nabla \mathbf{u} + \nabla \mathbf{u}^\top). \quad (14.4)$$

Note that  $\mathbf{f}$  has units of force per volume, and it may be in the form  $\mathbf{f} = \rho \mathbf{g}$ , where  $\mathbf{g}$  is the acceleration of gravity. In general these scalars, vector fields, and tensors will vary in space and time.

The notation used above may be a stumbling block. Using the 3D coordinate notation  $\mathbf{x} = (x_0, x_1, x_2)$ , the velocity gradient  $\nabla \mathbf{u}$  is a matrix,

$$\nabla \mathbf{u} = \begin{bmatrix} \frac{\partial u_0}{\partial x_0} & \frac{\partial u_0}{\partial x_1} & \frac{\partial u_0}{\partial x_2} \\ \frac{\partial u_1}{\partial x_0} & \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} \\ \frac{\partial u_2}{\partial x_0} & \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} \end{bmatrix} \iff (\nabla \mathbf{u})_{ij} = \frac{\partial u_i}{\partial x_j}.$$

Note that the rows of  $\nabla \mathbf{u}$  are the gradients of the components  $u_i$ . To compute the divergence of a tensor, such as  $\nabla \cdot \sigma$  in (14.1), one takes divergences of the columns of  $\sigma = [\sigma_{ij}]$ :

$$\nabla \cdot \sigma = \left\langle \frac{\partial \sigma_{00}}{\partial x_0} + \frac{\partial \sigma_{10}}{\partial x_1} + \frac{\partial \sigma_{20}}{\partial x_2}, \frac{\partial \sigma_{01}}{\partial x_0} + \frac{\partial \sigma_{11}}{\partial x_1} + \frac{\partial \sigma_{21}}{\partial x_2}, \frac{\partial \sigma_{02}}{\partial x_0} + \frac{\partial \sigma_{12}}{\partial x_1} + \frac{\partial \sigma_{22}}{\partial x_2} \right\rangle.$$

As the tensor  $D\mathbf{u}$  is symmetric, and thus by (14.3) the tensor  $\sigma$  is also symmetric, column/row distinctions will not be critical in our equations.

Equations (14.1)–(14.3) form the *Navier-Stokes model*. The velocity  $\mathbf{u}$ , pressure  $p$ , and stress tensor  $\sigma$  are usually treated as the unknowns. Vector equation (14.1) states *momentum conservation* [119], essentially Newton's second law  $m \mathbf{a} = \mathbf{F}$ , where the  $\partial \mathbf{u} / \partial t + \mathbf{u} \cdot \nabla \mathbf{u}$  is the acceleration of a fluid packet as it is carried along by the motion. Scalar equation (14.2) is *incompressibility*, a constraint on the flow. Tensor equation (14.3), the *constitutive relation* or *flow law* [69], says that the rates of deformation  $D\mathbf{u}$  of a small packet of fluid determine the stresses which that packet applies to its neighbors (and vice versa by Newton's third law). More precisely, the stresses other than pressure, i.e., the *deviatoric stresses*  $\sigma + pI$ , are proportional to  $D\mathbf{u}$  with coefficient  $2\mu$ . (This is the meaning of viscosity  $\mu$ .) Note that if the stresses  $\sigma$  are known then equations (14.2) and (14.3) determine the pressure:  $\text{tr}(D\mathbf{u}) = \nabla \cdot \mathbf{u} = 0 \implies p = -\text{tr}(\sigma)/d$ .

We can clarify the physical meaning by considering units. Stresses like  $\sigma$  and  $p$  are in Pascals,  $\text{Pa} = \text{N m}^{-2}$ , or force per area, while the dynamic viscosity  $\mu$  has units  $\text{Pa s} = \text{kg m}^{-1} \text{s}^{-1}$ . On the other hand, the spatial derivatives of velocity appearing in (14.1)–(14.3) have units of  $\text{s}^{-1}$ , so-called *strain rates*. The units for equation (14.1) are thus  $\text{Pa m}^{-1} = \text{kg m}^{-2} \text{s}^{-2} = \text{N m}^{-3}$ , so this equation either balances stress gradients or forces per unit volume according to taste.

Equations (14.1)–(14.3) exhibit two flavors of nonlinearity. First is the convective derivative  $\mathbf{u} \cdot \nabla \mathbf{u}$ , which accounts for turbulence [2]. Second is the possibility that the viscosity depends on deformation rates, for example when  $\mu = \mu(|D\mathbf{u}|)$  as happens in geodynamics and glaciers [108, 133], for example. However, while PETSc is well suited to such nonlinear problems, we will only consider the *linear Stokes model* in this chapter. This model arises from (14.1)–(14.3) by first setting the entire acceleration term on the left side of (14.1) to zero. That is, the Stokes model is for slow fluids in which the forces of inertia are neglected. Second, we will assume that the fluid is *Newtonian*, meaning that the dynamic viscosity  $\mu = \mu(\mathbf{x})$  is solution-independent. If we also eliminate the stress tensor then the equations combine to become the system

$$-\nabla \cdot (2\mu D\mathbf{u}) + \nabla p = \mathbf{f}, \quad (14.5)$$

$$\nabla \cdot \mathbf{u} = 0. \quad (14.6)$$

Vector equation (14.5) states that, in a slow fluid, the viscous stress and pressure gradients balance the body force. A less casual argument than this one would derive (14.5) by scaling the equations with the dimensionless Reynold's number  $R$  and then sending  $R \rightarrow 0$ ; for such an argument see [2].

System (14.5)–(14.6) is the strong form of our problem, but a clear derivation of the weak form will be important for applications, for example when the viscosity is variable, and for understanding preconditioner choices based on block structure. The proof of the following lemma about the matrix trace  $\text{tr}(A) = \sum_{i=0}^{d-1} a_{ii}$  is easy (Exercise 14.1). It says that symmetric and skew-symmetric matrices are orthogonal in a certain inner product.

**Lemma 14.1.** *If  $A, B \in \mathbb{R}^{d \times d}$  then  $\text{tr}(A^\top) = \text{tr}(A)$  and  $\text{tr}(AB) = \text{tr}(BA)$ . If  $A$  is symmetric ( $A^\top = A$ ) and  $B$  is skew-symmetric ( $B^\top = -B$ ) then  $\text{tr}(AB) = 0$ .*

**Definition.** For square matrices  $A, B$  we define the *Frobenius inner product*:

$$A : B = \text{tr}(A^\top B) = \sum_{i,j=0}^{d-1} a_{ij} b_{ij}. \quad (14.7)$$

Now suppose that the boundary of the domain  $\Omega \subset \mathbb{R}^d$  is decomposed into well-behaved disjoint sets,  $\partial\Omega = \partial_D\Omega \cup \partial_N\Omega$ . On  $\partial_D\Omega$  the velocity  $\mathbf{u}$  is equal to known values  $\mathbf{g}_D$  (*Dirichlet conditions*), so we define a space of admissible velocity fields,

$$\mathcal{X}_D = \left\{ \mathbf{v} \in (W^{1,2}(\Omega))^d \mid \mathbf{v} = \mathbf{g}_D \text{ on } \partial_D\Omega \right\}. \quad (14.8)$$

Let  $\mathcal{X}_0 = (W_0^{1,2}(\Omega))^d$  be the space with  $\mathbf{v} = 0$  on  $\partial_D\Omega$ . On  $\partial_N\Omega$ , with outward normal  $\mathbf{n}$ , a boundary stress  $\mathbf{g}_N$  is applied (*Neumann conditions*):

$$\sigma\mathbf{n} = (2\mu D\mathbf{u} - pI)\mathbf{n} = \mathbf{g}_N. \quad (14.9)$$

(Here “ $\sigma\mathbf{n}$ ” denotes the matrix-vector product, and  $\mathbf{n}$  is a column vector.) A solution of the Stokes problem is a velocity-pressure pair  $\mathbf{u} \in \mathcal{X}_D$  and  $p \in L^2(\Omega)$ .

In order to apply the FEM, or to state the problem in UFL/Firedrake, we need the weak form [19, 49]. It arises from integration by parts as follows. If  $\mathbf{u}, p$  are assumed to be smooth, and if  $\mathbf{v} \in \mathcal{X}_0$  is a velocity test function, and if  $\partial\Omega$  is nice enough to apply the divergence theorem, then multiplying (14.5) by  $\mathbf{v}$  and integrating yields

$$\int_\Omega -[\nabla \cdot (2\mu D\mathbf{u})] \cdot \mathbf{v} + \int_\Omega \nabla p \cdot \mathbf{v} = \int_\Omega \mathbf{f} \cdot \mathbf{v}. \quad (14.10)$$

Recall the integration-by-parts formula which results from the product rule  $\nabla \cdot (\varphi \mathbf{w}) = \nabla \varphi \cdot \mathbf{w} + \varphi \nabla \cdot \mathbf{w}$  and the divergence theorem, namely

$$\int_\Omega \varphi \nabla \cdot \mathbf{w} = \int_{\partial\Omega} \varphi \mathbf{w} \cdot \mathbf{n} - \int_\Omega \nabla \varphi \cdot \mathbf{w}. \quad (14.11)$$

Apply this technique to the viscous-stresses term in (14.10), denote the columns of  $D\mathbf{u}$  by  $(D\mathbf{u})_i$

and write the test function  $\mathbf{v}$  in components  $v_i$ :

$$\begin{aligned} \int_{\Omega} [\nabla \cdot (2\mu D\mathbf{u})] \cdot \mathbf{v} &= \sum_{i=0}^{d-1} \int_{\Omega} \nabla \cdot (2\mu(D\mathbf{u})_i) v_i \\ &= \sum_{i=0}^{d-1} \int_{\partial\Omega} 2\mu(D\mathbf{u})_i v_i \cdot \mathbf{n} - \int_{\Omega} 2\mu(D\mathbf{u})_i \cdot \nabla v_i \\ &= \int_{\partial\Omega} (2\mu(D\mathbf{u})\mathbf{n}) \cdot \mathbf{v} - \int_{\Omega} 2\mu D\mathbf{u} : \nabla \mathbf{v}^{\top} \end{aligned}$$

The last integral uses the Frobenius product (14.7). Because symmetric and skew-symmetric matrices are orthogonal in this inner product it follows that

$$D\mathbf{u} : \nabla \mathbf{v}^{\top} = D\mathbf{u} : \left[ \frac{1}{2} (\nabla \mathbf{v} + \nabla \mathbf{v}^{\top}) - \frac{1}{2} (\nabla \mathbf{v} - \nabla \mathbf{v}^{\top}) \right] = D\mathbf{u} : D\mathbf{v}.$$

Finally, dealing also with the pressure term in (14.10) using integration-by-parts formula (14.11), we have the new form

$$-\int_{\partial\Omega} [(2\mu D\mathbf{u} - pI)\mathbf{n}] \cdot \mathbf{v} + \int_{\Omega} 2\mu D\mathbf{u} : D\mathbf{v} - p \nabla \cdot \mathbf{v} = \int_{\Omega} \mathbf{f} \cdot \mathbf{v}. \quad (14.12)$$

Observe that the boundary integral in (14.12) is zero over  $\partial_D\Omega$ , because  $\mathbf{v} = \mathbf{0}$  there. Furthermore, the Neumann boundary quantity (14.9) has arisen naturally in this calculation.

The two-equation *weak form of the Stokes model* follows by using (14.9) in (14.12). The second equation arises from multiplying the negative of (14.6) by a scalar test function  $q \in L^2(\Omega)$  and integrating:

$$\int_{\Omega} 2\mu D\mathbf{u} : D\mathbf{v} - \int_{\Omega} p \nabla \cdot \mathbf{v} = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} + \int_{\partial_N\Omega} \mathbf{g}_N \cdot \mathbf{v} \quad (14.13)$$

$$-\int_{\Omega} q \nabla \cdot \mathbf{u} = 0. \quad (14.14)$$

Observe that  $p, q \in L^2(\Omega)$  are not differentiated in these equations, and that only first derivatives appear on  $\mathbf{u}$  and  $\mathbf{v}$ . The symmetric block structure seen later arises from the fact that the negative divergence is the adjoint of the gradient.

To see the structure of (14.13)–(14.14) more clearly we define bilinear forms

$$a(\mathbf{w}, \mathbf{v}) = \int_{\Omega} 2\mu D\mathbf{w} : D\mathbf{v}, \quad b(\mathbf{w}, q) = - \int_{\Omega} q \nabla \cdot \mathbf{w}. \quad (14.15)$$

In terms of a combined, symmetric bilinear form

$$k(\mathbf{u}, p; \mathbf{v}, q) = a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) + b(\mathbf{u}, q), \quad (14.16)$$

and a linear functional

$$\ell(\mathbf{v}) = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} + \int_{\partial_N\Omega} \mathbf{g}_N \cdot \mathbf{v}, \quad (14.17)$$

the solution pair  $\mathbf{u} \in \mathcal{X}_D$ ,  $p \in L^2(\Omega)$  satisfies

$$k(\mathbf{u}, p; \mathbf{v}, q) = \ell(\mathbf{v}) \quad \text{for all } \mathbf{v} \in \mathcal{X}_0 \text{ and } q \in L^2(\Omega). \quad (14.18)$$

“Weak form of the Stokes model” will, from now on, refer to either (14.13)–(14.14) or (14.18); they are equivalent.

The symmetry of bilinear forms  $a$  and  $k$  suggests a connection to optimization (Exercise 14.5). However, as shown in detail below, the symmetric matrix formed by discretizing  $k$  is indefinite, with both positive and negative eigenvalues, so (14.18) is not the extremal condition of a coercive functional. Instead the solution comes from minimizing in some directions and maximizing in others (Exercise 14.8). Supposing homogeneous boundary conditions for simplicity, one may observe that the velocity  $\mathbf{u}$  is the minimum of  $I(\mathbf{v}) = \int_{\Omega} \mu |D\mathbf{v}|^2 - \mathbf{f} \cdot \mathbf{v}$  over the subspace  $\mathcal{K} = \{\nabla \cdot \mathbf{v} = 0\} \subset \mathcal{X}_0$ , and then prove that (14.13) applies for some pressure, i.e., Lagrange multiplier,  $p \in L^2$  [51, Theorem 8.4.6]. While such a constrained-optimization approach allows one to show well-posedness of the continuum problem, a more direct approach, emphasizing the role of an inf-sup condition between the continuum spaces  $\mathcal{X}_0$  and  $L^2$ , is in [19, sections III.4 and III.6]. This condition will also arise as a practical stability consideration when choosing FE spaces.

If Dirichlet conditions apply on the entire boundary, namely if we are in the *enclosed flow* case  $\partial_D \Omega = \partial \Omega$ , and if  $c$  is any constant, then for  $\mathbf{v} \in \mathcal{X}_0$  we have

$$\int_{\Omega} c \nabla \cdot \mathbf{v} = c \int_{\partial \Omega} \mathbf{v} \cdot \mathbf{n} = 0. \quad (14.19)$$

That is, in this case (14.13) cannot have a unique solution for  $p$  because we may replace  $p$  by  $p + c$ . (Equivalently we observe that only  $\nabla p$  appears in (14.5) and not  $p$  itself.) In practice, nonuniqueness is resolved by telling the solver that the set of constant pressures is a null space. Also, under the same enclosed-flow hypothesis, incompressibility (14.6) implies

$$0 = \int_{\Omega} \nabla \cdot \mathbf{u} = \int_{\partial \Omega} \mathbf{g}_D \cdot \mathbf{n}. \quad (14.20)$$

That is, the average flow into  $\Omega$  must be zero, thus the Dirichlet boundary values  $\mathbf{g}_D$  must satisfy an integral condition.

The above derivation of the Stokes equations allows the viscosity  $\mu$  to vary in space. We only consider constant viscosity in computations, but our 2D code `stokes.py` (below) can handle variable viscosity  $\mu(x, y)$ . Furthermore one may easily extend the above derivation, and the code, to non-Newtonian fluids wherein viscosity depends on strain rates [108, 133]. On the other hand, when  $\mu$  is constant one may simplify the highest-order term in the momentum equation (14.5) into a *vector Laplacian*  $\nabla^2 \mathbf{u}$  (Exercises 14.2 and 14.3), i.e.,

$$-\mu \nabla^2 \mathbf{u} + \nabla p = \mathbf{f}. \quad (14.21)$$

This form is common in the literature (e.g., [49, 51]).

## Mixed FE methods and the discrete equations

“Mixed” finite element methods for the linear Stokes model enforce the weak form (14.18) over basis functions coming from distinct velocity and pressure spaces [49]. The resulting symmetric but indefinite matrix equations have a block structure in which the blocks correspond to the bilinear forms  $a$  and  $b$  in (14.15).

To show this structure, suppose we use test function spaces

$$\mathcal{V}^h \subset \mathcal{X}_0, \quad \mathcal{W}^h \subset L^2(\Omega) \quad (14.22)$$

with  $n_u$  (vector) velocity basis functions  $\phi_i \in \mathcal{V}^h$  and  $n_p$  (scalar) pressure basis functions  $\psi_i \in \mathcal{W}^h$ . Matrix entries are defined using the bilinear forms

$$A_{ij} = a(\phi_i, \phi_j), \quad B_{ij} = b(\phi_i, \psi_j).$$

Statement (14.18), enforced on  $\mathcal{V}^h \times \mathcal{W}^h$ , becomes the *discrete Stokes equations*

$$\begin{bmatrix} A & B^\top \\ B & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}. \quad (14.23)$$

The body force and boundary stresses combine into  $f_i = \ell(\phi_i)$ ; see (14.17).

To enforce Dirichlet conditions we must first extend  $\mathbf{g}_D$  to all of  $\Omega$  by using additional hat functions along the boundary (e.g., as in Chapter 10), and then adding trivial equations for the boundary nodes in  $\Omega_D$ , but the system of equations continues to have form (14.23). (The details will, in fact, be handled by Firedrake.) The discrete solution now consists of vectors  $\mathbf{u} \in \mathbb{R}^{n_u}$ ,  $p \in \mathbb{R}^{n_p}$  in bases  $\{\phi_i\}$ ,  $\{\psi_i\}$  respectively. Because of how the Dirichlet conditions are handled, the total number of degrees of freedom  $N = n_u + n_p$  includes the velocities and pressures along the full boundary  $\partial\Omega$ .

Observe that  $A \in \mathbb{R}^{n_u \times n_u}$  is an SPD matrix, and if  $\mu$  is constant then it discretizes the vector Laplacian  $-\mu\nabla^2$ . Matrix  $B \in \mathbb{R}^{n_p \times n_u}$  is the discretization of the negative divergence  $-\nabla \cdot$ . Because the transpose corresponds to integration by parts,  $B^\top$  is the discretization of the gradient  $\nabla$ . These blocks define the system matrix:

$$K = \begin{bmatrix} A & B^\top \\ B & 0 \end{bmatrix} \in \mathbb{R}^{N \times N}. \quad (14.24)$$

By writing out the corresponding scalar component equations in 2D, and ordering the velocity components  $\{u_x\}$  before  $\{u_y\}$ , (14.24) expands to

$$K = \begin{bmatrix} A_x & 0 & B_x^\top \\ 0 & A_y & B_y^\top \\ B_x & B_y & 0 \end{bmatrix}. \quad (14.25)$$

Here  $A_x, A_y$  are (discrete) scalar Laplacians and  $B_x, B_y$  are discretizations of  $-\partial/\partial x$  and  $-\partial/\partial y$ , respectively. However, because the velocity unknowns are actually interleaved  $\{(u_x)_0, (u_y)_0, (u_x)_1, (u_y)_1, \dots\}$ , block structure (14.25) is not what one sees when viewing the assembled matrix at run time. (On the other hand, use `-s_mat_type aij -s_ksp_view_mat` with the code below to show block decomposition (14.24).)

Firedrake will handle the assembly of the discrete equations, but the structure of block matrix  $K$  is still important. We will need to understand its spectral properties in order to make educated choices about efficient solvers, especially Krylov methods and preconditioners leading to optimal scaling. Thus we will have more to say about  $K$ , but first let us get a code running.

## A Stokes flow code

The Firedrake code `python/ch14/stokes.py` solves three different 2D boundary-value problems for the linear Stokes model, each on the unit square:

- The default is a *lid-driven cavity* on a unit square domain, a fluid dynamics favorite [49, 107]. Only the top boundary has nonzero (tangential) velocity, and, because the boundary conditions are Dirichlet, we tell the solver to remove the null space of constant pressure fields to make the problem well posed. Later we will illustrate a well-known phenomenon which arises in this example: corner eddies [2, 113].
- Option `-nobase` gives a variant of the default problem. It has homogeneous Neumann ( $\sigma \cdot \mathbf{n} = 0$ ) boundary conditions on the bottom, so fluid flows in and out, and it is well posed as is; the null space is trivial.

- Option `-analytical` gives a test problem from [107]. It can be derived by choosing a stream function, differentiating to get a divergence-free velocity field, choosing a simple pressure, and then solving equation (14.5) to generate the balancing body force (Exercise 14.14). Again the boundary conditions are all Dirichlet so the null space must be removed.

We show only short excerpts from `stokes.py`. The first specifies function spaces for a mixed FE method:

```
V = VectorFunctionSpace(mesh, 'CG', degree=K)
if dp:
    W = FunctionSpace(mesh, 'DG', degree=L)
else:
    W = FunctionSpace(mesh, 'CG', degree=L)
Z = V * W
```

Here the default `mesh` is a uniform triangulation of the unit square, but option `-quad` chooses quadrilaterals instead. One can also read an unstructured mesh from a file (e.g., the mesh shown later in Figure 14.9), and there is no restriction on the 2D domain  $\Omega$  as long as boundary conditions are consistently specified.

After initial creation, or having been read from a file, any mesh can be uniformly refined  $n$  times using option `-refine n`. In Firedrake one may set up a refinement hierarchy in which the initial mesh is the coarsest, and where each triangle is uniformly refined by a factor of four at each level. Because interpolation/restriction operators are also supported through Firedrake and DMPlex, our code thereby allows geometric multigrid (GMG, `-s_pc_type mg`); we will demonstrate this capability soon.

Options `-udegree` and `-pdegree` set the polynomial degrees  $K$  and  $L$  of elements in `V` and `W`, respectively, and a discontinuous pressure space is chosen by option `-dp`. For triangles, if  $K$  is one higher than  $L$  and the pressure space is continuous then  $Z=V*W$  is the *Taylor-Hood* element  $(P_{k+1})^2 \times P_k$  [139], with default  $P_2 \times P_1$ , the lowest-degree stable combination. We will compare several stable mixed methods below.

Next in the code we set boundary conditions. As the domain is  $\Omega = (0, 1)^2$ , the sides ( $x = 0, 1$ ) are labeled by 1, 2, the bottom ( $y = 0$ ) as 3, and the top ( $y = 1$ ) as 4. For the default and `nobase` problems the velocity on the top surface is tangential ( $\mathbf{u} \cdot \mathbf{n} = 0$ ) and quadratic ( $\mathbf{u} \cdot \mathbf{t} = x(1 - x)$ ). The code looks like this:

```
u_noslip = Constant((0.0, 0.0))
ux_lid = x * (1.0 - x)
u_lid = Function(V).interpolate(as_vector([ux_lid, 0.0]))
bcs = [ DirichletBC(Z.sub(0), u_noslip, (1,2,3)),
        DirichletBC(Z.sub(0), u_lid, (4,)) ]
```

In the default and `-analytical` problems the entire boundary is Dirichlet, thus we tell the solver that constant velocities form the null space of  $B$  (and  $K$ ):

```
ns = MixedVectorSpaceBasis(Z, [Z.sub(0), VectorSpaceBasis(constant=True)])
```

(No null space is set in the `-nobase` problem.) Observe that the  $A$  block in the system matrix  $K$  (14.24) is invertible as long as  $\partial_D\Omega$  is nonempty.

As with our earlier Poisson code, `stokes.py` solves a general, potentially nonlinear system  $\mathbf{F} = \mathbf{0}$  where, from the weak form (14.18),

$$\mathbf{F}(\mathbf{u}, p; \mathbf{v}, q) = a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) + b(\mathbf{u}, q) - \ell(\mathbf{v}). \quad (14.26)$$

(Recall that  $a$ ,  $b$ , and  $\ell$  are defined in (14.15) and (14.17).) A `Function` is needed to store the solution and `TestFunctions` are used in defining  $\mathbf{F}$ :

```

up = Function(Z)
u,p = split(up)
v,q = TestFunctions(Z)
Du = 0.5 * (grad(u)+grad(u).T)
Dv = 0.5 * (grad(v)+grad(v).T)
F = (2.0 * mu * inner(Du,Dv) - p * div(v) - div(u) * q \
      - inner(f_body,v)) * dx

```

Consider what Firedrake will do with this UFL description of the weak form. Using the chosen FE spaces it will create a residual function for SNES call-back, namely a computable function  $\mathbf{F} : \mathbb{R}^N \rightarrow \mathbb{R}^N$  corresponding to (14.26). Conceptually speaking, to do this it will generate an ordered basis  $\{(\phi_j, \psi_j)\}_{j=0}^{N-1}$  of the mixed FE space and represent the solution  $(\mathbf{u}, p)$  as a linear combination in this basis. Then it will compute the components of the residual,  $F(\mathbf{u}, p)_j = a(\mathbf{u}, \phi_j) + b(\phi_j, p) + b(\mathbf{u}, \psi_j) - \ell(\phi_j)$ , by looping over the elements to do local assembly, and the same ordered basis will supply test functions. The resulting function is generally similar to our naive FE residual in Chapter 10, for example, but now generated from a high-level UFL description of the problem.

A Jacobian function  $J_{\mathbf{F}}$  is also computed by Firedrake via symbolic differentiation from the UFL form of  $\mathbf{F}$  [126]. That is, a Jacobian call-back is created with no user effort! This is a huge advance over hand coding such a function and/or needing to fall back to `-snes_fd_color`.

The solver is now called in one line which sets boundary conditions, the null space, and solver parameters:

```

solve(F == 0, up, bcs=bcs, nullspace=ns, options_prefix='s',
      solver_parameters={...})

```

Our substantial discussion of solver parameters is deferred until after we address the spectral properties of the system matrix. For now we will show no more lines of code from `stokes.py`, but note it has fewer than 200 substantive lines. Its additional bookkeeping parts include parsing options, creating a mesh, packaging preconditioning options (covered below), and reporting on the solver and solution. Finally, option `-o` saves the solution in a Paraview-readable form.

A first run might simply show the assembled matrix  $K$  for a small grid:

```

(firedrake) $ ./stokes.py -refine 1 -s_ksp_view_mat
solving on 5 x 5 grid with P_2 x P_1 Taylor-Hood elements ...
Mat Object: (s_) 1 MPI processes
  type: nest
  Matrix object:
    type=nest, rows=2, cols=2
    MatNest structure:
      (0,0) : type=seqbaij, rows=162, cols=162
      (0,1) : type=seqaij, rows=162, cols=25
      (1,0) : type=seqaij, rows=25, cols=162
      (1,1) : type=seqaij, rows=25, cols=25

```

(Recall that running Firedrake require activation of the Python virtual environment; see Chapter 13.) We see here that a Firedrake mixed FE method exploits the MATNEST type, with a BAIJ (blocked) type for the velocity block and the AIJ type for the other blocks; compare forms (14.24) and (14.25). To actually see the sparsity pattern one must reset the matrix type:

```
| (firedrake) $ ./stokes.py -refine 1 -s_mat_type aij -s_ksp_view_mat draw
```

Figure 14.1 shows the result. As expected from (14.24),  $K$  has a large  $A$  block, a wide block  $B$  representing the (negative) divergence, its tall transpose  $B^\top$ , which is the gradient, and a small zero block to complete the diagonal, the dimensions of which appear in the MATNEST view.



**Figure 14.1.** Sparsity of  $K$  for a uniform  $5 \times 5$  grid and  $P_2 \times P_1$  elements.

The code sets the SNES type to KSPONLY, and the default KSP is GMRES, but for mixed FE methods using MATNEST Firedrake chooses Jacobi as the default PC. This would seem to be incorrect given that the diagonal of  $K$  has many zeros, but in fact the PCJACOBI type sets diagonal zeros to 1.0 automatically, so convergence is possible. However, these defaults do not form an adequate solver! For example, the runs

```
| (firedrake) $ ./stokes.py -refine LEV -s_ksp_converged_reason
```

exceed  $10^4$  KSP iterations at `-refine 3`, a very coarse  $17 \times 17$  (uniform) grid.

A first alternative is a direct solver. Indeed, the combination of nested-dissection LU with a diagonal-block shift (for invertibility), namely

```
-s_ksp_type preonly -s_pmat_type aij -s_pc_type lu \
-s_pc_factor_shift_type inblocks
```

succeeds in under one minute run time on levels up to `-refine 7` ( $257 \times 257$  grid). At that point the scaling and memory usage become too poor (as expected). Another combination is unpreconditioned MINRES:

```
-s_ksp_type minres -s_pc_type none
```

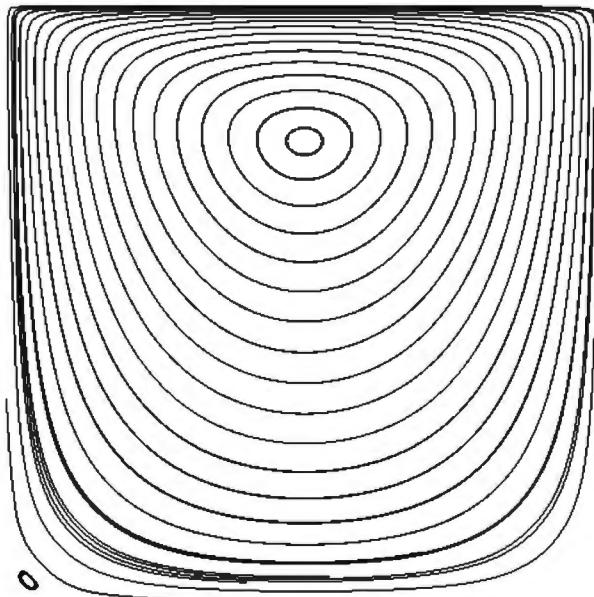
This gives solver convergence in roughly comparable time, on the same levels up to a  $257 \times 257$  grid, but with growing KSP iterations. However, neither of these solvers scales properly. In the next section we will address preconditioning more seriously.

For now we show an attainable result on a modest grid (Figure 14.2):

```
| (firedrake) $ ./stokes.py -refine 5 -s_ksp_type minres -s_pc_type none \
-s_ksp_converged_reason -o lid5.pvd
```

Option `-showinfo` shows that  $n_u = 3.3 \times 10^4$  and  $n_p = 4.2 \times 10^3$  in this case, thus  $N = 3.7 \times 10^4$ . (Note we will soon solve problems with  $N = O(10^7)$  in comparable time.)

The velocity field in the figure is visualized by streamlines [2] using the “Stream Tracer” functionality in Paraview. We see the expected flow, a clockwise motion because the tangential velocity on the lid is positive. There is also evidence of an eddy in the lower-left corner.



**Figure 14.2.** Streamlines from a low-resolution ( $65 \times 65$  uniform grid) solution.

## The discrete Stokes matrix and its spectral personality

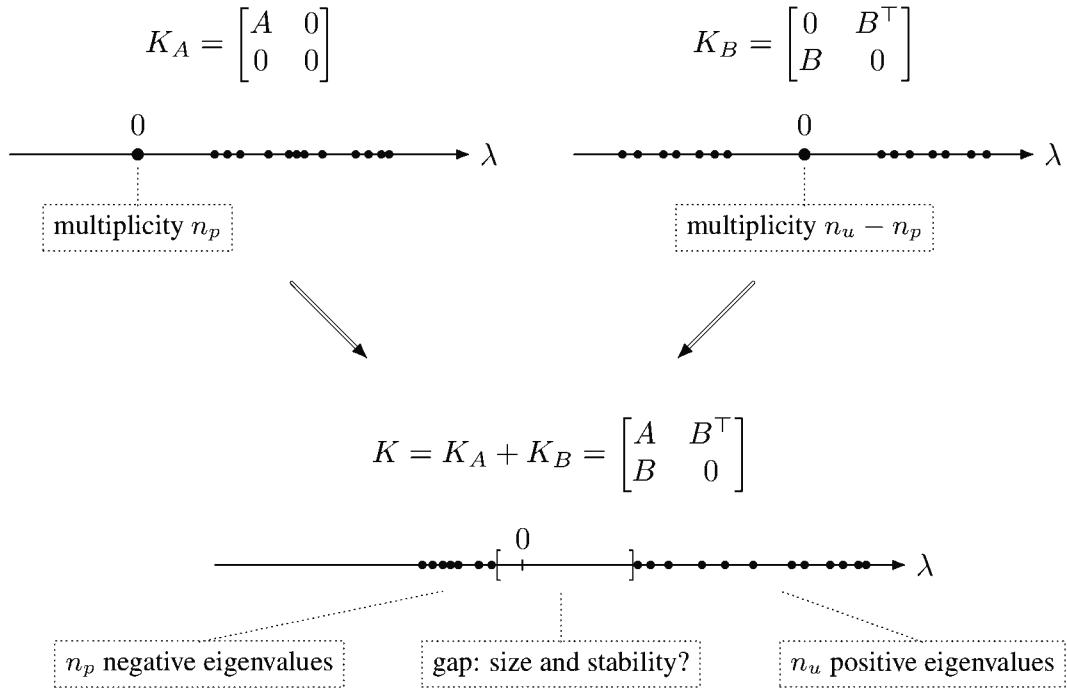
We will solve linear equations (14.23) by a Krylov method, and convergence depends essentially on the location of the eigenvalues of the preconditioned discrete Stokes matrix  $K$ . Therefore spectral properties of  $K$  need to be understood in order to make educated choices about such solvers.

As generality is not really needed for this and the next two sections, we assume that both  $\partial_D\Omega$  and  $\partial_N\Omega$  have positive measure. Thus the continuous Stokes problem is well posed [49]; the -nobase problem serves as a concrete example.

Recall from (14.24) that  $K \in \mathbb{R}^{N \times N}$  where  $N = n_u + n_p$  is the total number of degrees of freedom. Since  $K$  is symmetric it has only real eigenvalues and, by well-posedness, there will be no zero eigenvalues. The velocity block  $A$  is SPD, with two copies of the scalar Laplacian. Well-posedness also implies that  $B$  must be wide ( $n_u \geq n_p$ ) and have full row rank [49]. Similarly,  $B^\top$  has full column rank and a trivial null space (Exercise 14.4). However, the subspace  $\text{null}(B)$  needs to be large because it is the subspace of (discretely) divergence-free velocity fields in which we seek a velocity solution  $\mathbf{u}$  satisfying the momentum equation (14.5).

It is useful to decompose  $K = K_A + K_B$  and sketch a picture of how the spectrum of  $K$  is built from these parts; see Figure 14.3. It is easy to show that  $K_A$  and  $K_B$  separately have spectrum as shown (Exercises 14.6 and 14.7). Specifically, the spectrum of  $K_B$  is symmetric around the origin, and it consists of exactly  $n_p$  positive eigenvalues,  $n_p$  negative eigenvalues, and a zero eigenvalue of multiplicity  $n_u - n_p$ . Also, because  $BB^\top$  approximates the scalar operator  $-\nabla^2$ , the nonzero eigenvalues of  $K_B$  are approximate square roots of Laplacian eigenvalues.

While it is not true that eigenvalues are nice functions of matrix entries, and “ $\sigma(K) = \sigma(K_A) \cup \sigma(K_B)$ ” is false, we can see how the sketch is correct in important ways. Think of the eigenvalues of  $K$  as starting from those of  $K_B$  and being perturbed by viscosity. The  $\mu = 0$  case, an inviscid fluid, is a singular limit in which the momentum equation reduces to  $\nabla p = \mathbf{f}$ , so the pressure gradient balances the body force and the pressure is hydrostatic, but incompressibility also applies. This decoupled, nonunique limit includes a large space of velocity solutions in the null space of  $B$ , namely divergence-free fields. Adding  $\mu > 0$  then assigns a viscous-dissipation cost, determined by the action of  $A$ , on these modes. We therefore expect that the zero eigenvalues of the  $\mu = 0$  case will move to the right as  $\mu$  increases from zero.



**Figure 14.3.** A sketch of how  $\sigma(K)$  is built from parts of  $K$ .

To say this algebraically, the following factorization shows that  $K$  has the same number of positive eigenvalues as  $K_A$  and the same number of negative eigenvalues as  $K_B$ :

$$K = \begin{bmatrix} A & B^\top \\ B & 0 \end{bmatrix} = \begin{bmatrix} I & 0 \\ BA^{-1} & I \end{bmatrix} \begin{bmatrix} A & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I & A^{-1}B^\top \\ 0 & I \end{bmatrix}. \quad (14.27)$$

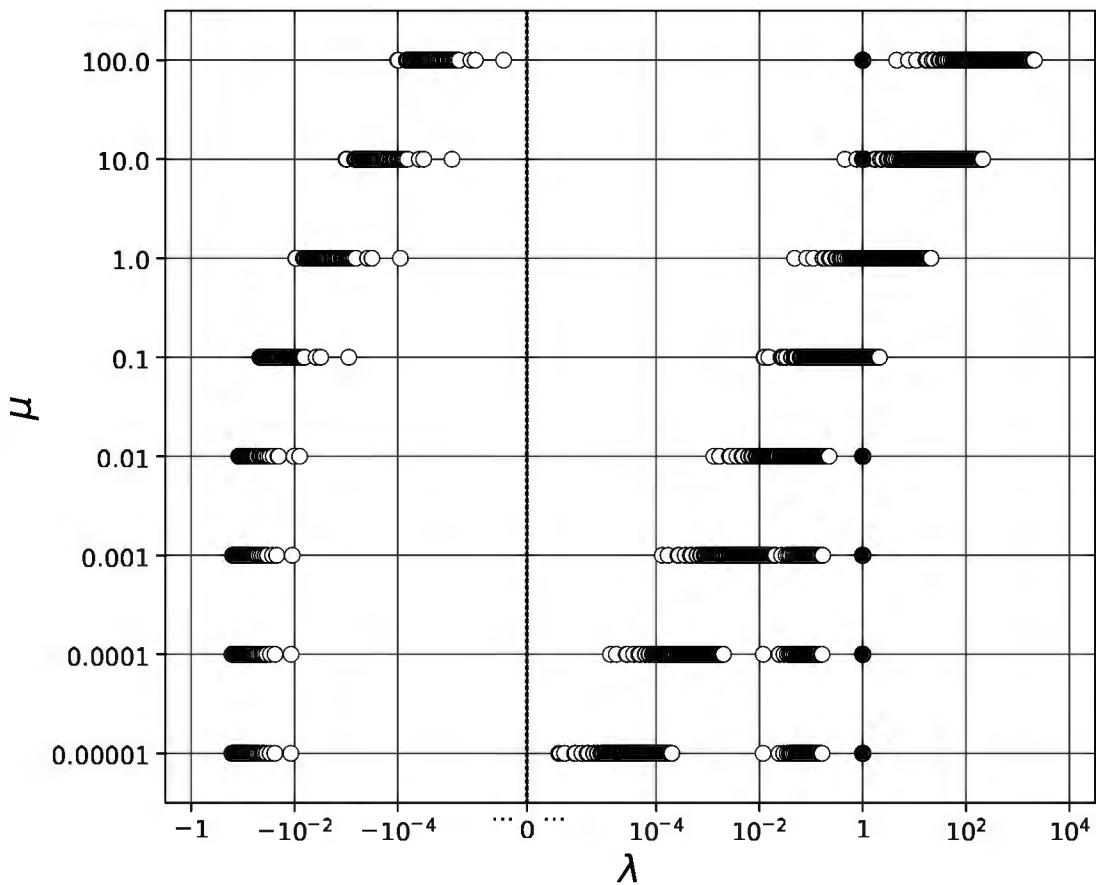
Here

$$S = -BA^{-1}B^\top \in \mathbb{R}^{n_p \times n_p} \quad (14.28)$$

is called the *Schur complement* [64] of the block  $A$  in  $K$ . One can show (Exercise 14.10) that  $-S$  is SPD because  $A$  is SPD and  $B$  has full rank. Factorization (14.27) says that  $K$  is *congruent* to the block-diagonal matrix  $\tilde{K} = \begin{bmatrix} A & 0 \\ 0 & S \end{bmatrix}$ , i.e., there is an invertible matrix  $X$  so that  $K = X\tilde{K}X^\top$ . Sylvester's law of inertia [49, 64] asserts that congruent symmetric matrices have the same number of negative, zero, and positive eigenvalues. Since  $\tilde{K}$  has  $n_u$  positive eigenvalues from  $A$  and  $n_p$  negative eigenvalues from  $S$ , the sketch is qualitatively correct.

The size of the gap around zero in  $\sigma(K)$  will grow as  $\mu$  increases, but the gap also depends on the domain  $\Omega$  and the mesh resolution  $h > 0$ . The condition number  $\kappa(K)$  will also grow as  $h \rightarrow 0$  because the mesh supports higher-frequency velocity modes, and computational results for the -nobase problem confirm this understanding. Figure 14.4, which has (signed) logarithmic scaling on the horizontal axis, comes from numerical computations using a  $P_2 \times P_1$  mixed FE method on a coarse  $9 \times 9$  grid. Note that, because of how Firedrake implements Dirichlet boundary conditions, a high-multiplicity  $\lambda = 1$  eigenvalue always appears in the spectrum of  $K$  (solid dots), independently of  $\mu$ .

The figure shows how the eigenvalues move to the right as the viscosity  $\mu$  increases from  $10^{-5}$  to  $10^2$ . For small  $\mu$  the spectrum of  $K$  is a balanced set similar to  $\sigma(K_B)$ , but with an additional cluster of small positive eigenvalues. For large  $\mu$  the spectrum becomes a set more like  $\sigma(K_A)$  but with small-magnitude negative eigenvalues instead of a null space. As  $\mu$  goes from  $10^{-5}$  to  $10^0$  the condition number  $\kappa(K)$  decreases from  $O(10^6)$  to its smallest values just below  $O(10^3)$  when  $\mu \approx 0.01$ , and it increases again to  $O(10^9)$  for the stiff fluid with  $\mu = 10^2$ .



**Figure 14.4.** The computed eigenvalues  $\lambda$  of  $K$  depend on the viscosity  $\mu$ .

## Preconditioning a symmetric, indefinite, and block system

Preconditioners for the discrete Stokes equations must be adapted to the structure and spectrum of  $K = \begin{bmatrix} A & B^\top \\ B & 0 \end{bmatrix}$ . Because  $K$  is symmetric and indefinite, with both negative and positive eigenvalues, MINRES [49, 66] is a natural choice, but then the preconditioning matrix will need to be SPD because symmetrical preconditioning requires it (Exercise 14.9 and Chapter 2). Alternatively we may disregard symmetry and use GMRES.

All of our approaches will be based on the block structure of  $K$ , but they will also exploit existing preconditioners for the velocity-velocity Laplacian block  $A$ . For that block we have excellent multigrid preconditioners, either GMG (Chapters 6 and 7) or AMG (Chapter 10).

Recall that for the biharmonic problem at the end of Chapter 7, block preconditioners, using `-pc_type fieldsplit` and `-pc_fieldsplit_type additive`, applied a single GMG V-cycle to invert the diagonal blocks and led to an optimal solution. However, *all* diagonal blocks were invertible Laplacians in that example.

We now embark on three routes, seeing where they lead.

**Route 1.** If only we could invert the diagonal blocks of  $K$ , but the lower-right pressure-pressure block is zero! So, suppose we change that block. This is done by so-called *stabilized mixed* FE methods [49]; see Exercise 14.10. However, we will only do computations with stable mixed methods here because the following routes, for preconditioning  $K$  itself, will yield effective solvers.

**Route 2.** One preconditioning “extreme” (Chapter 2) is to fully invert  $K$ , and at least this can be done in a blockwise manner. Gauss-Jordan elimination, using the Schur complement  $S = -BA^{-1}B^\top$  as in (14.27), takes the following form based on two block-diagonal and two block-triangular factors:

$$\underbrace{\begin{bmatrix} I & -A^{-1}B^\top \\ 0 & I \end{bmatrix}}_{\text{solve for } \mathbf{u}} \underbrace{\begin{bmatrix} I & 0 \\ 0 & S^{-1} \end{bmatrix}}_{\text{solve for } p} \underbrace{\begin{bmatrix} I & 0 \\ -B & I \end{bmatrix}}_{\substack{\text{eliminate } \mathbf{u} \\ \text{from } p \text{ eq.}}} \underbrace{\begin{bmatrix} A^{-1} & 0 \\ 0 & I \end{bmatrix}}_{\text{invert } A \text{ block}} \begin{bmatrix} A & B^\top \\ B & 0 \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}.$$

(The reader should check this calculation. See Exercises 14.10 and 14.11.)

If  $A$  and  $S$  are invertible then multiplying-out the factors yields

$$K^{-1} = \begin{bmatrix} A^{-1}(I + B^\top S^{-1}BA^{-1}) & -A^{-1}B^\top S^{-1} \\ -S^{-1}BA^{-1} & S^{-1} \end{bmatrix}. \quad (14.29)$$

Note that  $(S^{-1}BA^{-1})^\top = A^{-1}B^\top S^{-1}$  and thus  $K^{-1}$  is symmetric as expected.

Of course, actually assembling  $K^{-1}$  or  $A^{-1}$  should not be done, because it would cause a complete loss of sparsity, and anyway the plan is to replace the action of  $A^{-1}$  by an approximation, a multigrid preconditioner. However, we must find a way to apply  $S^{-1}$  in some manner. Noting that  $S$  is smaller than  $A$ , i.e.,  $n_p < n_u$ , so assembling  $S^{-1}$  is a credible approach, but we will find that it is not necessary.

Dropping terms from the above product constructs preconditioners. Noting that the block-triangular factors have unit diagonal, suppose we drop those factors. An approximation remains,

$$K^{-1} \approx \begin{bmatrix} I & 0 \\ 0 & S^{-1} \end{bmatrix} \begin{bmatrix} A^{-1} & 0 \\ 0 & I \end{bmatrix} = \begin{bmatrix} A^{-1} & 0 \\ 0 & S^{-1} \end{bmatrix}, \quad (14.30)$$

and in fact it matches the “invert the diagonal” intent in Route 1. That is, we may try the matrix

$$M_D = \begin{bmatrix} A & 0 \\ 0 & -S \end{bmatrix} \quad (14.31)$$

as the preconditioning material (Chapter 2) for a MINRES calculation, for example. We have also changed the sign on the lower-right block because we must supply a SPD preconditioner. It turns out that this will work well because the spectrum of  $M_D^{-1}K$  is known; see below.

Alternatively, one might sacrifice symmetry with a lower-triangular approximation

$$K^{-1} \approx \begin{bmatrix} I & 0 \\ 0 & S^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -B & I \end{bmatrix} \begin{bmatrix} A^{-1} & 0 \\ 0 & I \end{bmatrix} = \begin{bmatrix} A^{-1} & 0 \\ -S^{-1}BA^{-1} & S^{-1} \end{bmatrix} \quad (14.32)$$

corresponding to preconditioning-material matrix

$$M_L = \begin{bmatrix} A & 0 \\ B & S \end{bmatrix}. \quad (14.33)$$

Applying  $M_L^{-1}$  has nearly the same computation cost as  $M_D^{-1}$ , the only difference being that we also apply the sparse matrix  $B$ . (The lower right block can either have a flipped sign or not, without significantly affecting performance as a preconditioner [116].)

Applying  $M_D$  or  $M_L$  as a preconditioner requires a fast method for applying  $S^{-1}$ , i.e., for solving  $Sp = r$ . That is, we must still figure out how to approximately invert, i.e., precondition, the Schur complement block  $S$  while being efficient in flops and memory.

**Route 3.** Success in preconditioning  $K$  using any matrix  $M$  will depend on the spectrum of

$$T = M^{-1}K. \quad (14.34)$$

Ever since Chapter 2 we have been thinking that the goal is to choose  $M$  so that  $\sigma(T)$  is clustered tightly around one, that is, so that  $M^{-1} \approx K^{-1}$ . However, we should remember how norm-minimizing Krylov methods actually work. What we need for GMRES, for example, is only that  $\sigma(T)$  be clustered near a few nonzero locations in the complex plane, relatively separated from zero, thus that low-degree polynomials exist which are small on  $\sigma(T)$ . The same applies for MINRES, but on the real line.

Consider the eigenvalues of  $T = M^{-1}K$  if  $M$  is a real SPD diagonal matrix, but in the simplest  $2 \times 2$  case. That is, suppose  $K = \begin{bmatrix} a & b \\ b & 0 \end{bmatrix}$  and  $M = \begin{bmatrix} a & 0 \\ 0 & x \end{bmatrix}$  where  $a > 0$ ,  $b \neq 0$ , and  $x > 0$ . Note there is no way to choose  $x$  to bring  $T$  close to the identity,

$$T = \begin{bmatrix} 1 & a^{-1}b \\ x^{-1}b & 0 \end{bmatrix},$$

and  $\det(T) = -a^{-1}x^{-1}b^2$  is negative anyway. If, however, we normalize this value to  $-1$ , i.e. we make  $\det(T) = -1$  by choosing  $x = a^{-1}b^2$ , then the characteristic equation of  $T$  is  $\lambda^2 - \lambda - 1 = 0$  so the eigenvalues are  $\lambda = \frac{1}{2}(1 \pm \sqrt{5})$ .

The reader who has not already forgotten Route 2 might also notice that the formula  $x = a^{-1}b^2$  generalizes to  $X = BA^{-1}B^\top = -S$ . However, in the discrete Stokes equations  $B \approx -\nabla \cdot$  is not invertible, and this generates a known eigenvalue of  $T = M^{-1}K$ . In fact, suppose  $\mathbf{u} \in \text{null}(B)$  and consider a block-diagonal preconditioner  $M = \begin{bmatrix} A & 0 \\ 0 & X \end{bmatrix}$ . Then

$$T \begin{bmatrix} \mathbf{u} \\ 0 \end{bmatrix} = \begin{bmatrix} A^{-1} & 0 \\ 0 & X^{-1} \end{bmatrix} \begin{bmatrix} A & B^\top \\ B & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{u} \\ 0 \end{bmatrix}.$$

Thus  $T = M^{-1}K$  has an eigenvalue  $\lambda = 1$ . The multiplicity of  $\lambda = 1$  is at least as large as the null space of  $B$ .

Now, the remarkably recent observation made in [116] is that the informal reasoning in the last two paragraphs, which yields three real eigenvalues of  $T$ , represents the general case for Schur-complement preconditioning using  $M_D$ . See Exercise 14.12 for the proof of the following theorem.

**Theorem 14.2.** Assume  $K$  in (14.24) is invertible, let  $S = -BA^{-1}B^\top$ , and consider  $M_D$  in (14.31). Then  $T = M_D^{-1}K$  satisfies

$$(T - 1)(T^2 - T - 1) = 0,$$

and thus  $T$  has at most three distinct eigenvalues, namely  $\lambda = 1$  and  $\lambda = \frac{1}{2}(1 \pm \sqrt{5})$ .

Thus if we were to use the exact matrix  $M_D$  as a preconditioner then MINRES would solve our discrete Stokes problem in at most three iterations. A variant of the argument [116] shows that GMRES needs at most two iterations using  $M_L$  from (14.33) for preconditioning. These are well-known instances where an effective preconditioning matrix exists whose inverse is *not* an approximate inverse of the system matrix.

At this point, having seen that the Schur complement  $S$  arises in different ways, including factorization (14.27), we are motivated to consider its spectral properties. Preconditioning the Stokes equations requires finding an inexpensive approximation of  $S^{-1}$ .

Fact 21. The Stokes equations have a zero diagonal block which requires attention. *The equations arise as an equality-constrained minimization problem, so a zero block appears in a mixed FE method. One must deal with this in some manner. For stable elements, Schur decompositions via fieldsplit still need a preconditioner for the Schur block.*

## Stable elements and Schur complements

The matrix  $S = -BA^{-1}B^\top$  should be a well-behaved operator with a bounded inverse. Its factors approximate differential operators and their inverses, and they multiply together in a way that suggests cancellation, as follows. When  $\mu > 0$  is constant,  $A \approx -\mu\nabla^2$  and  $B \approx -\nabla\cdot$ , so  $BB^\top \approx -\nabla^2$  also, and, ignoring issues of commutativity, it is reasonable to conclude

$$S \approx -(-\nabla\cdot)(-\mu\nabla^2)^{-1}\nabla \approx -\mu^{-1}. \quad (14.35)$$

(Recall that we are assuming that both  $\partial_D\Omega$  and  $\partial_N\Omega$  have positive measure, thus that the problem is well posed, and that  $A$  is invertible.) That is,  $S$  wants to be a constant (diagonal) operator with a single negative eigenvalue. Of course this argument is far too casual, but it correctly predicts that the eigenvalues of  $S$  are clustered near  $-\mu^{-1}$  as long as the mixed FE spaces are chosen in a good way. This choice of FE spaces, and its effect on the invertibility of the Schur complement, is now the topic.

Our first step is nontrivial. The bilinear form  $b(\mathbf{v}, q) = -\int_\Omega q \nabla \cdot \mathbf{v}$  satisfies a nondegeneracy condition, the *inf-sup inequality*, over the Hilbert spaces  $\mathcal{X}_0 = (H_0^1)^2$  and  $L^2(\Omega)$ . Namely, there exists  $\gamma > 0$  so that

$$\inf_{q \neq 0} \sup_{\mathbf{v} \neq 0} \frac{b(\mathbf{v}, q)}{\|\mathbf{v}\|_{\mathcal{X}_0} \|q\|_{L^2}} \geq \gamma. \quad (14.36)$$

Standard references prove this inequality and describe its role in proving the well-posedness of saddle-point problem (14.18) [19, 24, 49]. Exercise 14.13 shows that a weaker kind of nondegeneracy, namely that  $b(v, q) = 0$  for all  $v$  implies  $q = 0$ , follows from (14.36).

Inequality (14.36) plays at least two roles in numerical computations. First, if it holds for FE spaces  $\mathcal{V}^h$  and  $\mathcal{W}^h$ , and if  $\gamma$  is independent of the mesh spacing parameter  $h > 0$ , then the mixed FE method converges. In fact, the following theorem ([137, Theorem 2.1]; see also [49, Theorem 3.1]) applies if the solution is smooth. In such an *a priori* estimate the exact-solution norms are fixed independent of any aspect of the numerical method, and the theorem provides an expected rate of convergence as  $h \rightarrow 0$ . We will confirm these rates in actual computations using the -analytical problem case.

**Theorem 14.3.** Suppose  $\mathbf{u}$  and  $p$  solve weak form (14.18) and are smooth. Let  $\mathcal{V}^h \subset \mathcal{X}_0$  be a  $P_k$  or  $Q_k$  FE space and  $\mathcal{W}^h \subset L^2(\Omega)$  be a  $P_\ell$  or  $Q_\ell$  space. Suppose  $\mathbf{u}^h \in \mathcal{V}^h$  and  $p^h \in \mathcal{W}^h$  solve the discrete Stokes equations (14.23). If  $\gamma > 0$  exists, independent of  $h$ , so that (14.36) holds for all  $\mathbf{v} \in \mathcal{V}^h$  and  $q \in \mathcal{W}^h$ , then there are positive constants, independent of  $h$ , so that

$$\|\mathbf{u} - \mathbf{u}^h\|_{\mathcal{X}_0} + \|p - p^h\|_{L^2} \leq C_1 h^k \|\mathbf{u}\|_{H^{k+1}} + C_2 h^{\ell+1} \|p\|_{H^{\ell+1}}. \quad (14.37)$$

Moreover, if the domain  $\Omega$  is convex then there are positive constants so that

$$\|\mathbf{u} - \mathbf{u}^h\|_{(L^2)^2} \leq C_3 h^{k+1} \|\mathbf{u}\|_{H^{k+1}} + C_4 h^{\ell+2} \|p\|_{H^{\ell+1}}. \quad (14.38)$$

The inf-sup inequality (14.36), while abstract, is also relevant to the construction of Schur complement preconditioning methods. In fact, the constant  $\gamma$  measures the norm of the square root of the inverse of  $S$ . To make this connection precise, suppose that in (14.36) we integrate by parts in  $b(q, \mathbf{v})$  and expand the definitions of inf and sup. Thus for all  $q \in L^2$  there exists  $\mathbf{v} \in \mathcal{X}_0$  so that

$$(\mathbf{v}, \nabla q) \geq \gamma \|\mathbf{v}\|_{\mathcal{X}_0} \|q\|_{L^2}. \quad (14.39)$$

(The left side now uses the  $(L^2)^2$  inner product  $(\mathbf{v}, \mathbf{w}) = \int_\Omega \mathbf{v} \cdot \mathbf{w}$ .) Now let  $E$  be the square root of the positive vector Laplacian  $-\mu\nabla^2$ . (There is a unique self-adjoint, positive, and unbounded operator  $E$  so that  $E^2 = -\mu\nabla^2$  [127].) Furthermore,  $a(\mathbf{v}, \mathbf{w}) = (E\mathbf{v}, E\mathbf{w})$  for all  $\mathbf{v}, \mathbf{w} \in \mathcal{X}_0$ ,

so  $(E \cdot, E \cdot)$  is equivalent to the ordinary inner product on  $\mathcal{X}_0 = (H_0^1)^2$ . In (14.39) define  $\mathbf{z} = E\mathbf{v}$  to get  $\mathbf{z} \in (L^2)^2$ . Thus there is  $\Gamma > 0$  such that for all  $q \in L^2$  there exists a  $\mathbf{z} \in (L^2)^2$  so that

$$(\mathbf{z}, E^{-1}\nabla q) \geq \Gamma \|\mathbf{z}\|_{(L^2)^2} \|q\|_{L^2}. \quad (14.40)$$

Equivalently, inequality (14.40) says that for all  $q \in L^2$  there exists a unit-length  $\mathbf{z} \in (L^2)^2$  so that  $(\mathbf{z}, E^{-1}\nabla q) \geq \Gamma \|q\|_{L^2}$ . This is really just a lower bound on the norm of  $E^{-1}\nabla q$ ,

$$\inf_{q \neq 0} \frac{\|E^{-1}\nabla q\|_{(L^2)^2}}{\|q\|_{L^2}} \geq \Gamma_1, \quad (14.41)$$

where  $\Gamma_1 \geq \Gamma > 0$ . What we have shown in (14.41) is that the inf-sup inequality is equivalent to a lower bound on a certain operator, which thus has no kernel:

$$Z = E^{-1}\nabla : L^2 \rightarrow (L^2)^2 \quad \text{is bounded below by } \Gamma_1 > 0.$$

It is a small step to square the operator and observe that  $Z^\top Z : L^2 \rightarrow L^2$  is bounded below by  $\Gamma_1^2 > 0$ . But now we have returned to heuristic (14.35),

$$Z^\top Z = \nabla^\top E^{-1} E^{-1} \nabla = (-\nabla \cdot)(-\mu \nabla^2)^{-1} \nabla,$$

that is, the operator  $Z^\top Z$ , which has a lower bound because the inf-sup condition holds, is essentially the Schur complement up to a sign:  $S \approx -Z^\top Z$ .

From now on we include a mesh parameter  $h > 0$  in our notation:

$$S_h = -B_h A_h^{-1} B_h^\top. \quad (14.42)$$

Uniform invertibility of the Schur complement  $S_h$ , under mesh refinement, is important to algorithmic scalability. Specifically, in using block preconditioners  $M_D$  (14.30) or  $M_L$  (14.32), or approximations thereof, control on  $\|S_h^{-1}\|$  is needed to get well-clustered spectrum of the preconditioned system matrix. However, for some choices of mixed FE spaces  $\mathcal{V}^h \times \mathcal{W}^h$  this is not guaranteed [19, 24, 49]. There are two possible failure modes:

- (i)  $B_h^\top$  could have a nontrivial null space, or
- (ii)  $S_h^{-1}$  might exist but the condition numbers  $\kappa(S_h)$  might grow fast as  $h \rightarrow 0$ .

In the latter case convergence is at risk.

We will see that in fact failure mode (i) occurs for certain well-known unstable mixed FE choices, but otherwise the estimated condition numbers  $\kappa(S_h)$  are modest, and so we expect convergence when using such “stable” mixed FE choices in computations. In fact, consider the runs

```
(firedrake) $ ./stokes.py -nobase -s_ksp_type minres -s_pc_type none \
    -s_mat_type aij -s_ksp_view_mat binary:FILE.dat \
    -s_ksp_rtol 1.0 MIXED -refine LEV
```

Here **MIXED** corresponds to the choices of elements in the first column of Table 14.1. These runs simply assemble the  $K_h$  matrix over a uniform mesh of triangles and save it to **FILE.dat**; from the option **-s\_ksp\_rtol 1.0** the KSP “succeeds” (and stops) after one iteration. We use only coarse grids **LEV=1, 2, 3, 4** because, in producing the table, we compute  $S_h$  and  $\kappa(S_h)$  by direct linear algebra using the definition (14.42), and such computations clearly do not scale as  $h \rightarrow 0$ . The table shows that the first three mixed FE methods, which we expect from the literature to be stable [49, 107], yield uniformly invertible Schur complements  $S_h$ . The last two mixed FE methods, expected to be unstable, in fact exhibit failure mode (i).

**Table 14.1.** Above the line: 2-norm condition numbers  $\kappa(S_h)$ . Below the line: the dimension of  $\text{null}(B_h^\top)$ , i.e., the rank deficiency of  $B_h$ .

|                  | $5 \times 5$ | $9 \times 9$ | $17 \times 17$ | $33 \times 33$ |
|------------------|--------------|--------------|----------------|----------------|
| $P_2 \times P_1$ | 75.5         | 113          | 138            | 154            |
| $Q_2 \times Q_1$ | 33.9         | 49.3         | 60.4           | 67.6           |
| $P_2 \times P_0$ | 5.63         | 6.61         | 7.08           | 7.27           |
| $P_1 \times P_1$ | (4)          | (4)          | (4)            | (4)            |
| $P_1 \times P_0$ | (8)          | (16)         | (32)           | (64)           |

## Options for Schur+GMG preconditioners

How do these Schur-complement preconditioning ideas translate into PETSC solver options? Note that the solver parameter space is now enormous. It combines choices for FE spaces, solver options associated to the block structure of  $K$ , plus all the options for the inversion of the diagonal blocks, namely for preconditioners for  $A$  and  $S$ . Preconditioners for  $A$  include the full space of multigrid parameters (Chapters 6 and 7), but preconditioners for  $S$  require new choices which are covered below. We must limit the choices just to have a sane testing strategy.

In all cases, we will combine Schur-complement block-structured preconditioning with a single GMG V-cycle on the velocity block  $A$ . Thus the following common options apply to all the runs below:

```
-s_pc_type fieldsplit -s_pc_fieldsplit_type schur \
-s_fieldsplit_0_ksp_type preonly -s_fieldsplit_0_pc_type mg \
-s_fieldsplit_1_ksp_type preonly
```

Choosing `fieldsplit` of type `schur` is more or less required for solving a stable mixed FE method because the lower-right block in  $K$  is identically zero. For example, the `additive` type of `fieldsplit`, used for the biharmonic equation in Chapter 7, would not work.

The choice of GMG is, of course, based on our results for the Poisson equation (Chapter 6). That is, a single GMG V-cycle is an  $O(N)$  operation with excellent spectral properties. Despite the large parameter space for the GMG method, we use only the default settings. (See the exploration of GMG options in Chapter 6, and Exercise 14.15 for comparison to AMG.)

However, we will explore several variations on Schur preconditioning, so the code `stokes.py` packages options into two bundles. (See the code itself for how a Python dictionary is used to define the bundles.) The first bundle is `-schurgmg`, which chooses the block form for the preconditioning “material”  $M$ :

- `-schurgmg diag` uses  $M = M_D$  from (14.31),
- `-schurgmg lower` uses  $M = M_L$  from (14.33), and
- `-schurgmg full` uses  $M = K$ .

Thus a `-schurgmg` bundle includes a Schur factorization choice,

```
-s_pc_fieldsplit_schur_fact_type diag|lower|full
```

plus all of the above common options. Note that MINRES is a natural KSP choice for `diag`, but GMRES works well in all cases and is obligatory for the `lower` and `full` choices; the user sets `-s_ksp_type` accordingly. For type `diag` a bundled sign-flip option generates an approximation to  $M_D$  in (14.31):

```
-s_pc_fieldsplit_schur_scale -1.0
```

The second bundle, option `-schurpre`, addresses how the Schur complement  $S_h$  is approximated, that is, how it is applied as a preconditioner  $\hat{S}_h^{-1}$ . Here we are guided by which schemes are available in PETSC, and by the literature, to propose two choices:

- `-schurpre selfp` specifies an approximation to  $S_h$ :

$$\hat{S}_h = D_{\tilde{S}_h} \text{ where } \tilde{S}_h = -B_h(D_{A_h})^{-1}B_h^\top \in \mathbb{R}^{n_p \times n_p}, \quad (14.43)$$

where  $D_A$  is the diagonal of  $A$ . Note that  $\tilde{S}_h$  could be assembled into a relatively small matrix which retains some sparsity, but in fact  $\hat{S}_h$  is only its diagonal, which is even less expensive. The PETSC options in this bundle are

```
-s_pc_fieldsplit_schur_precondition selfp \
-s_fieldsplit_1_pc_type jacobi
```

- `-schurpre mass` uses the sparse pressure-space *mass matrix*, defined using a basis of hat functions  $\psi_j$  for  $\mathcal{W}^h$ :

$$Q_{ij} = \int_{\Omega} \psi_i \psi_j. \quad (14.44)$$

In a stable mixed FE scheme  $\mu^{-1}Q$  is known to be spectrally equivalent to  $-S_h$  [49, Theorem 3.22], though with wide bounds. The assembled mass matrix is approximately inverted using the ICC method. To implement this one defines a Firedrake “auxiliary operator” [92] which computes  $\mu^{-1}Q$ . That is, `stokes.py` defines the following Python class containing the UFL weak form for (14.44):

```
class Mass(AuxiliaryOperatorPC):

    def form(self, pc, test, trial):
        a = (1.0/mu) * inner(test, trial)*dx
        bcs = None
        return (a, bcs)
```

In PETSC’s view, the resulting approximation  $\hat{S}_h$  is formed starting from the zero pressure-pressure block in  $K$ , i.e., “ $A_{11}$ ” in [10], so the options are

```
-s_pc_fieldsplit_schur_precondition a11 \
-s_fieldsplit_1_pc_type python \
-s_fieldsplit_1_pc_python_type __main__.Mass \
-s_fieldsplit_1_aux_pc_type bjacobi \
-s_fieldsplit_1_aux_sub_pc_type icc
```

In the next section we will test the algorithmic scaling of these solvers, but we can quickly see some of the most promising combinations. First we consider solutions of the lid-driven cavity on a fine uniform grid of  $1024 \times 1024$  quadrilaterals, using  $Q_2 \times Q_1$  Taylor-Hood elements and  $N = 9.4 \times 10^6$  degrees of freedom:

```
| (firedrake) $ ./stokes.py -quad -refine 9 -s_ksp_type W \
| -schurmgm X -schurpre Y
```

We measure iterations (`-s_ksp_converged_reason`) and the flops (`-log_view`) for all six cases  $X = \text{diag}|\text{lower}|\text{full}$  and  $Y = \text{selfp}|\text{mass}$ . We use  $W = \text{minres}$  for the `diag` cases, but otherwise  $W = \text{gmres}$ .

The results in Table 14.2 show that the `full` block structure is more expensive in a flops-per-iteration sense, and it does not yield a reduction in iterations versus `lower`. This one comparison

**Table 14.2.** KSP iterations and total flops (times  $10^9$ ) for solutions of the default lid-driven cavity problem on a  $1024 \times 1024$  grid of quadrilaterals and  $Q_2 \times Q_1$  elements. The boxed cases reappear in Figures 14.7 and 14.8.

| X \ Y | selfp |     | mass |     |
|-------|-------|-----|------|-----|
| diag  | 32    | 238 | 19   | 151 |
| lower | 17    | 143 | 22   | 179 |
| full  | 17    | 251 | 22   | 318 |

**Table 14.3.** The same measurements on the same solvers as in Table 14.2, but for a highly nonuniform triangulation and  $P_2 \times P_1$  elements.

| X \ Y | selfp |      | mass |      |
|-------|-------|------|------|------|
| diag  | 30    | 55.6 | 21   | 40.7 |
| lower | 16    | 34.0 | 26   | 52.8 |
| full  | 15    | 55.6 | 26   | 92.5 |

is far from definitive, but we also consider a nonuniform mesh, shown in Figure 14.9 below, and with five levels of refinement. Using  $P_2 \times P_1$  elements yields  $N = 3.2 \times 10^6$  degrees of freedom. Measuring the same quantities as before, the results in Table 14.3 confirm our conclusions. There is no reason to use a full Schur block structure, and indeed the `diag+mass` and `lower+selfp` solvers perform best. These tables explain why we will treat the solver options

```
-s_ksp_type gmres -schurgmg lower -schurpre selfp
```

as the defaults from now on.

## Convergence and solver performance

As the reader has surely come to expect, we will now test `stokes.py` for convergence, algorithmic scaling (optimality), and parallel scaling.

The rate of convergence will depend on the (mixed) element type, but the theoretical rates are mostly known. Based on the literature of mixed methods [24, 49], and the above results on the norms  $\|S_h^{-1}\|$ , we will only use the following stable element types [107]:

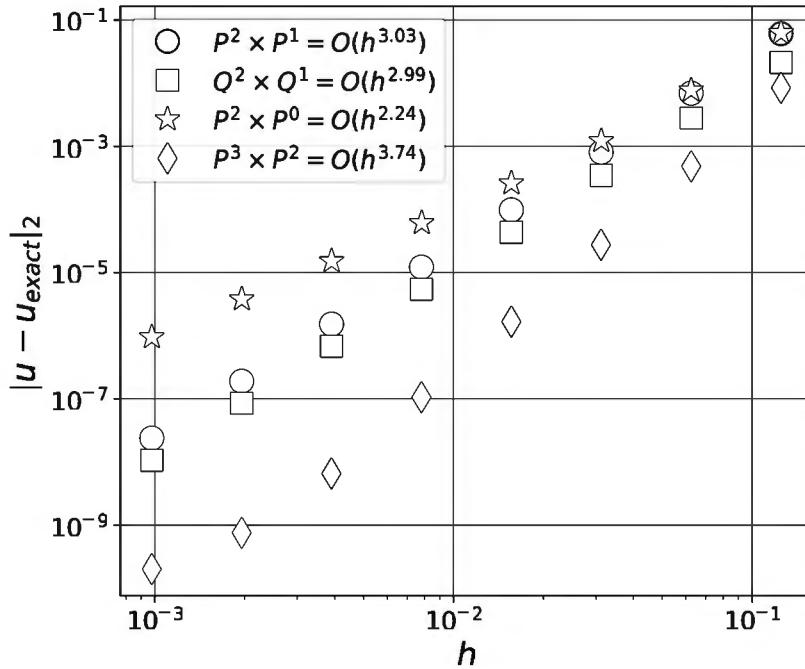
$P_2 \times P_1$ : The default-degree Taylor-Hood element on triangles.

$Q_2 \times Q_1$ : The Taylor-Hood element on quadrilaterals: `-quad`.

$P_2 \times P_0$ : “Continuous-discontinuous” (CD) elements on triangles, with piecewise-constants for pressure: `-udegree 2 -pdegree 0 -dp`.

$P_3 \times P_2$ : Higher-degree Taylor-Hood on triangles: `-udegree 3 -pdegree 2`.

We use uniform meshes, starting from a  $9 \times 9$  mesh, through seven levels of refinement to one with  $1025 \times 1025$  nodes. For  $P_2 \times P_1$ ,  $Q_2 \times Q_1$ , and  $P_2 \times P_0$  elements the finest mesh yields  $N \approx 10^7$  degrees of freedom, and double that for  $P_3 \times P_2$  elements. When measuring



**Figure 14.5.**  $L^2$ -norm convergence of velocity  $\mathbf{u}$  using stable mixed elements.

convergence we choose the following options wherein MIXED corresponds to the choices above:

```
(firedrake) $ ./stokes.py -analytical -refine LEV -s_ksp_rtol 1.0e-8 \
-s_ksp_type gmres -schurmgm lower -schurpre selfp MIXED
```

The smooth `-analytical` exact solution is suitable for testing convergence, but note that correct evaluation of the numerical error requires a high-degree FE interpolant of the exact solution. (This is needed because in a (mixed) space  $\mathcal{V}^h \times \mathcal{W}^h$  the FE solution is actually closer to the exact solution than the interpolant of the exact solution, a fundamental property of conforming FE schemes [49].) Thus we interpolate the exact solution in  $P_{k+2} \times P_{k+1}$  for a  $P_k \times P_{k-1}$  computation, for example. Also, we only consider  $h$ -refinement even though  $p$ -refinement, i.e., increasing  $k$ , would be effective. The reader is encouraged to experiment with other element types and refinement modes.

Figure 14.5 shows that  $L^2$  velocity errors for the Taylor-Hood elements ( $P_k \times P_{k-1}$  and  $Q_k \times Q_{k-1}$ ) go to zero at  $O(h^{k+1})$  rates while pressure errors decay at  $O(h^k)$  (Figure 14.6), exactly as expected from (14.38) in Theorem 14.3. The  $P_2 \times P_0$  (CD) method has slower convergence, even for velocity, because of “corruption” from the low-degree pressure approximation. The higher-order Taylor-Hood elements converge most rapidly as expected.

Before measuring performance, recall the basic ideas of optimal algorithmic scaling. The KSP repeatedly applies  $T = M^{-1}K$ , where  $K$  here is the sparse, block matrix in (14.24). We want each preconditioner application  $M^{-1}$  to be a fast  $O(N)$  operation, and we want the resulting number of KSP iterations to be independent of  $N$  and  $h$  (Chapter 7). In order for the latter to hold  $T$  must have clustered spectrum, corresponding to easy-to-generate Krylov polynomials (Chapter 2), for example as addressed in Theorem 14.2. If these goals are achieved then the whole solution method should be  $O(N)$ , i.e., optimal.

We consider the algorithmic scaling of the three boxed solver choices in Table 14.2. Each combines Schur-complement block structure and a single GMG V-cycle on the velocity block. Algorithmic scaling here does not depend strongly on element type so we use the same uniform meshes and  $Q_2 \times Q_1$  elements as in Figures 14.5 and 14.6.

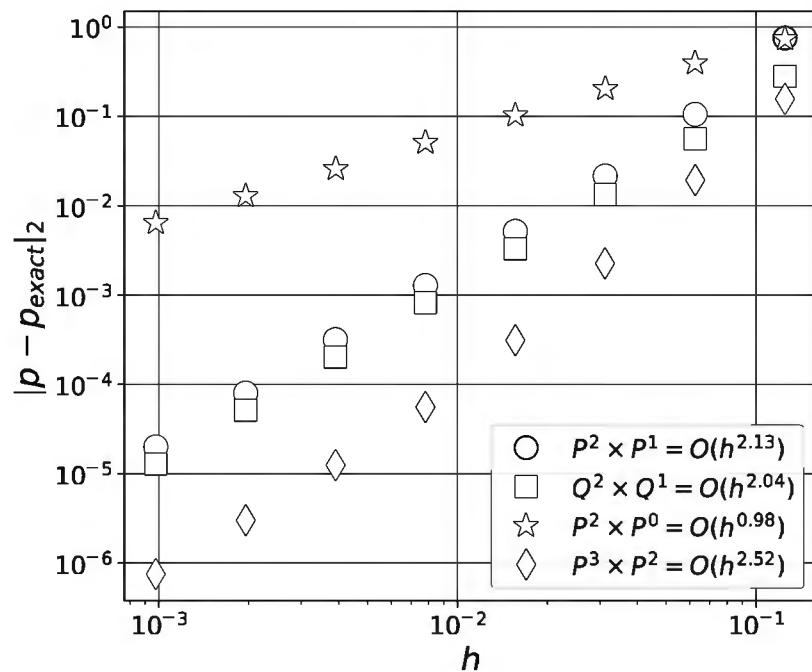


Figure 14.6.  $L^2$ -norm convergence of pressure.

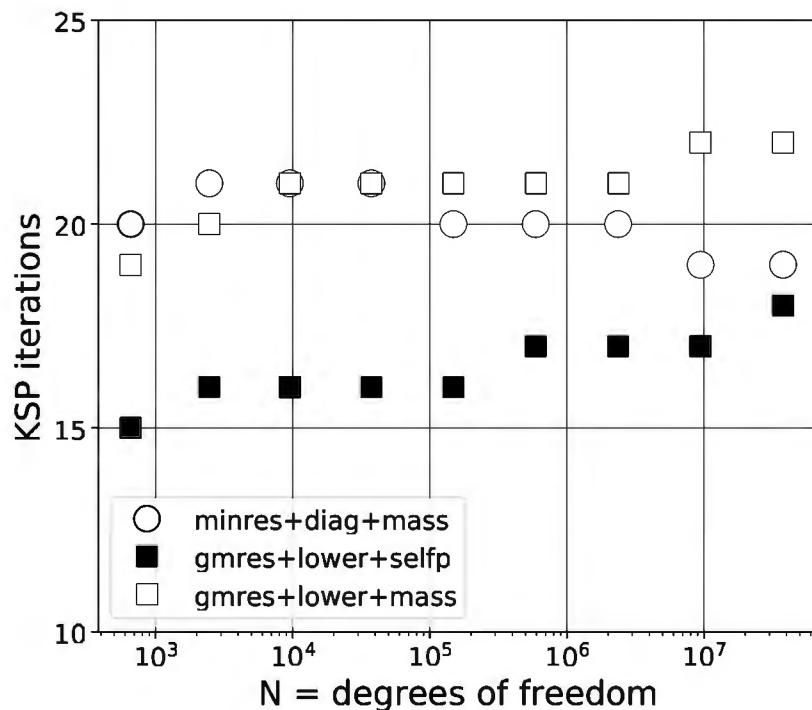


Figure 14.7. Preconditioned Krylov iterations for three Schur-GMG solvers.

Figures 14.7 and 14.8 show the number of iterations and the amount of work per degree of freedom  $N$ , respectively, over a more than four orders-of-magnitude increase in  $N$ . All of the solvers show optimality, as desired, with essentially level graphs. Note that the work is essentially a multiple of the number of KSP iterations, except that preconditioner setup operations, included in the work estimate, are amortized over the iterations. Because of the fixed number of nonzeros

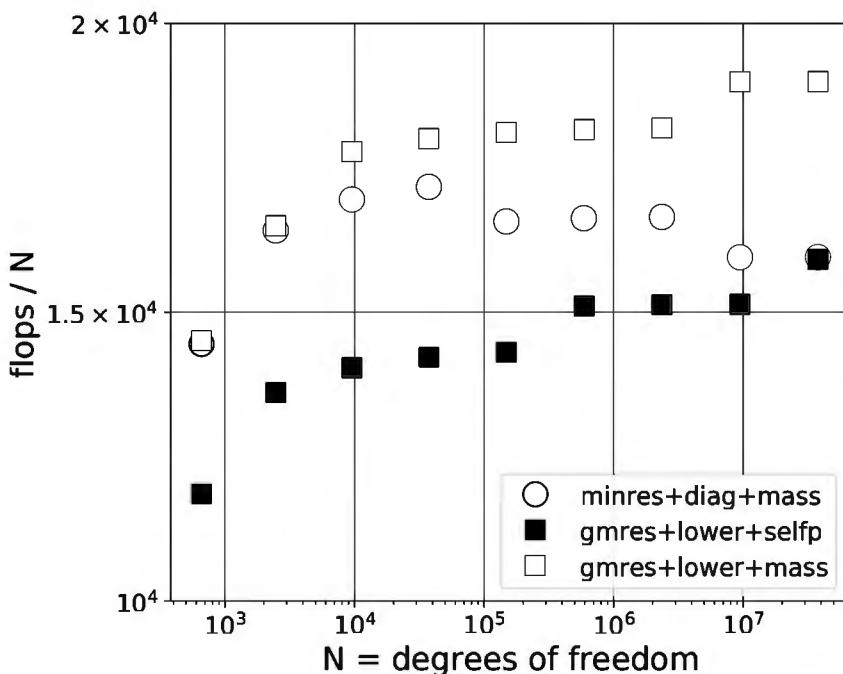


Figure 14.8. Flops per degree of freedom for the same solvers.

per row in  $K$ , the amount of work required to apply  $T = M^{-1}K$  is indeed  $O(N)$  in all cases; this can be confirmed in `-log_view` output.

Finally we look at parallel weak-scaling wherein the number of degrees of freedom per process is fixed (Chapter 8). We use a  $9 \times 9$  coarse mesh:

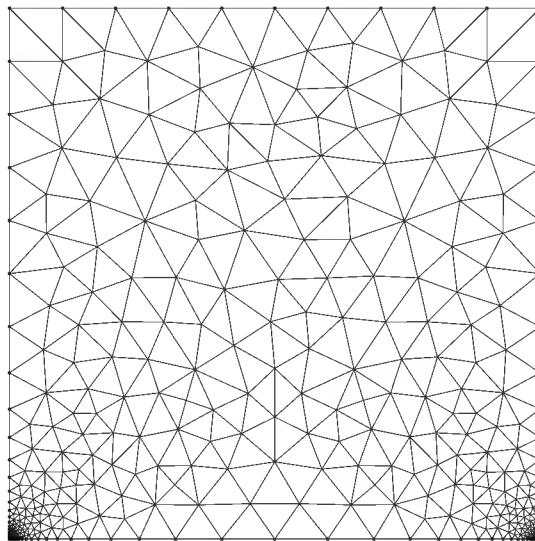
```
(firedrake) $ mpiexec -n P ./stokes.py -quad -mx 9 -my 9 -refine LEV \
-s_ksp_type gmres -schurmgm lower -schurpre selfp
```

Choosing  $P = 1, 4, 16, 64$  and  $\text{LEV} = 5, 6, 7, 8$ , respectively, yields  $257 \times 257$  meshes on each process, with about  $6 \times 10^5$  degrees of freedom.

Table 14.4. Weak scaling for `stokes.py` with GMRES, `-schurmgm lower`, and `-schurpre selfp`.

| $P$ | $N$               | KSP iterations | average flops         |
|-----|-------------------|----------------|-----------------------|
| 1   | $5.9 \times 10^5$ | 17             | $0.89 \times 10^{10}$ |
| 4   | $2.4 \times 10^6$ | 20             | $1.03 \times 10^{10}$ |
| 16  | $9.4 \times 10^6$ | 20             | $1.03 \times 10^{10}$ |
| 64  | $3.8 \times 10^7$ | 21             | $1.08 \times 10^{10}$ |

The results in Table 14.4 show good weak scaling, but a little thought shows this is no surprise. The only significant solver difference between the serial and parallel cases is a difference deep inside GMG, namely that the smoother switches from Gauss-Seidel (GS) to its processor-block version. (Note the coarse grid problem is solved redundantly by LU.) This smoother change, and imperfect algorithmic scaling (Figures 14.7 and 14.8) accounts for the small increase in iterations.



**Figure 14.9.** A graded triangulation suitable for eddy hunting. This 794-element mesh is the multigrid coarse level.

## Moffatt eddies

The solvers we have developed for the Stokes problem are powerful. We show this by revealing small features of the modeled flow in the lid-driven cavity. In theory this flow exhibits infinitely many, exponentially small corner eddies (vortices) [2, 113]. Finding eddies, by exploiting the scalability of the solvers, plus Firedrake’s ability to manage unstructured meshes, demonstrates the effectiveness of our methods.

Consider the triangulation in Figure 14.9. The grading in the corners was generated by a short script `python/ch14/lidbox.py`. It defines a geometry with a variable “characteristic length” along the boundary, varying from 0.1 down to 0.001 in the lower corners. Gmsh uses this value as a size target for the triangulation (Chapter 10).

To run the example, build the mesh as follows:

```
| $ ./lidbox.py graded.geo
| $ gmsh -2 graded.geo      # generates graded.msh
```

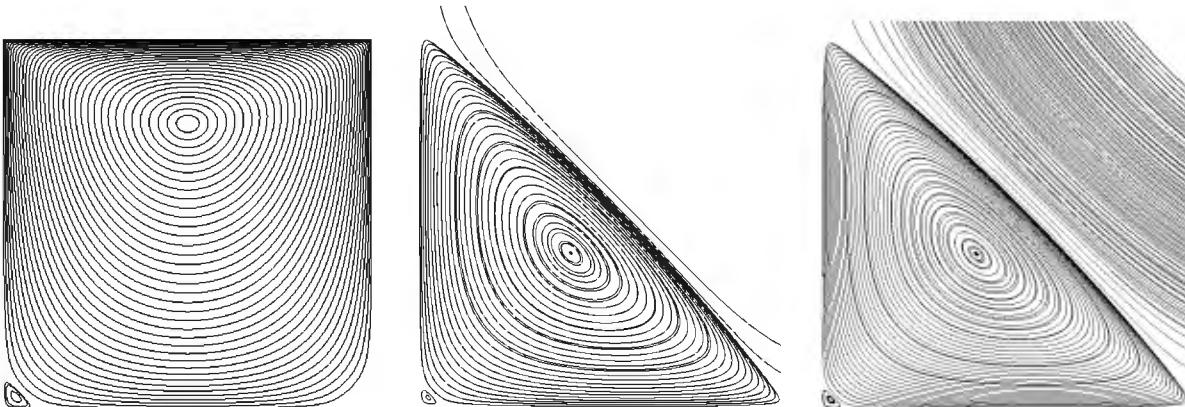
The following command then reads the mesh and refines each triangle five times; each becomes  $4^5 = 1024$  triangles. It then solves the equations using GMRES iteration, the preferred Schur-GMG preconditioner bundle, and a tight convergence tolerance:

```
| (firedrake) $ ./stokes.py -mesh graded.msh -refine 5 \
|           -s_ksp_type gmres -schurmg lower -schurpre selfp \
|           -s_ksp_rtol 1.0e-12 -o lid.pvd
```

Additional options `-showinfo` `-s_ksp_converged_reason` will aid in understanding this run. Note that the mesh has  $8.1 \times 10^5$  elements for  $N = 3.2 \times 10^6$  degrees of freedom.

The GMG preconditioner uses six-level V-cycles to approximately invert the velocity-velocity block, down to the coarse level (the mesh in `graded.msh`) where a direct LU solver is used. The solver needs 39 KSP iterations, less than 8 GB memory, and about two minutes run time. Because `-refine 3,4` levels cause 38 and 39 iterations, respectively, the evidence suggests optimality.

We visualize streamlines from output file `lid.pvd` using Paraview’s “Stream Tracer” functionality. Manual zooming, and reseeding of the Tracer, reveals the first three successive eddies



**Figure 14.10.** Zoomed views of the corner eddies, starting with the full square  $\Omega = (0, 1)^2$  (left) and expanding each time by a factor of more than 10 (middle and right). The third-level eddy appears in the right-hand view.

(Figure 14.10). The flow speed is very low in the corners because of the no-slip ( $\mathbf{u} = 0$ ) condition along the walls, but computed streamlines accurately reveal the flow direction. For the right-angled corners in this experiment, Moffatt's similarity-solution analysis [113] predicts a sequence of eddies with each about 1/16th the size of the previous, thus the resolved third eddy has linear dimensions around  $16^{-3} \approx 2 \times 10^{-4}$ ; see Exercise 14.16. The author was unable to find the fourth eddy in this or the `-refine 6` result.

## Exercises

- 14.1. Prove Lemma 14.1. Then show that (14.7) defines an inner product.
- 14.2. (a) Starting from the definition of  $\nabla \mathbf{u}$ , then by commuting mixed derivatives and using incompressibility, show that  $\nabla \cdot (\nabla \mathbf{u}) = \mathbf{0}$ .  
 (b) On the other hand, confirm that  $\nabla \cdot (\nabla \mathbf{u}^\top) = \langle \nabla^2 u_0, \nabla^2 u_1, \nabla^2 u_2 \rangle$ . This defines the vector Laplacian symbol  $\nabla^2 \mathbf{u}$ .
- 14.3. From the notation in the previous exercise, derive the constant-viscosity strong-form momentum equation (14.21) from (14.5).
- 14.4. The bilinear form defined in (14.15),  $b(\mathbf{v}, q) = -\int_{\Omega} q \nabla \cdot \mathbf{v}$ , acts on distinct infinite-dimensional spaces, thus it becomes a rectangular matrix under FE discretization. While we do not expect it to be coercive like the bilinear form  $a$ , nor invertible in any sense, the nondegeneracy condition in part (b) may apply.
  - (a) Suppose  $b(\mathbf{v}, q)$  is a finite-dimensional bilinear form on  $\mathbf{v} \in \mathbb{R}^m$  and  $q \in \mathbb{R}^n$ . Assuming bases of these spaces, define a matrix  $M \in \mathbb{R}^{m \times n}$  which represents  $b$ . (Essentially,  $M = B^\top$ .)
  - (b) Show that  $M$  has full column rank if and only if
 
$$b(\mathbf{v}, q) = 0 \text{ for all } \mathbf{v} \text{ implies } q = 0.$$
- 14.5. Consider definition (14.16) for the bilinear form  $k$ . This involves a choice of sign to combine (14.13) and (14.14).

- (a) Our choice makes the Stokes system matrix symmetric—confirm this—so that we may use MINRES.
- (b) An alternative definition appears in some literature (e.g., [107]), namely

$$\hat{k}(\mathbf{u}, p; \mathbf{v}, q) = a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) - b(\mathbf{u}, q).$$

Show  $\hat{k}$  is not symmetric, but that the discrete equations generated by  $\hat{k}$ , namely the system

$$\begin{bmatrix} A & B^\top \\ -B & 0 \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}, \quad (14.45)$$

is positive-semidefinite. How can this fact be exploited?

- 14.6. Consider matrix  $K_A$  (Figure 14.3) and assume that  $A \in \mathbb{R}^{n_u \times n_u}$  is SPD. Show that  $\sigma(K_A) \subset \{0\} \cup [\alpha_1, \alpha_2]$  where  $\alpha_i > 0$  and the zero eigenvalue has multiplicity  $n_p$ . Give tight bounds  $\alpha_i$  in terms of matrix norms.
- 14.7. Now consider  $K_B$  (Figure 14.3) and assume that  $B \in \mathbb{R}^{n_p \times n_u}$  has full rank, with  $n_u \geq n_p$ . Note that  $K_B$  is symmetric.
- (a) Show that if  $K_B \mathbf{z} = \lambda \mathbf{z}$  for  $\mathbf{z} \neq 0$  then there is  $\tilde{\mathbf{z}} \neq 0$  such that  $K_B \tilde{\mathbf{z}} = -\lambda \tilde{\mathbf{z}}$ . (*Hint.* Negate a portion of the vector.)
  - (b) Show that  $K_B$  has a zero eigenvalue of exact multiplicity  $n_u - n_p$ .
  - (c) Show that  $BB^\top$  is SPD and that if  $BB^\top \mathbf{p} = \mu^2 \mathbf{p}$  then there is  $\mathbf{z} \neq 0$  such that  $K_B \mathbf{z} = \mu \mathbf{z}$ . (*Hint.* Suppose  $\mathbf{z} = (cB^\top \mathbf{p}, \mathbf{p})$  and find  $c$ .)
- 14.8. Stokes model equations (14.5)–(14.6) are the variational equations of a saddle-point problem, but justifying this claim requires care. To start, assume homogeneous Dirichlet conditions on  $\Omega \subset \mathbb{R}^d$  and define

$$G(\mathbf{v}, q) = \int_{\Omega} \mu |D\mathbf{v}|^2 - q \nabla \cdot \mathbf{v} - \mathbf{f} \cdot \mathbf{v} \quad (14.46)$$

for  $\mathbf{v} \in (W_0^{1,2})^d = \mathcal{X}_0$  and  $q \in L^2$ , and where  $|D\mathbf{v}|^2 = D\mathbf{v} : D\mathbf{v}$  [49]. Consider the following saddle-point problem which we will relate to the strong form (14.5)–(14.6):

$$\inf_{\mathbf{v} \in \mathcal{X}_0} \sup_{q \in L^2} G(\mathbf{v}, q). \quad (14.47)$$

- (a) Fix  $\mathbf{v} \in \mathcal{X}_0$  such that  $\nabla \cdot \mathbf{v} \neq 0$ . Show that  $\sup_q G(\mathbf{v}, q) = +\infty$ . (*Hint.* Choose a sequence  $q_n \in L^2$ .)
- (b) Suppose  $p$  is nice enough so that  $\nabla p \in (L^2)^d$ . Let  $\mathcal{K} = \{\mathbf{v} \mid \nabla \cdot \mathbf{v} = 0\}$ , a linear subspace of  $\mathcal{X}_0$ . Show that if  $\mathbf{v} \in \mathcal{K}$  then  $\int_{\Omega} \mathbf{v} \cdot \nabla p = 0$ . (*Gradients are orthogonal to divergence-free fields.*)
- (c) On  $\mathcal{K}$  the functional  $G$  is independent of  $q$ , so define

$$H(\mathbf{v}) = G|_{\mathcal{K}}(\mathbf{v}, q) = \int_{\Omega} \mu |D\mathbf{v}|^2 - \mathbf{f} \cdot \mathbf{v}.$$

Show that if  $\mathbf{u}, p$  are classical solutions of (14.5)–(14.6) then  $\mathbf{u}$  minimizes  $H$  over  $\mathcal{K}$ . (*Hint.* Consider  $H(\mathbf{u} + \mathbf{w})$  where  $\mathbf{w} \in \mathcal{K}$ , integrate by parts, and use part (b).) Conclude that a classical solution of (14.5)–(14.6) solves (14.47).

- (d) Conversely, suppose  $\mathbf{u} \in \mathcal{X}_0$  and  $p \in L^2$  solve (14.47), and that  $G(\mathbf{u}, p) < +\infty$ . Use (a) to show  $\nabla \cdot \mathbf{u} = 0$ .
- (e) Suppose also that  $\mathbf{u}, p$  are sufficiently regular to do the next steps. Show that if  $\mathbf{v} \in \mathcal{X}_0$  then

$$G(\mathbf{v}, p) = \int_{\Omega} \mu |D\mathbf{v}|^2 + \nabla p \cdot \mathbf{v} - \mathbf{f} \cdot \mathbf{v}.$$

Conclude that  $\mathbf{u}, p$  solve (14.5) and (14.6). (*Hint.* Consider  $G(\mathbf{u} + \epsilon \mathbf{w}, p)$ , where  $\mathbf{w} \in \mathcal{X}_0$ , and integrate by parts when needed.)

- 14.9. Suppose  $K \in \mathbb{R}^{N \times N}$  is symmetric and  $M = HH^\top \in \mathbb{R}^{N \times N}$  is SPD. Show that  $M^{-1}K$  and  $H^{-1}KH^{-\top}$  have the same eigenvalues.
- 14.10. How was the Schur complement discovered? This exercise might make the algebra natural. Assume that  $A$  is SPD and  $B$  has full rank.

- (a) Do a block row operation on linear system (14.23) to yield the triangular system

$$\begin{bmatrix} A & B^\top \\ 0 & -BA^{-1}B^\top \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ -BA^{-1}f \end{bmatrix}.$$

- (b) Let  $S = -BA^{-1}B^\top$  be the *Schur complement* of  $A$  in  $K$ . Show that  $-S$  is SPD.
- (c) Continuing (a), proceed to the formal solution

$$p = -S^{-1}BA^{-1}f, \quad u = A^{-1}(I + B^\top S^{-1}BA^{-1})f.$$

(Note that only  $A$  and  $S$  are inverted here.)

- (d) Parts (a) and (c) use blockwise Gaussian elimination and back substitution, respectively. Continue with Gauss-Jordan elimination [143] to generate diagonal form, the congruence factorization (14.27):

$$\begin{bmatrix} A & B^\top \\ B & 0 \end{bmatrix} = \begin{bmatrix} I & 0 \\ BA^{-1} & I \end{bmatrix} \begin{bmatrix} A & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I & A^{-1}B^\top \\ 0 & I \end{bmatrix}.$$

- (e) Assuming  $C$  is symmetric and positive-semidefinite, generalize parts (a)–(d) to the equation

$$\begin{bmatrix} A & B^\top \\ B & -C \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}.$$

Here  $S = -(C + BA^{-1}B^\top)$  is the Schur complement. (This generalization is used for stabilized mixed FE methods [49].)

- 14.11. Verify inverse (14.29).
- 14.12. Prove Theorem 14.2. (*Hint.*  $A^{-1}B^\top(BA^{-1}B^\top)^{-1}B$  is a projection. Or see [116]; it is only four pages long!)
- 14.13. (*This sequel to Exercise 14.4 explores the meaning of inf-sup inequality (14.36), by comparing to the finite-dimensional case.*) Suppose  $b(v, q)$  is a bilinear form on  $v \in \mathbb{R}^m$  and  $q \in \mathbb{R}^n$ , and consider any norms on these spaces. Nondegeneracy of such a finite-dimensional bilinear form is equivalent to an inf-sup inequality. In fact, the following four conditions are equivalent, as will be shown below in parts (a) and (b):

- (i) For  $\ell \in (\mathbb{R}^n)^*$ , a solution  $q$  to  $b(v, q) = \ell(v)$ , for all  $v$ , is unique;
- (ii)  $b(v, q) = 0$  for all  $v$  implies  $q = 0$ ;
- (iii)  $\sup_{v \neq 0} \frac{b(v, q)}{\|v\|} = 0$  implies  $q = 0$ ;
- (iv) there exists  $\gamma > 0$  so that  $\inf_{q \neq 0} \sup_{v \neq 0} \frac{b(v, q)}{\|v\| \|q\|} \geq \gamma$ .

However, if the dimension were instead infinite then (iv) is stronger than the others. That is, in function spaces there may be no inf-sup constant even if the form is nondegenerate in the sense of (ii), for example.

- (a) Show: (i)  $\iff$  (ii)  $\iff$  (iii).
  - (b) Show: (iii)  $\iff$  (iv). (*Hint.* Consider the SVD.)
  - (c) Let  $b(v, q) = \int_0^1 xv(x)q(x) dx$  for  $v, q \in L^2[0, 1]$ . Show (ii) holds but not (iv).
- 14.14. This problem justifies the -analytical exact solution in `stokes.py`.
- (a) Recall that, by definition,  $\Psi$  is a stream function for  $\mathbf{u}$  if  $\mathbf{u} = \langle \partial\Psi/\partial y, -\partial\Psi/\partial x \rangle$  ([2]; Exercise 11.19). Show then that  $\nabla \cdot \mathbf{u} = 0$ .
  - (b) Let  $\Psi(x, y) = \frac{1}{4\pi} \sin(4\pi x) \sin(4\pi y)$  and find the corresponding  $\mathbf{u}$ . On  $\Omega = (0, 1)^2$ , let  $\mathbf{g}_D = \mathbf{u}|_{\partial\Omega}$  and  $p(x, y) = \pi \cos(4\pi x) \cos(4\pi y)$ . Assume  $\mu = 1$  and derive  $\mathbf{f}$  so that equation (14.5) holds. Confirm the formulas in the source code.
  - (c) Contour the stream function using your favorite tool.
  - (d) Compare a streamline view of a numerical solution  $\mathbf{u}$  (using your preferred resolution and solver options).
- 14.15. In `stokes.py`, geometric multigrid is bundled into the -schurmg options. Modify this to also allow algebraic multigrid (AMG), either `gamg` or `hypre`, for preconditioning the velocity-velocity block. (Note that `gamg` requires the MATAIJ type.) After confirming convergence, compare results with those in Figures 14.7 and 14.8.
- 14.16. Moffatt's [113] analysis shows that, because the ratio of successive eddy sizes goes to zero as the angle  $\alpha$  goes to zero, small corner angles should be easier cases in which to find eddies. In the case of  $\alpha \approx 30^\circ$ , for example, laboratory observations reveal two eddies [2]. Modify `lidbox.py` to build a triangular lid-driven cavity with arbitrary corner angle  $0^\circ < \alpha < 180^\circ$  and then use `stokes.py` and Paraview, and solver choices as discussed in the text, to reveal as many eddies as possible for your preferred corner angle. Karniadakis and Sherwin [87, Figures 1.3, 1.4], for example, use a graded mesh of only 30 triangular elements, but with polynomial degree 17, to reveal *nine* eddies in a  $28.1^\circ$  corner; reproduce this result. Also confirm that eddies disappear around  $\alpha = 146^\circ$ .
- 14.17. A goal for the design of `DMFlex` is dimension independence [95]. While the UFL statement of the Stokes weak form is already dimension independent, details of the mesh and the boundary conditions inevitably depend on the dimension.
- (a) Without changing any functionality, factor out the mesh-specific and boundary-condition parts of `stokes.py` into a module `meshbc.py`. Rename the remainder as `stoked.py`; it has essentially unaltered UFL weak form, solver configuration, and post-processing.

- (b) Add a dimension option `-d 2|3` to `stoked.py`. Implement 3D meshes, both uniform and refined in the corners, and lid-driven boundary conditions, on the unit cube  $\Omega = (0, 1)^3$ . (*Decide on what these boundary conditions should be and add to `meshbc.py` accordingly.*) Add a 3D `.geo` file for the refined corners case, and mesh it with `gmsh -3`. Your code should now run in 3D; test this.
- (c) Decide on and implement a scheme for verification of the 3D results. (Either modify the `-analytical` case or set up 3D boundary conditions for which the flow is the same as the 2D case.)
- (d) Can you use the recommended Schur+GMG preconditioners to show optimality in the 3D case? (*This can get expensive.*)
- (e) Add option `-d 1` for completeness. Just how trivial is your 1D solution? (Adding a body force makes it slightly less trivial.)

## Appendix

# Some numerical facts of life

At several places in the book we state inconvenient facts related to the complicated procedure of solving PDEs numerically. They are repeated below, with their page numbers.

Fact 1. On a digital computer there are unavoidable limitations to numerical accuracy. *If real numbers are represented in floating point with machine precision  $\epsilon$ , then the solution of  $A\mathbf{u} = \mathbf{b}$  can only be computed within an error  $O(\kappa(A)\epsilon)$ , where  $\kappa(A)$  is the condition number.* 11

Fact 2. Dense direct linear algebra has a high cost. *For a dense matrix  $A \in \mathbb{R}^{N \times N}$ , computation of the solution to  $A\mathbf{u} = \mathbf{b}$ , by a direct method such as Gauss elimination (LU decomposition), whether forming  $A^{-1}$  or not, requires  $O(N^3)$  operations.* 11

Fact 3. You should not assemble the inverse. *The matrices  $A$  arising from discretized PDEs are often sparse, but their inverses  $A^{-1}$  are usually dense and may not even fit in memory.* 11

Fact 4. The linear system solver error is not the numerical error of the PDE method. *Though solving a linear system  $A\mathbf{u} = \mathbf{b}$  may be part of your method, making  $\|\mathbf{e}\| = \|\mathbf{v} - \mathbf{u}\|$  small for this system does not control the discretization error or the total numerical error.* 12

Fact 5. The rate of residual reduction in a Krylov iteration is at the mercy of the spectrum of your preconditioned matrix. *Whatever Krylov iteration you choose, whether norm-minimizing or not, good performance depends on the spectral properties of your matrix. A fast solver for a PDE problem must somehow make the spectral properties of the preconditioned matrix so good that the Krylov choice becomes almost unimportant.* 21

Fact 6. Learning PETSC requires viewing solver objects at run time. *If you did not view the solver with `-ksp_view`, `-snes_view`, or `-ts_view` then you probably do not know what it did, even if it succeeded. Viewing solvers is the first step to understanding their composition.* 29

Fact 7. Parallel preconditioning generally depends on processor count. *When the number of MPI processes changes, a block-matrix or domain-decomposition preconditioner also changes.* 38

Fact 8. Understanding the theory of convergence and stability for FD schemes requires thinking globally, beyond the local truncation error. *One must consider either the norms or eigenvalues of the family of matrices which are generated as the mesh is refined, and these are global considerations.* 58

Fact 9. Error stagnation will occur at some level of refinement. *For a given floating-point precision, at some point in the refinement path the round-off error will become comparable with the discretization error. Beyond this level, convergence cannot be verified.* 83

Fact 10. Residual-evaluation code must be correct. *All other choices about solving nonlinear equations—Jacobian-evaluation methods, linear solvers, line search, initial iterates, etc.—are irrelevant if your implementation of the nonlinear residual  $\mathbf{F}(\mathbf{x})$  is wrong.* 90

Fact 11. Stability is obligatory in a numerical scheme. *You may seek greater accuracy, but exponential growth of the approximation to a bounded or decaying solution is never acceptable.* 109

Fact 12. Effective preconditioners aren't like Krylov iterations. *If you are already using a norm-minimizing Krylov method then you need to add a fundamentally different idea to build a fast solver. In PETSc such ideas are lumped into the preconditioner paradigm. LU decomposition, domain decomposition, multigrid, and fieldsplit are examples.* 131

Fact 13. Stalling numerical processes must be wrong [21]. *Whenever the computer grinds very hard for small or slow effect, there must be a better way to achieve the same goal.* 134

Fact 14. Strong scaling requires that each process be kept busy on a problem of substantial size. *For a parallel PDE solver with  $N$  total degrees of freedom shared over  $P$  processes, something like  $N/P > 10^5$  is suggested.* 207

Fact 15. Parallel efficiency requires assembling matrices using the same distribution as the solver [134]. *The vast majority of matrix entries should be generated on the process where they will be used most. Do not expect much benefit from setting up a big system elsewhere and then reading it into PETSc to “solve it in parallel.”* 212

Fact 16. The easiest way to make software scalable is to make it sequentially inefficient [73]. *Both deliberate and accidental attempts to “game” weak scaling come down to wasting time on each process, relative to the performance of the best solution method in serial.* 213

Fact 17. Parallel reductions are nondeterministic at the bit level. *Because floating-point arithmetic is not exactly associative, different orders of arrival during reductions will affect results.* 215

Fact 18. Achieving good-looking numerical advection results requires effort. *Numerical results for simple advection tend to reveal that high-frequency components are transported at the wrong rates. Nonlinear flux-limiters or slope-limiters can correct this, and are worth the effort.* 296

Fact 19. Advection is not stagnation. *Discretizations and iterations for advection-diffusion equations, either as solvers or multigrid smoothers, must be stable on coarse grids and follow the flow. Stagnation is only an issue when the mesh Peclet number is small and the choice of advection discretization unimportant.* 305

Fact 20. Defining “optimal” as  $O(N)$  work for  $N$  discrete degrees of freedom makes no sense for spectral methods. *In a successful spectral method the degrees of freedom are worth much more than they are in a fixed-polynomial-degree FE/FD/FV method.* 336

Fact 21. The Stokes equations have a zero diagonal block which requires attention. *The equations arise as an equality-constrained minimization problem, so a zero block appears in a mixed FE method. One must deal with this in some manner. For stable elements, Schur decompositions via fieldsplit still need a preconditioner for the Schur block.* 356

# Bibliography

- [1] S. ABHYANKAR, J. BROWN, E. M. CONSTANTINESCU, D. GHOSH, B. F. SMITH, AND H. ZHANG, *PETSc/TS: A modern scalable ODE/DAE solver library*, Tech. Rep. arXiv:1806.01437, arXiv, 2018. (Cited on p. 95)
- [2] D. J. ACHESON, *Elementary Fluid Dynamics*, Oxford University Press, Oxford, 1990. (Cited on pp. 312, 343, 344, 345, 348, 351, 365, 369)
- [3] M. ADAMS, M. BREZINA, J. HU, AND R. TUMINARO, *Parallel multigrid smoothing: polynomial versus Gauss–Seidel*, J. Comput. Phys., 188 (2003), pp. 593–610. (Cited on pp. 21, 22, 131, 133, 137, 182)
- [4] M. ADAMS, R. SAMTANEY, AND A. BRANDT, *Toward textbook multigrid efficiency for fully implicit resistive magnetohydrodynamics*, J. Comput. Phys., 229 (2010), pp. 6208–6219. (Cited on p. 176)
- [5] M. ALNÆS, A. LOGG, K. OLGAARD, M. ROGNES, AND G. WELLS, *Unified Form Language: A domain-specific language for weak formulations of partial differential equations*, ACM Trans. Math. Softw., 40 (2014), pp. 9:1–9:37. (Cited on pp. 274, 331)
- [6] G. M. AMDAHL, *Validity of the single processor approach to achieving large scale computing capabilities*, in Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, Association for Computing Machinery, New York, 1967, pp. 483–485. (Cited on p. 206)
- [7] U. M. ASCHER AND L. R. PETZOLD, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, SIAM, Philadelphia, 1998. (Cited on pp. x, 97, 98, 99, 104, 106, 108, 120, 121, 127)
- [8] U. ASCHER, S. RUUTH, AND R. SPITERI, *Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations*, Appl. Numer. Math., 25 (1997), pp. 151–167. (Cited on pp. 122, 125)
- [9] C. BAIOCCHI, *Sur un problème à frontière libre traduisant le filtrage de liquides à travers des milieux poreux*, C.R. Acad. Sci. Paris., 273 (1971), pp. 1215–1217. (Cited on p. 329)
- [10] S. BALAY ET AL., *PETSc Users Manual*, Tech. Rep. ANL-95/11 - Revision 3.12, Argonne National Laboratory, 2019. (Cited on pp. ix, xiv, 7, 15, 20, 31, 41, 44, 46, 47, 95, 110, 138, 181, 192, 200, 201, 202, 266, 267, 338, 339, 360)
- [11] S. BALAY, W. D. GROPP, L. C. MCINNES, AND B. F. SMITH, *Efficient management of parallelism in object-oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhäuser, Basel, 1997, pp. 163–202. (Cited on p. ix)
- [12] W. BANGERTH, R. HARTMANN, AND G. KANSCHAT, *deal.II—a general purpose object oriented finite element library*, ACM Trans. Math. Softw., 33 (2007), pp. 24:1–24:27. (Cited on p. 274)

- [13] D. BARKLEY, *Spiral meandering*, in Chemical Waves and Patterns, R. Kapral and K. Showalter, eds., Kluwer, Dordrecht, 1995, pp. 163–189. (Cited on p. 128)
- [14] J. BARRETT AND W. LIU, *Finite element approximation of the  $p$ -Laplacian*, Math. Comput., 61 (1993), pp. 523–537. (Cited on p. 221)
- [15] S. BENSON AND T. MUNSON, *Flexible complementarity solvers for large-scale applications*, Optim. Methods Softw., 21 (2006), pp. 155–168. (Cited on pp. 320, 321, 322, 328)
- [16] M. BENZI, *Preconditioning techniques for large linear systems: a survey*, J. Comput. Phys., 182 (2002), pp. 418–477. (Cited on pp. 17, 130)
- [17] P. BOGACKI AND L. SHAMPINE, A 3 (2) pair of Runge-Kutta formulas, Appl. Math. Lett., 2 (1989), pp. 321–325. (Cited on p. 99)
- [18] K. BRABAZON, M. HUBBARD, AND P. JIMACK, *Nonlinear multigrid methods for second order differential operators with nonlinear diffusion coefficient*, Comput. Math. Appl., 68 (2014), pp. 1619–1634. (Cited on pp. 184, 221)
- [19] D. BRAESS, *Finite Elements: Theory, Fast Solvers, and Applications in Elasticity Theory*, Cambridge University Press, Cambridge, UK, 3rd ed., 2007. (Cited on pp. x, 187, 246, 247, 264, 345, 347, 357, 358)
- [20] A. BRANDT AND C. W. CRYER, *Multigrid algorithms for the solution of linear complementarity problems arising from free boundary problems*, SIAM J. Sci. Stat. Comput., 4 (1983), pp. 655–684. (Cited on pp. 329, 330)
- [21] A. BRANDT AND O. E. LIVNE, *Multigrid Techniques: 1984 Guide with Applications to Fluid Dynamics*, Classics Appl. Math. 67, SIAM, Philadelphia, revised ed., 2011. (Cited on pp. xi, 129, 134, 136, 137, 152, 163, 169, 170, 303, 372)
- [22] A. BRANDT, S. MCCORMICK, AND J. RUGE, *Algebraic multigrid (AMG) for sparse matrix equations*, in Sparsity and Its Applications, D. Evans, ed., Cambridge University Press, Cambridge, UK, 1985, pp. 257–284. (Cited on p. 129)
- [23] G. BRATU, *Sur les équation intégrales non linéaires*, Bull. Soc. Math. France, 42 (1914), pp. 113–142. (Cited on pp. 93, 196)
- [24] S. BRENNER AND L. SCOTT, *The Mathematical Theory of Finite Element Methods*, Springer, New York, 3rd ed., 2007. (Cited on pp. 187, 190, 357, 358, 361)
- [25] W. L. BRIGGS AND V. E. HENSON, *The DFT: An Owner's Manual for the Discrete Fourier Transform*, SIAM, Philadelphia, 1995. (Cited on p. 135)
- [26] W. L. BRIGGS, V. E. HENSON, AND S. F. MCCORMICK, *A Multigrid Tutorial*, SIAM, Philadelphia, 2nd ed., 2000. (Cited on pp. 16, 62, 129, 132, 136, 152, 153, 156, 169, 184)
- [27] J. BROWN, M. G. KNEPLEY, D. A. MAY, L. C. MCINNES, AND B. SMITH, *Composable linear solvers for multiphysics*, in 2012 11th International Symposium on Parallel and Distributed Computing, IEEE, New York, 2012, pp. 55–62. (Cited on pp. 17, 137, 138, 193)
- [28] J. BROWN, B. SMITH, AND A. AHMADIA, *Achieving textbook multigrid efficiency for hydrostatic ice sheet flow*, SIAM J. Sci. Comput., 35 (2013), pp. B359–B375. (Cited on pp. 176, 210)
- [29] P. R. BRUNE, M. G. KNEPLEY, B. F. SMITH, AND X. TU, *Composing scalable nonlinear algebraic solvers*, SIAM Rev., 57 (2015), pp. 535–565. (Cited on pp. xiii, 67, 91, 209, 221, 229, 234)
- [30] E. BUELER, *Stable finite volume element schemes for the shallow ice approximation*, J. Glaciol., 62 (2016), pp. 230–242. (Cited on pp. 221, 234, 319)

- [31] J. BUTCHER, *Numerical Methods for Ordinary Differential Equations*, Wiley, Chichester, 2nd ed., 2008. (Cited on pp. 98, 99)
- [32] J. CHANG, M. S. FABIEN, M. G. KNEPLEY, AND R. T. MILLS, *Comparative study of finite element methods using the time-accuracy-size (TAS) spectrum analysis*, SIAM J. Sci. Comput., 40 (2018), pp. C779–C802. (Cited on p. 213)
- [33] J. CHANG, K. NAKSHATRALA, M. KNEPLEY, AND L. JOHNSON, *A performance spectrum for parallel computational frameworks that solve PDEs*, Concurrency Comput. Pract. Exp., 30 (2018). (Cited on pp. 202, 207)
- [34] G. CHARTRAND, L. LESNIAK, AND P. ZHANG, *Graphs & Digraphs*, CRC, Boca Raton, 5th ed., 2011. (Cited on pp. 84, 85, 338)
- [35] D. L. CHOPP, *Introduction to High Performance Scientific Computing*, Software Environments Tools 30, SIAM, Philadelphia, 2019. (Cited on p. 199)
- [36] P. G. CIARLET, *The Finite Element Method for Elliptic Problems*, Classics Appl. Math. 40, SIAM, Philadelphia, 2002. Reprint of the 1978 original. (Cited on pp. 190, 221, 238, 243, 245, 315, 332)
- [37] T. F. COLEMAN AND J. J. MORÉ, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM J. Numer. Anal., 20 (1983), pp. 187–209. (Cited on pp. 84, 85)
- [38] R. COURANT, K. FRIEDRICHS, AND H. LEWY, *Über die partiellen Differenzengleichungen der mathematischen Physik*, Math. Ann., 100 (1928), pp. 32–74. (Cited on p. 283)
- [39] L. DALCIN, R. PAZ, P. KLER, AND A. COSIMO, *Parallel distributed computing using Python*, Adv. Water Resources, 34 (2011), pp. 1124–1139. (Cited on pp. xiii, 331)
- [40] J. E. DENNIS AND R. B. SCHNABEL, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, Englewood Cliffs, 1983. (Cited on pp. 90, 91, 92)
- [41] P. DEUFLHARD, P. LEINEN, AND H. YSERENTANT, *Concepts of an adaptive hierarchical finite element code*, IMPACT Comput. Sci. Eng., 1 (1989), pp. 3–35. (Cited on p. 160)
- [42] E. DOEDEL, H. B. KELLER, AND J. P. KERNEVEZ, *Numerical analysis and control of bifurcation problems (I): bifurcation in infinite dimensions*, Int. J. Bifurcation Chaos, 1 (1991), pp. 493–520. (Cited on p. 93)
- [43] V. DOLEAN, P. JOLIVET, AND F. NATAF, *An Introduction to Domain Decomposition Methods: Algorithms, Theory, and Parallel Implementation*, SIAM, Philadelphia, 2015. (Cited on p. 129)
- [44] T. A. DRISCOLL, K.-C. TOH, AND L. N. TREFETHEN, *From potential theory to matrix iterations in six steps*, SIAM Rev., 40 (1998), pp. 547–578. (Cited on p. 130)
- [45] M. DRYJA AND O. WIDLUND, *An additive variant of the Schwarz alternating method for the case of many subregions*, Computer Science Technical Report 339, Courant Institute, 1987. (Cited on pp. 152, 155, 156)
- [46] D. DUNAVANT, *High degree efficient symmetrical Gaussian quadrature rules for the triangle*, Int. J. Numer. Methods Eng., 21 (1985), pp. 1129–1148. (Cited on p. 173)
- [47] D. L. EAGER, J. ZAHORJAN, AND E. D. LAZOWSKA, *Speedup versus efficiency in parallel systems*, IEEE Trans. Comput., 38 (1989), pp. 408–423. (Cited on p. 204)
- [48] V. EIJKHOUT, *Introduction to High Performance Scientific Computing*, Lulu.com, 2nd ed., 2015. (Cited on pp. xiv, 199, 200, 202, 206, 207, 215)
- [49] H. C. ELMAN, D. J. SILVESTER, AND A. J. WATHEN, *Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics*, Oxford University Press, Oxford, 2nd ed., 2014. (Cited on pp. x, 20, 61, 63, 65, 134, 152, 154, 169, 222, 232, 233, 240, 244, 245, 246, 247,

- 252, 258, 299, 300, 301, 302, 303, 304, 305, 307, 312, 332, 335, 340, 343, 345, 347, 348, 352, 353, 354, 357, 358, 360, 361, 362, 367, 368)
- [50] L. C. EVANS, *The 1-Laplacian, the  $\infty$ -Laplacian and differential games*, in Perspectives in Nonlinear Partial Differential Equations, vol. 446 of Contemporary Mathematics, American Mathematical Society, Providence, 2007, pp. 245–254. (Cited on pp. 221, 238)
- [51] ———, *Partial Differential Equations*, Graduate Studies in Mathematics, American Mathematical Society, Providence, 2nd ed., 2010. (Cited on pp. x, 43, 44, 78, 111, 127, 148, 183, 196, 219, 220, 221, 233, 243, 245, 259, 280, 315, 317, 318, 327, 347)
- [52] ———, *An Introduction to Stochastic Differential Equations*, American Mathematical Society, Providence, 2013. (Cited on p. 272)
- [53] R. FALGOUT, *An introduction to algebraic multigrid computing*, Comput. Sci. Eng., 8 (2006), pp. 24–33. (Cited on pp. 129, 267)
- [54] A. FORSGREN AND T. ODLAND, *On the connection between the conjugate gradient method and quasi-newton methods on quadratic problems*, Comput. Optim. Appl., 60 (2015), pp. 377–392. (Cited on p. 230)
- [55] A. C. FOWLER, *Mathematical Models in the Applied Sciences*, Cambridge University Press, Cambridge, UK, 1997. (Cited on pp. 299, 300)
- [56] R. W. FREUND, *A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems*, SIAM J. Sci. Comput., 14 (1993), pp. 470–482. (Cited on p. 306)
- [57] A. H. GEBREMEDHIN, F. MANNE, AND A. POTHEIN, *What color is your Jacobian? Graph coloring for computing derivatives*, SIAM Rev., 47 (2005), pp. 629–705. (Cited on pp. 84, 85)
- [58] A. GEORGE, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363. (Cited on p. 63)
- [59] C. GEUZAINIE AND J. REMACLE, *Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities*, Int. J. Numer. Methods Eng., 79 (2009), pp. 1309–1331. (Cited on pp. 249, 250)
- [60] D. GILBARG AND N. TRUDINGER, *Elliptic Partial Differential Equations of Second Order*, Springer, New York, 2001. Reprint of the 1998 edition. (Cited on pp. 183, 219, 243, 299)
- [61] R. GLOWINSKI AND J. RAPPAZ, *Approximation of a nonlinear elliptic problem arising in a non-Newtonian fluid flow model in glaciology*, ESAIM Math. Model. Numer. Analysis, 37 (2003), pp. 175–186. (Cited on p. 221)
- [62] S. GODUNOV, *A difference method for numerical calculation of discontinuous solutions of the equations of hydrodynamics*, Mat. Sb., 89 (1959), pp. 271–306. (Cited on pp. 287, 288)
- [63] D. GOLDBERG, *What every computer scientist should know about floating-point arithmetic*, ACM Comput. Surveys, 23 (1991), pp. 5–48. (Cited on p. 214)
- [64] G. GOLUB AND C. V. LOAN, *Matrix Computations*, Johns Hopkins University Press, Baltimore, 4th ed., 2013. (Cited on pp. 9, 13, 15, 17, 20, 21, 22, 30, 130, 133, 134, 137, 165, 176, 353)
- [65] C. GRÄSER AND R. KORNHUBER, *Multigrid methods for obstacle problems*, J. Comput. Math. (2009), pp. 1–44. (Cited on p. 324)
- [66] A. GREENBAUM, *Iterative Methods for Solving Linear Systems*, Frontiers Appl. Math. 17, SIAM, Philadelphia, 1997. (Cited on pp. x, 9, 15, 16, 17, 19, 20, 22, 39, 62, 65, 130, 131, 134, 176, 306, 354)

- [67] A. GREENBAUM AND T. CHARTIER, *Numerical Methods: Design, Analysis, and Computer Implementation of Algorithms*, Princeton University Press, Princeton, 2012. (Cited on p. 171)
- [68] A. GREENBAUM, V. PTÁK, AND Z. STRAKOŠ, Any nonincreasing convergence curve is possible for GMRES, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 465–469. (Cited on pp. xiii, 20)
- [69] R. GREVE AND H. BLATTER, *Dynamics of Ice Sheets and Glaciers*, Advances in Geophysical and Environmental Mechanics and Mathematics, Springer, Berlin, 2009. (Cited on p. 344)
- [70] D. GRIFFITHS, J. DOLD, AND D. SILVESTER, *Essential Partial Differential Equations: Analytical and Computational Aspects*, Springer, Cham, 2015. (Cited on p. x)
- [71] I. GRIVA, S. G. NASH, AND A. SOFER, *Linear and Nonlinear Optimization*, SIAM, 2nd ed., 2008. (Cited on p. 328)
- [72] W. GROPP, E. LUSK, AND A. SKJELLMØN, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA, 2nd ed., 1999. (Cited on pp. ix, 3, 6, 8, 24, 182, 199, 214)
- [73] W. D. GROPP, Exploiting existing software in libraries: Successes, failures, and reasons why, in Object Oriented Methods for Inter-operable Scientific and Engineering Computing: Proceedings of the 1998 SIAM Workshop, M. E. Henderson, C. R. Anderson, and S. L. Lyons, eds., SIAM, Philadelphia, 1999, pp. 21–29. (Cited on pp. 213, 372)
- [74] J. L. GUSTAFSON, Reevaluating Amdahl’s law, Commun. ACM, 31 (1988), pp. 532–533. (Cited on pp. 204, 207)
- [75] G. HAGER AND G. WELLEIN, *Introduction to High Performance Computing for Scientists and Engineers*, CRC, Boca Raton, 2011. (Cited on p. 199)
- [76] A. HARTEN, High resolution schemes for hyperbolic conservation laws, J. Comput. Physics, 49 (1983), pp. 357–393. (Cited on p. 288)
- [77] V. E. HENSON AND U. M. YANG, BoomerAMG: A parallel algebraic multigrid solver and preconditioner, Appl. Numer. Math., 41 (2002), pp. 155–177. (Cited on pp. xiv, 266)
- [78] V. HERNANDEZ, J. E. ROMAN, AND V. VIDAL, SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems, ACM Trans. Math. Softw., 31 (2005), pp. 351–362. (Cited on p. xiv)
- [79] M. HESTENES AND E. STIEFEL, Methods of conjugate gradients for solving linear systems, J. Res. Nat. Bur. Stand., 49 (1952), pp. 409–436. (Cited on p. 19)
- [80] N. J. HIGHAM, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 2nd ed., 2002. (Cited on p. 41)
- [81] M. HINTERMÜLLER, K. ITO, AND K. KUNISCH, The primal-dual active set strategy as a semi-smooth Newton method, SIAM J. Optim., 13 (2003), pp. 865–888. (Cited on p. 319)
- [82] M. HIRSCH, S. SMALE, AND R. DEVANEY, *Differential Equations, Dynamical Systems, and an Introduction to Chaos*, Academic, San Diego, 2nd ed., 2004. (Cited on pp. 96, 276)
- [83] M. HOMOLYA, R. C. KIRBY, AND D. A. HAM, Exposing and exploiting structure: optimal code generation for high-order finite element methods, Tech. Rep. arXiv:1711.02473, arXiv, 2017. (Cited on p. 332)
- [84] W. HUNDSDORFER AND J. G. VERWER, *Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations*, vol. 33 of Springer Series in Computational Mathematics, Springer, Berlin, 2003. (Cited on pp. x, xii, 112, 123, 128, 285, 286, 288, 289, 297, 299, 311)

- [85] T. ISAAC AND M. G. KNEPLEY, *Support for non-conformal meshes in PETSc's DMFlex interface*, Tech. Rep. arXiv:1508.02470, arXiv, 2015. (Cited on pp. 331, 337, 339)
- [86] G. JOUVET AND E. BUELER, *Steady, shallow ice sheets as obstacle problems: Well-posedness and finite element approximation*, SIAM J. Appl. Math., 72 (2012), pp. 1292–1314. (Cited on p. 319)
- [87] G. KARNIADAKIS AND S. SHERWIN, *Spectral/hp Element Methods for Computational Fluid Dynamics*, Oxford University Press, Oxford, 2nd ed., 2013. (Cited on pp. x, 172, 173, 331, 335, 336, 369)
- [88] G. KARYPIS AND V. KUMAR, *Parallel multilevel k-way partitioning scheme for irregular graphs*, SIAM Rev., 41 (1999), pp. 278–300. (Cited on p. 274)
- [89] C. T. KELLEY, *Solving Nonlinear Equations with Newton's Method*, Fund. Algorithms 1, SIAM, Philadelphia, 2003. (Cited on pp. 68, 71, 72, 73, 74, 89, 91)
- [90] B. KERNIGHAN AND D. RITCHIE, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, 2nd ed., 1988. (Cited on pp. x, 5, 75, 263, 289)
- [91] D. KINDERLEHRER AND G. STAMPACCHIA, *An Introduction to Variational Inequalities and their Applications*, Academic, New York, 1980. (Cited on pp. 78, 315, 317)
- [92] R. C. KIRBY AND L. MITCHELL, *Solver composition across the PDE/linear algebra barrier*, SIAM J. Sci. Comput., 40 (2018), pp. C76–C98. (Cited on p. 360)
- [93] B. KIRK, J. PETERSON, R. STOGNER, AND G. CAREY, *libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations*, Eng. Comput., 22 (2006), pp. 237–254. (Cited on p. 274)
- [94] B. KLEMENS, *21st Century C: C Tips from the New School*, O'Reilly, Boston, 2nd ed., 2014. (Cited on pp. x, 5)
- [95] M. KNEPLEY AND D. KARPEEV, *Mesh algorithms for PDE with Sieve I: Mesh distribution*, Sci. Program., 17 (2009), pp. 215–230. (Cited on pp. 331, 369)
- [96] D. KNOLL AND D. KEYES, *Jacobian-free Newton–Krylov methods: a survey of approaches and applications*, J. Comput. Phys., 193 (2004), pp. 357–397. (Cited on pp. 86, 87, 88)
- [97] B. KOREN, *A robust upwind discretization method for advection, diffusion and source terms*, in Numerical Methods for Advection-Diffusion Problems, C. Vreugdenhil and B. Koren, eds., vol. 45 of Notes on Numerical Fluid Mechanics, Vieweg, Braunschweig, 1993, pp. 117–138. (Cited on p. 287)
- [98] R. KORNHUBER, *Monotone multigrid methods for elliptic variational inequalities I*, Numer. Math., 69 (1994), pp. 167–184. (Cited on pp. 329, 330)
- [99] A. N. KRYLOV, *On the numerical solution of the equation by which in technical questions frequencies of small oscillations of material systems are determined*, Izv. Akad. Nauk SSR, Otd. Mat. Estestv. Nauk, 7 (1931), pp. 491–539. (Cited on p. 17)
- [100] M. LANGE, L. MITCHELL, M. G. KNEPLEY, AND G. J. GORMAN, *Efficient mesh management in Firedrake using PETSc DMFlex*, SIAM J. Sci. Comput., 38 (2016), pp. S143–S155. (Cited on pp. 331, 337, 338, 339)
- [101] H. LANGTANGEN AND A. LOGG, *Solving PDEs in Python: The FEniCS Tutorial I*, Simula SpringerBriefs on Computing, Springer, New York, 2017. (Cited on p. 331)
- [102] P. D. LAX AND R. D. RICHTMYER, *Survey of the stability of linear finite difference equations*, Commun. Pure Appl. Math., 9 (1956), pp. 267–293. (Cited on p. 58)

- [103] R. J. LEVEQUE, *Finite Volume Methods for Hyperbolic Problems*, Cambridge University Press, Cambridge, UK, 2002. (Cited on pp. x, xii, 283, 289, 296, 297, 310, 311)
- [104] ———, *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*, SIAM, Philadelphia, 2007. (Cited on pp. x, 56, 57, 98, 112, 116, 284, 285)
- [105] X. S. LI, *An overview of SuperLU: Algorithms, implementation, and user interface*, ACM Trans. Math. Softw., 31 (2005), pp. 302–325. (Cited on p. 15)
- [106] J. LIOUVILLE, *Sur l'équation aux différences partielles  $\frac{d^2 \log \lambda}{dudv} \pm \frac{\lambda}{2a^2} = 0$* , J. Math. Pures Appl., 18 (1853), pp. 71–72. (Cited on pp. 93, 196)
- [107] A. LOGG, K.-A. MARDAL, AND G. N. WELLS, eds., *Automated Solution of Differential Equations by the Finite Element Method*, Springer, Heidelberg, 2012. (Cited on pp. 274, 331, 332, 348, 349, 358, 361, 367)
- [108] D. MAY, J. BROWN, AND L. L. POURHIET, *A scalable, matrix-free multigrid preconditioner for finite element discretizations of heterogeneous Stokes flow*, Comput. Methods Appl. Mech. Eng., 290 (2015), pp. 496–523. (Cited on pp. 344, 347)
- [109] D. MAY, P. SANAN, K. RUPP, M. KNEPLEY, AND B. SMITH, *Extreme-scale multigrid components within PETSc*, in Proceedings of the Platform for Advanced Scientific Computing Conference, ACM, New York, 2016. (Cited on pp. xi, 44, 175, 181, 212, 213, 214)
- [110] J. MCCALPIN, *Memory bandwidth and machine balance in current high performance computers*, IEEE Computer Society Technical Committee on Computer Architecture Newsletter (1995), pp. 19–25. (Cited on p. 202)
- [111] R. C. MCOWEN, *Partial Differential Equations: Methods and Applications*, Prentice-Hall, Englewood Cliffs, 2nd ed., 2003. (Cited on p. x)
- [112] J. MEIJERINK AND H. A. VAN DER VORST, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric m-matrix*, Math. Comput., 31 (1977), pp. 148–162. (Cited on pp. 15, 30, 34)
- [113] H. K. MOFFATT, *Viscous and resistive eddies near a sharp corner*, J. Fluid Mech., 18 (1964), pp. 1–18. (Cited on pp. 348, 365, 366, 369)
- [114] K. W. MORTON, *Numerical Solutions of Convection-Diffusion Problems*, Chapman & Hall, London, 1996. (Cited on pp. 279, 299, 311)
- [115] K. W. MORTON AND D. F. MAYERS, *Numerical Solutions of Partial Differential Equations: An Introduction*, Cambridge University Press, Cambridge, UK, 2nd ed., 2005. (Cited on pp. x, 47, 53, 57, 58, 78, 104, 112, 115, 116, 184, 240, 280, 283, 285, 297)
- [116] M. F. MURPHY, G. H. GOLUB, AND A. J. WATHEN, *A note on preconditioning for indefinite linear systems*, SIAM J. Sci. Comput., 21 (2000), pp. 1969–1972. (Cited on pp. 355, 356, 368)
- [117] N. M. NACHTIGAL, S. C. REDDY, AND L. N. TREFETHEN, *How fast are nonsymmetric matrix iterations?*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 778–795. (Cited on pp. xiii, 20)
- [118] J. NOCEDAL AND S. WRIGHT, *Numerical Optimization*, Springer, New York, 2nd ed., 2006. (Cited on pp. 13, 19, 88, 91, 229, 234, 239, 318, 328)
- [119] J. OCKENDON, S. HOWISON, A. LACEY, AND S. MOVCHAN, *Applied Partial Differential Equations*, Oxford University Press, Oxford, revised ed., 2003. (Cited on pp. x, 43, 78, 112, 183, 245, 279, 280, 344)
- [120] B. ØKSENDAL, *Stochastic Differential Equations: An Introduction with Applications*, Springer, Heidelberg, 5th ed., 2000. (Cited on p. 272)

- [121] D. PADUA, *Encyclopedia of Parallel Computing*, vol. 4, Springer, New York, 2011. (Cited on p. 214)
- [122] C. C. PAIGE AND M. A. SAUNDERS, *Solution of sparse indefinite systems of linear equations*, SIAM J. Numer. Anal., 12 (1975), pp. 617–629. (Cited on p. 20)
- [123] J. E. PEARSON, *Complex patterns in a simple system*, Science, 261 (1993), pp. 189–192. (Cited on pp. 120, 121, 125)
- [124] Y. PERES AND S. SHEFFIELD, *Tug-of-war with noise: a game-theoretic view of the  $p$ -Laplacian*, Duke Math. J., 145 (2008), pp. 91–120. (Cited on pp. 221, 238)
- [125] R. PLATO, *Concise Numerical Mathematics*, no. 57 in Graduate Studies in Mathematics, American Mathematical Society, Providence, 2003. (Cited on pp. 106, 171)
- [126] F. RATHGEBER ET AL., *Firedrake: automating the finite element method by composing abstractions*, ACM Trans. Math. Softw., 43 (2016), pp. 24:1–24:27. (Cited on pp. xii, xiii, 210, 252, 274, 331, 350)
- [127] M. REED AND B. SIMON, *Methods of Modern Mathematical Physics I: Functional Analysis*, Academic, revised and enlarged ed., 1980. (Cited on pp. 58, 357)
- [128] L. F. RICHARDSON, *The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam*, Philos. Trans. R. Soc. A, 210 (1911), pp. 307–357. (Cited on p. 12)
- [129] P. J. ROACHE, *The Method of Manufactured Solutions for Code Verification*, in Computer Simulation Validation: Fundamental Concepts, Methodological Frameworks, and Philosophical Perspectives, C. Beisbart and N. Saam, eds., Springer, New York, 2019, pp. 295–318. (Cited on p. 93)
- [130] J.-F. RODRIGUES, *Obstacle Problems in Mathematical Physics*, vol. 134 of North-Holland Mathematics Studies, Elsevier, Amsterdam, 1987. (Cited on pp. 315, 316, 317, 319)
- [131] Y. SAAD, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, 2nd ed., 2003. (Cited on pp. 17, 20, 39)
- [132] Y. SAAD AND M. H. SCHULTZ, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 856–869. (Cited on p. 20)
- [133] C. SCHOOF AND I. J. HEWITT, *Ice-sheet dynamics*, Annu. Rev. Fluid Mech., 45 (2013), pp. 217–239. (Cited on pp. 344, 347)
- [134] B. SMITH, P. BJORSTAD, AND W. GROPP, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press, Cambridge, UK, 1996. (Cited on pp. ix, 17, 129, 141, 143, 147, 148, 150, 151, 156, 167, 168, 212, 372)
- [135] J. SMITH, *The coupled equation approach to the numerical solution of the biharmonic equation by finite differences. I*, SIAM J. Numer. Anal., 5 (1968), pp. 323–339. (Cited on p. 190)
- [136] P. SONNEVELD, *CGS, a fast Lanczos-type solver for nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 36–52. (Cited on p. 306)
- [137] R. STENBERG, *Analysis of mixed finite elements methods for the Stokes problem: a unified approach*, Math. Comput., 42 (1984), pp. 9–23. (Cited on p. 357)
- [138] P. K. SWEBY, *High resolution schemes using flux limiters for hyperbolic conservation laws*, SIAM J. Numer. Anal., 21 (1984), pp. 995–1011. (Cited on p. 288)
- [139] C. TAYLOR AND P. HOOD, *A numerical solution of the Navier-Stokes equations using the finite element technique*, Comput. Fluids, 1 (1973), pp. 73–100. (Cited on p. 349)

- [140] J. L. THOMAS, B. DISKIN, AND A. BRANDT, *Textbook multigrid efficiency for the incompressible Navier–Stokes equations: high Reynolds number wakes and boundary layers*, Comput. Fluids, 30 (2001), pp. 853–874. (Cited on p. 176)
- [141] L. N. TREFETHEN, *Maxims about numerical mathematics, science, computers, and life on earth*. Modified from a list handed out in Trefethen’s Spring 1997 Cornell course “Software Tools for Computational Science,” 1997. (Cited on p. 109)
- [142] ———, *Spectral Methods in MATLAB*, Software Environments Tools 10, SIAM, Philadelphia, 2000. (Cited on pp. x, 22, 83, 175, 285, 311, 331, 335, 336)
- [143] L. N. TREFETHEN AND D. BAU III, *Numerical Linear Algebra*, SIAM, Philadelphia, 1997. (Cited on pp. x, xi, 6, 9, 10, 11, 13, 17, 19, 20, 34, 58, 167, 214, 234, 309, 336, 368)
- [144] U. TROTTERBERG, C. W. OOSTERLEE, AND A. SCHULLER, *Multigrid*, Elsevier, Oxford, 2001. (Cited on pp. x, xi, 16, 129, 132, 136, 155, 156, 157, 159, 169, 181, 184, 187, 196, 213, 267, 305, 307, 312)
- [145] A. M. TURING, *The chemical basis of morphogenesis*, Philos. Trans. R. Soc. London, Ser. B, 237 (1952), pp. 37–72. (Cited on p. 121)
- [146] H. A. VAN DER VORST, *Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 13 (1992), pp. 631–644. (Cited on p. 306)
- [147] B. VAN LEER, *Towards the ultimate conservative difference scheme ii: Monotonicity and conservation combined in a second-order scheme*, J. Comput. Phys., 14 (1974), pp. 361–370. (Cited on p. 288)
- [148] ———, *Upwind and high-resolution methods for compressible flow: From donor cell to residual-distribution schemes*, in 16th AIAA Computational Fluid Dynamics Conference, Orlando, 2006, AIAA, Reston, p. 3559. (Cited on p. 287)
- [149] G. VAN ROSSUM AND F. DRAKE, *An Introduction to Python*, Network Theory Ltd., Surrey, 2011. (Cited on p. 331)
- [150] P. VANĚK, J. MANDEL, AND M. BREZINA, *Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems*, Computing, 56 (1996), pp. 179–196. (Cited on p. 267)
- [151] H. VON KOCH, *Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire*, Ark. Mat., 1 (1904), pp. 681–704. (Cited on p. 272)
- [152] A. WATHEN, *Preconditioning*, Acta Numerica, 24 (2015), pp. 329–376. (Cited on pp. 17, 20, 130, 134)
- [153] P. WESSELING, *Principles of Computational Fluid Dynamics*, Springer, Berlin, 2001. (Cited on pp. 52, 93, 289, 343)
- [154] J. O. WILKES AND S. W. CHURCHILL, *The finite-difference computation of natural convection in a rectangular enclosure*, AIChE J., 12 (1966), pp. 161–166. (Cited on p. 300)
- [155] S. WILLIAMS, A. WATERMAN, AND D. PATTERSON, *Roofline: An insightful visual performance model for multicore architectures*, Commun. ACM, 52 (2009), pp. 65–76. (Cited on p. 202)
- [156] G. WORSTER, *Understanding Fluid Flow*, Cambridge University Press, Cambridge, UK, 2009. (Cited on p. 343)
- [157] D. YOUNG, *Iterative methods for solving partial difference equations of elliptic type*, Trans. Am. Math. Soc., 76 (1954), pp. 92–111. (Cited on p. 132)



# Index

- $\infty$ -Laplacian, 238
- adaptive mesh refinement, 187
- additive Schwarz method, 16, 17, 148
  - convergence, 150
  - advection equations, 279
    - solution by characteristics, 280, 299, 309
  - advection-diffusion equations, 279
  - advection dominated, 299
  - exponential boundary layer, 300
  - inflow boundary condition, 299
  - internal layer, 299
  - Peclet number, 299
  - reduced problem, 299
- Ahdahl's law, 206
- algebraic multigrid, *see also* multigrid, 16, 129, 193, 266, 272, 322, 335
  - classical, 267, 277
  - coarse grids, 266
  - smoothed aggregation, 267
- alternating Schwarz method, 148
- AMG, *see* algebraic multigrid
- application context, 70, 81
- arithmetic intensity, 202
- ASM, *see* additive Schwarz method
- backward stable methods, 11
- Bash shell, xi
- BFGS, *see* quasi-Newton methods
- biharmonic equation, 189
- binary files, 272
- block Jacobi, 16, 17, 133
- BoomerAMG, *see* Hypre
- Bratu equation, *see* Liouville-Bratu equation
- bytes transferred, 202
- C programming language, x
- cache memory, 200
- calculus of variations, 220, 318
- Chebyshev iterative method, 21
- CHKERRQ() macro, 7
- Cholesky decomposition, 15, 16
- circulant matrix, 309
- classical iterations
  - via richardson, 137
- cluster architecture, 200
- coercive functional, 219, 318
- complementarity problem, *see* variational inequality
- compute node, 200
- concurrency (definition), 199
- condition number, 10
  - relation to rounding errors, 11
- conservation of energy, 43
- convex functional, 219, 317
- core (within a CPU chip), 200
- diffusion-reaction equation, 78, 120
- direct linear solver, 31, 351
  - ksp\_type preonly, 31
  - nested-dissection ordering, 63
- Dirichlet boundary conditions, 44
- discrete conservation, 281
- dispersion, 282
- DM, xiv, 337
  - defined as "data management," "distributed mesh," or "distribution manager", 44
  - DMCreateMatrix(), 80, 262
  - dm\_view, 54
  - ghost nodes, 47
  - implementing mesh infrastructure without, 243
- DMDA, 44, 213, 269, 315
  - boundary types, 46

- 
- creating, 44  
`-da_grid_x`, 44  
`-da_grid_y`, 44  
`-da_refine`, 46  
`DMDACreate1d()`, 65  
`DMDACreate2d()`, 44  
`DMDACreate3d()`, 66  
`DMDALocalInfo`, 50, 79, 147  
`DMDASetUniformCoords()`, 144  
`DMDASNESSetFunctionLocal()`, 88  
`DMDASNESSetJacobianLocal()`, 88  
`DMDASNESSetObjectiveLocal()`, 227  
`DMDAVecGetArray()`, 79  
 grid hierarchy, 180  
 locally owned part of grid, 50  
 parallel distribution of grid, 45  
 periodic boundary conditions, 121  
 refinement, 46  
 stencil types, 46  
 stencil width, 46  
 viewing graphically, 54  
**DMplex**, 129, 274, 331, 337  
 closure operation, 339  
 directed acyclic graph, 338  
 domain decomposition, 129, 137  
 local and global vector representation, 141  
 download example codes, 5  
`-draw_pause`, 27  
  
 eigenvalue of a matrix, 13  
 error  
     in linear system, 12  
     numerical, 47, 57  
         code for computing norm, 34  
         stagnation, 83  
 Euler-Lagrange equation, 220, 275  
  
 FD methods, *see* finite difference methods  
 FE methods, *see* finite element methods  
 finite difference methods, x  
     centered diffusion scheme, 184  
     consistency, 56  
     convergence rate, 57, 83  
     convergent scheme, 57  
     error equation, 57  
     forward-time centered-space, 310  
     Lax-Richtmeyer equivalence theorem, 58  
     leapfrog, 310  
     local truncation error, 57  
  
     stable scheme, 58  
     stencil, 83  
     symmetrizing the equations, 49  
 finite element methods, x, 37, 219  
     bilinear functions, 222  
     efficiency relative to structured-grid FD, 269  
     element residual, 247  
     Galerkin method, 245  
     h/p methods, 331, 335  
     hat function, 224  
     mixed method, 343, 347  
      $P_1$  elements, 246  
     parallel distribution of meshes, 274  
      $P_k$  elements, 274, 334  
      $Q_1$  elements, 188, 222  
      $Q_k$  elements, 274, 335  
     quadrature, 171  
     reference element, 171, 222, 248  
     stiffness matrix, 247  
     test functions, 244, 246  
     trial functions, 244, 246  
 finite volume methods, x, 279, 300  
     centered-flux formula, 281, 286  
     control volume, 280  
     Courant-Friedrichs-Lowy (CFL) condition, 283, 310  
     dispersion relations, 310  
     donor cell method, *see* first-order upwinding  
     first-order upwinding, 283, 286, 296  
     flux-limiter, *see also* high-resolution schemes, 286, 292  
     Godunov's barrier theorem, 287, 288  
     high-resolution schemes, 283, 286, 287, 296  
     Koren limiter, 287  
     leapfrog scheme, 285  
     mesh Peclet number, 300, 302  
     modified equation analysis, 285, 310  
     monotonicity-preserving scheme, 289  
     nonoscillatory discretizations, *see* high-resolution schemes  
     numerical diffusion, 282  
     total variation diminishing, 288  
     van Leer limiter, 288  
 Firedrake library, xiii, 274, 331  
 flops (floating-point operations), 5, 200  
 flops rate, 200

- fluids, *see also* Stokes model  
 eddies, 365  
 enclosed flow, 347  
 flow law, 344  
 incompressibility, 344  
 lid-driven cavity, 348  
 momentum conservation, 344  
 Navier-Stokes model, 344  
 Newtonian viscosity, 344  
 strain rate tensor, 344  
 stream function, 312  
 stress tensor, 344  
 viscosity, 344  
 flux conservation equation, 279  
 free boundary problem, *see* variational inequality  
 Frobenius inner product, 345  
 FV methods, *see* finite volume methods  
 Gauss quadrature, 232  
 Gauss-Seidel iteration, 132, 303  
 amplification factor, 170  
 Fourier analysis of smoothing, 169  
 is PC type **sor**, 16  
 is a preconditioner, 131  
 smoothing factor, 170  
 Gaussian elimination, *see* LU decomposition  
 geometric multigrid, *see also* multigrid, 16, 129, 137, 157, 193, 303, 322, 335  
 additive Schwarz method as a smoother, 164  
 optimality, 177  
 rediscretization, 147, 154, 158, 305  
 ghost points, 339  
 Git, 5  
`git grep`, 7  
 GMG, *see* geometric multigrid  
 Gmsh, 249, 272  
 .geo files, 249  
 refinement methods, 251  
 grep, 7  
 grid sequencing, 160  
 heat equation, 111  
 diffusivity, 111  
 Fourier's law, 43  
 Neumann condition, 115  
 help string, 6  
 using grep on -help output to find options, 7  
 high frequency mode, 135, 170  
 high performance computing, 199  
 $h$ -refinement, 335, 362  
 Hypre, xiv, 266  
 ICC, *see* incomplete Cholesky  
 ILU, *see* incomplete LU  
 imbalance ratio, 200  
 incomplete Cholesky, 15, 16, 254, 264  
 scaling of iterations, 63  
 incomplete LU, 15, 16  
 as a smoother, 307  
 index set, 138, 252  
 induced matrix norm, 10  
 inf-sup inequality, 357  
 injection matrix, 138  
 interconnect, 200  
 capacity, 202  
 internal layer, 312  
 inverse matrix, 10  
 dense for sparse matrices, 11  
 IS, *see* index set  
`ISGetIndices()`, 256  
 Jacobi iteration, 14, 16, 131  
 independent of concurrency, 38  
 is a preconditioner, 131  
 Jacobian, 68  
 FD evaluation via coloring, 84  
 algorithms, 85  
 column-intersection graph, 84  
 distance-2 coloring, 84  
 finite difference (FD) evaluation, 71  
 of a weak form, 240  
 Picard approximation, 259  
 Jacobian-free Newton-Krylov, 86, 187, 229, 258, 297  
 with preconditioning operator, 88  
 JFNK, *see* Jacobian-free Newton-Krylov  
 Koch snowflake, 272  
 Krylov space, 17  
 Krylov space methods, *see also* KSP , 17  
 approximation theory, 18  
 as iterative refinement, 21  
 CG, *see* conjugate gradients  
 Chebyshev, 21  
 as a smoother, 22, 137  
 relation to polynomials, 22  
 conjugate gradients, 19, 254, 334

algorithm, 39  
 as optimization algorithm, 19  
 bound on iterations, 62  
 error reduction and polynomials, 19  
 norm minimizing, 19  
 preconditioned, 40  
 scaling of iterations under grid refinement, 61  
 generalized minimum residuals, 20, 301, 351, 356  
     avoiding restarts, 60  
     restarts, 20  
 GMRES, *see* generalized minimum residuals  
 Jacobi and Gauss-Seidel are preconditioners, 20  
 minimum residuals, 20, 343, 351, 356  
 MINRES, *see* minimum residuals  
 Richardson iteration, 18  
     as a smoother, 136  
 KSP, 21, 27  
     bcgs, 20  
     bicg, 20  
     cg, 21  
     cgne, 20  
     cgs, 20  
     chebyshev, 21  
     defaults, 30, 56  
     gmres, 21  
     -ksp\_atol, 30  
     -ksp\_converged\_reason, 36  
     -ksp\_max\_it, 162  
     -ksp\_monitor, 30  
     -ksp\_monitor\_solution, 55  
     -ksp\_monitor\_true\_residual, 195  
     -ksp\_norm\_type, 162  
     -ksp\_rtol, 30  
     KSPSetFromOptions(), 54  
     KSPSetNullSpace(), 277  
     KSPSetOperators(), 28, 54  
         two Mat arguments to set, 28  
     KSPSolve(), 29  
     -ksp\_type, 21  
     -ksp\_view, 29  
     -ksp\_view\_eigenvalues, 41  
     -ksp\_view\_mat, 35  
     -ksp\_view\_rhs, 35  
     -ksp\_view\_singularvalues, 62, 234  
     minres, 21  
     parallel defaults, 31  
     preonly, 15  
     richardson, 21  
     table of types, 21  
     tfqmr, 20  
     viewing Mat graphically, 55  
     viewing convergence with line graph, 55  
     viewing solution graphically, 55  
 Laplacian operator, 43  
     5-point stencil, 123  
     9-point stencil, 123  
     finite difference approximation, 47  
 line search, *see* Newton iteration  
 Liouville-Bratu equation, 245, 277, 312  
 Lipschitz function, 96  
 -log\_view, 7, 72, 200  
 LU decomposition, xi, 11, 15, 16  
      $O(N^3)$  operations for dense matrices, 11  
 machine precision, 11  
 make  
     makefiles, 5  
 manufactured solutions, 245  
 Mat, 23  
     create and configure, 23  
     loading from file, 35  
     MATAIJ, 369  
     MATAIJ type, 24, 350  
     MatAssemblyBegin(), 26  
     MatAssemblyEnd(), 26  
     MATBAIJ type, 350  
     MatGetOwnershipRange(), 33  
     -mat\_is\_symmetric, 185  
     MATMPIAIJ type, 25  
     MatMult(), 33  
     MATNEST type, 350  
     MatNorm(), 10  
     MATSEQAIJ type, 27  
     MatSetFromOptions(), 23  
     MatSetValues(), 26, 76  
     MatStencil, 51  
     Mat-Vec product, 25  
     -mat\_view, 23, 26, 27  
     may not have entries, 23  
     preallocation of memory, 262  
     saving  
         binary format, 27  
         MATLAB text format, 27  
     setting entries, 25  
     viewing entries, 26

- viewing graphically, 55  
matrix, 9  
matrix splitting, 16, 131  
and simple iteration, 16  
`-memory_view`, 201  
Message Passing Interface, ix, 3  
broadcast, 8  
collective operations, 24  
communicator, 6, 24  
`MPI_Allreduce()`, 6, 226  
MPICH, 4  
`MPI_Comm_rank()`, 6  
`mpiexec`, 5  
rank, 5, 40  
reduce, 8  
send and receive, 8  
method of lines, 95, 112, 120, 280, 289  
minimal surface equation, 183  
MPI, *see* Message Passing Interface  
multigrid, xi, *see also* algebraic multigrid,  
*see also* geometric multigrid, *see also* subgrid, 16, 129  
Chebyshev-based smoothers can adapt to  
anisotropy, 164  
classical iterations as smoothers, 156  
coarse-grid correction, 143, 154, 156  
coarse-grid matrix, 143  
Galerkin, 143, 153, 158, 266, 271  
coarsest grid problem, 158  
computational cost model, 159  
Dryja and Widlund two-level scheme,  
155, 164  
exposure of the solver, 161  
full approximation scheme, 209  
full cycle, 160  
incomplete factorizations as smoothers,  
307  
Kaskade cycle, 160  
monolithic solver versus `fieldsplit`,  
193  
nonlinear full cycle, 160, 186, 235  
optimality, 187  
uses `-snes_grid_sequence`, 160  
on `fieldsplit` blocks, 193  
parallelization, 178  
pre- and post-smoothers, 157, 168  
prolongation matrix, 138  
redundant coarse-grid solves, 181  
restriction matrix, 138  
full-weighting, 153  
smoothing operator, 305  
two-grid scheme, 156  
V-cycle, 158, 305  
W-cycle, 159  
Nelder-Mead method, 229  
Neumann boundary conditions, 238  
Newton iteration, 67, 319  
globalization, 74, 90  
grid sequencing, 186  
initial iterates, 186  
line search, 90  
merit function, 91  
sufficient decrease, 91  
using an objective, 232  
merit function, 91  
Picard approximation, 259  
quadratic convergence, 73, 90  
reduced-space method, 328  
Newton-Krylov methods, xi  
Newton-multigrid method, 184  
nonlinear complementarity problem  
(NCP), *see* variational inequality  
nonlinear conjugate gradients, 229  
nonlinear diffusion equations, 221  
nonlinear Poisson equation, 243  
nonuniform memory access, 200  
objective functional, 219  
obstacle problem, *see* variational inequality  
ODEs (ordinary differential equations), 95  
A-stable, 106  
absolutely stable, 104, 105  
test equation, 105  
absolutely stable (revised definition), 106  
adaptive Runge-Kutta implicit/explicit  
methods (ARKIMEX), 122  
adaptive time-stepping, 99, 294  
backward Euler method, 97  
backwards differentiation formula, 99,  
119, 125  
consistency of a scheme, 97  
convergence of a scheme, 98  
Crank-Nicolson method, 104, 109, 119,  
125, 281, 297  
embedded-pair adaptive methods, 99  
Euler method, 97  
implicit methods, 97, 297  
implicit/explicit methods (IMEX), 120  
initial value problems, 95

local truncation error, 97  
 one-step numerical methods, 98  
 order of a numerical method, 98  
 region of absolute stability, 106, 284  
 Runge-Kutta methods, 98, 293, 294
 

- classical fourth-order method RK4, 98
- explicit trapezoid method RK2a, 98, 281
- meaning of a tableau, 98
- RK3bs, 99, 294
- solved by TS objects, 95

 stability function of a scheme, 106  
 stability of a scheme, 98  
 stages of a one-step numerical method, 98  
 stiff decay, 108, 114
 

- test equation, 127

 stiff systems, 104  
 theta methods, 103, 120  
 trapezoid method, 104  
 optimal solution methods, xii, 175, 336
 

- p*-Helmholtz equation, 236
- for hyperbolic PDEs, 177
- memory limited is a feature, 177
- Newton-Krylov, 176
- overall strategy, 176
- versus “textbook multigrid efficiency”, 176

 optimization, 237
 

- active-set method, 328
- conjugate gradient, 239
- equality constraints, 328
- inequality constraints, 328
- relation to PDEs, 237
- symmetry of the Hessian, 275

 options
 

- `PetscOptionsBegin()` and `PetscOptionsEnd()`, 33
- `PetscOptionsEnum()`, 289
- `PetscOptionsInt()`, 33

 parallel efficiency, 204  
 parallel partitioning of meshes, 274  
 parallel reductions, 6
 

- are nondeterministic, 214
- in CG, 39

 parallel speedup, 204  
 Paraview, 335, 365  
 PC, 15
 

- `asm`, 17

 bjacobi, 17  
 cholesky, 15  
 fieldsplit, 17, 128, 137, 189, 191, 343, 354
 

- additive, 128, 192
- multiplicative, 191
- schur, 343

 gamg, 16, 197, 266, 369  
 hypre, 197, 266, 369  
 icc, 15  
 ilu, 15
 

- ILU(0) is serial default, 30
- ILU(1), 308

 jacobi, 16, 351  
 lu, 15  
 mg, 16, 157, 193
 

- how to control, 161

 none, 15  
 parallel defaults, 31
 

- `-pc_asm_overlap`, 148
- `-pc_factor_mat_ordering_type`, 63
- `-pc_factor_shift_type`, 351
- `-pc_fieldsplit_type`, 192
- `-pc_mg_levels`, 158

 PCSetup needs to be parallel, 212  
`-pc_sor_forward`, 133  
`-pc_type`, 15  
 redundant, 17, 164, 181, 213  
 sor, 16
 

- `-sub_pc_type`, 38

 svd, 15  
 table of types, 15  
 telescope, 17, 213, 308  
 PDEs (partial differential equations), xii
 

- basic numerical concerns, 55
- classification and coverage, xii
- quasilinear elliptic, 183, 243, 259
- semilinear elliptic, 183
- strong form, 221
- weak form, 220, 244

 PETSc (Portable, Extensible Toolkit for Scientific computation), ix  
 configuration and installation, 4  
 error macros, 7  
 floating-point types, 4
 

- `-help`, 6
- `-help intro`, 6

 initializing and finalizing, 6  
 integer types, 4  
 is not a silver bullet, x

- measuring run time, 8  
 object-oriented but written in C, 23  
 PETSC\_ARCH, 5  
 PETSC\_COMM\_SELF, 6  
 PETSC\_COMM\_WORLD, 6  
 PETSC\_DECIDE, 46  
 PETSC\_DIR, 5  
 PetscErrorCode, 7  
 PetscFinalize(), 4  
 PetscInitialize(), 4  
 PetscPrintf(), 6  
 PetscViewer, 35  
 profiling by logging stage times, 264  
 trace-backs, 7  
 Users Manual, xiv  
 -with-debugging=0, 66  
 -with-debugging=1, 7  
**petsc4py**, xiii, 331  
 $p$ -Helmholtz equation, 220  
 Picard iteration, 240, 259  
     as modified Newton iteration, 259  
     convergence, 276  
 $p$ -Laplacian operator, 221  
     regularization, 233  
 Poisson equation, 43, 143, 243, 332  
 porous medium equation, 245, 269  
 power spectrum, 135  
 preallocation of matrices, 272  
 preconditioner, 130  
 preconditioning, xi  
     additive composition, 142  
     additive two-level scheme, 155  
     black-box preconditioner, 130  
     composition of subgrid corrections, 142  
     conditions for effectiveness, 14  
     depends upon concurrency, 60  
     extremes, 15  
     left and right, 14  
     material, 29  
         supplying to SNES, 76  
     multiplicative composition, 142, 159, 167  
     parallel, 38  
     relation to spectrum, 14  
     symmetric, 15, 39, 354  
 $p$ -refinement, 335, 362  
 problem size (definition), 199  
 process placement, 201  
 Python, 331  
 quadrature, 171  
 degree, 171  
 Gauss-Legendre, 171  
 symmetric rules for triangles, 173  
 weight function, 171  
 quasi-Newton methods, 229  
 reaction-diffusion equation, *see also* diffusion-reaction equation  
 residual  
     of a linear system, 12  
     of a nonlinear system, 67  
     relation to  $A^\top A$ -norm of error, 20  
 Richardson iteration, *see also* simple iteration, 12, 131  
     is steepest descent, 13  
     no convergence test with SOR, 133  
     no right-preconditioned form, 131  
 Robin boundary conditions, 244  
 row scaling of a matrix, 48  
 run time (definition), 200  
 Schur complement, 343, 353, 355, 368  
 simple iteration, 16, 131  
     is left-preconditioned Richardson, 16  
 singular value of a matrix, 13  
 singular-value decomposition, 15, 16  
 SLEPc, 41  
 smoother  
     Chebyshev, 137  
     general KSP, 136  
     Richardson avoids parallel communication, 136  
 smoothing factor of a linear iteration, 136  
 SNES (Scalable Nonlinear Equation Solver), *see also* Newton iteration, 68, 315, 319  
     SNESFunction type, 75  
     SNESJacobianFunction type, 75  
     actions inside newtonls, 71  
     basic advice, 90  
     call-backs, 68  
     cast application context to void\*, 75  
     convergence tolerances, 72, 73  
     finite-differenced Jacobians, *see* Jacobian function evaluations may dominate cost of Newton solve, 72  
     line-search options, 91  
     monitor example, 188  
     ncg, 229  
     newtonls, 71

- options for Jacobian usage, 88  
 passing parameters, 74  
`qn`, 229  
 signature of Jacobian call-back, 75  
 signature of residual call-back, 70  
`-snes_atol`, 74  
`-snes_converged_reason`, 73  
`-snes_fd`, 71, 228, 257  
 does not scale for PDEs, 83  
`-snes_fd_color`, 71, 228, 262, 269, 272  
`-snes_fd_function`, 228  
`SNESGetDM()`, 145  
`SNESGetSolution()`, 145  
`-snes_grid_sequence`, 235, 324  
 replaces `-da_refine`, 186  
`-snes_mf`, 71, 228, 258  
`-snes_mf_operator`, 89, 187, 228, 229, 269  
`-snes_monitor`, 70  
`SNESMonitorSet()`, 92  
`-snes_monitor_solution`, 235  
`-snes_rtol`, 70, 74  
`SNESSetFromOptions()`, 69  
`SNESSetFunction()`, 70  
`SNESSetJacobian()`, 76  
 two Mat arguments to set, 75  
`SNESSolve()`, 69  
`-snes_stol`, 74  
`-snes_test_jacobian`, 82, 297  
`-snes_type`, 71, 229  
`-snes_view`, 79  
`-snes_vi_monitor`, 323  
 user-supplied Jacobian function, 74  
 testing, 77  
 variational inequality subtypes, 315  
 visualization, 324  
`vinewtonrsls`, 320  
`vinewtonssls`, 320  
 Sobolev space  $W^{1,p}(\Omega)$ , 219  
 socket (within a compute node), 200  
 solver complexity, 175, 236  
 SOR, *see* successive over-relaxation  
 sparse matrix, 11  
 SPD matrix, *see* symmetric and positive-definite matrix  
 spectral methods, 175, 331, 335  
 spectral radius of a matrix, 13  
 spectrum of a matrix, 13  
 SSOR, *see* symmetric successive over-relaxation  
 stagnation, 134  
 must be wrong, 134  
 static scaling, 183, 207  
 stencil  
 box, 48, 121, 222  
 star, 48, 292  
 Stokes model, *see also* fluids, 344  
 boundary conditions, 345  
 continuous-discontinuous elements, 361  
 discrete equations, 348  
 inf-sup inequality, 357  
 mass matrix, 360  
 stable elements, 358  
 system matrix, 348  
 eigenvalues, 352  
 Taylor-Hood elements, 361  
 weak form, 346  
 well-posedness, 347  
`streams` benchmark, 202  
 strictly convex functional, 220  
 strong scaling, 203  
 $N/P$  must be sufficiently large, 204  
 structured grids, *see* DM $\Delta$   
 subgrid, *see also* domain decomposition,  
*see also* multigrid, 137  
 correction matrix, 141  
 matrix, 139  
 prolongation matrix, 138  
 restriction matrix, 138  
 successive over-relaxation, 16, 132  
 SVD, *see* singular-value decomposition  
 Sylvester's law of inertia, 353  
 symmetric and positive-definite matrix, 13  
 associated norm, 19  
 symmetric successive over-relaxation, 133, 304  
 TAO library, xiv, 229  
 total variation, 310  
 transpose of a matrix, 10  
 triangulation, 246  
 topology versus geometry, 252  
 tridiagonal matrix, 34  
 TS (time-stepping), xiv, *see also* ODEs, 95  
`arkimex`, 125  
`bdf`, 99, 120  
`beuler`  
 is  $\theta = 1$  theta type, 110

- cn**
  - is  $\theta = 1/2$  theta type, 110
- initialization, 100
- Jacobians, 109
  - for implicit part, 124
- monitor function, 116
- rk**, 98, 103, 119
  - default is RK3bs, 99
  - run-time options, 102
  - setting tolerances, 103
  - theta type, 110
    - ts\_adapt\_type, 103, 120
    - ts\_arkimex\_type, 125
    - ts\_atol, 103
    - ts\_bdf\_order, 120
    - ts\_dt, 103
    - ts\_init\_time, 103
    - ts\_max\_time, 103
    - ts\_monitor, 102
    - ts\_monitor\_solution, 102
    - ts\_rk\_type, 105
    - ts\_rtol, 103
    - ts\_type, 100
    - ts\_view, 117
  - writing trajectory, 102
- UFL**, *see* Unified Form Language
- Unified Form Language, 274, 331, 350
- uniform ellipticity, 233, 243
- valgrind**, xi, 41, 202
- variational inequality, 315, 318
  - active set, 316
  - admissible set, 317
  - bound constraints, 320
  - complementarity problem, 318
  - Fischer-Burmeister function, 322
  - inactive set, 316
  - interior condition, 315, 318
- Karush-Kuhn-Tucker conditions, 318
- NCP (nonlinear complementarity problem) function, 322
- Newton-multigrid methods, 322
- nondegenerate solution, 319
- optimal solvers, 325
- reduced-space method, 320
- semismooth method, 320
- weak scaling, 325
- Vec**, 23
  - distinction between `VecDuplicate()` and `VecCopy()`, 33
  - distinction between
    - `VecRestoreArray()` and `VecDestroy()`, 70
  - interleaved storage, 190
  - loading from file, 35
  - parallel layout, 24
  - sequential layout, 24
  - `VecAssemblyBegin()`, 24
  - `VecAssemblyEnd()`, 24
  - `VecAXPY()`, 34
  - `VecGetArray()`, 70, 256
  - `VecGetArrayRead()`, 70, 256
  - `VecNorm()`, 10, 34, 294
  - `VecSetSizes()`, 24
  - `VecSetValues()`, 24
  - `vec_view`, 29
- vector Laplacian, 347
- vectors
  - bold font used for, 9
  - square brackets used for entries, 9
- VI**, *see* variational inequality
  - weak bounded, 210
  - weak efficiency, 210
  - weak scaling, 207, 308, 325
    - by making code serially inefficient, 212



The Portable, Extensible Toolkit for Scientific Computation (PETSc) is an open-source library of advanced data structures and methods for solving linear and nonlinear equations and for managing discretizations. This book uses these modern numerical tools to demonstrate how to solve nonlinear partial differential equations (PDEs) in parallel. It starts from key mathematical concepts, such as Krylov space methods, preconditioning, multigrid, and Newton's method. In PETSc these components are composed at run time into fast solvers.

Discretizations are introduced from the beginning, with an emphasis on finite difference and finite element methodologies. The example C programs of the first 12 chapters, listed on the inside front cover, solve (mostly) elliptic and parabolic PDE problems. Discretization leads to large, sparse, and generally nonlinear systems of algebraic equations. For such problems, mathematical solver concepts are explained and illustrated through the examples, with sufficient context to speed further development.

*PETSc for Partial Differential Equations*

- addresses both discretization and fast solvers for PDEs;
- emphasizes practice more than theory;
- contains well-structured examples, with advice on run-time solver choices;
- demonstrates how to achieve high performance and parallel scalability; and
- builds on the reader's understanding of fast solver concepts when applying the Firedrake Python finite element solver library in the last two chapters.

This textbook, the first to cover PETSc programming for nonlinear PDEs, provides a fast on-ramp for graduate students and researchers to a major area of high-performance computing for science and engineering. It is suitable as a supplement for courses in scientific computing or numerical methods for differential equations.

**Ed Bueler** is a Professor of Applied Mathematics at the University of Alaska Fairbanks. He teaches numerical analysis, was a lead author of the Parallel Ice Sheet Model, and has been learning PETSc since 2006.

---

For more information about SIAM books, journals,  
conferences, memberships, or activities, contact:



Society for Industrial and Applied Mathematics  
3600 Market Street, 6th Floor  
Philadelphia, PA 19104-2688 USA  
+1-215-382-9800 • Fax +1-215-386-7999  
[siam@siam.org](mailto:siam@siam.org) • [www.siam.org](http://www.siam.org)

SE31

ISBN: 978-1-611976-30-4

A standard 1D barcode representing the ISBN number 978-1-611976-30-4.

9781611976304

