SOLVING NONLINEAR HIGH-ORDER PDE SYSTEMS: METHODOLOGY AND A CLAWPACK LIBRARY [*]

JONATHAN CLARIDGE [†], R. LEVY [‡], AND J. WONG [§]

Abstract. This work presents a methodology for solving nonlinear systems of PDEs in a finite volume or difference framework with three main goals: to solve problems up to fourth-order without restrictions on boundary conditions, to minimize per-problem code-development, and enable rigorous, automated convergence testing on problems with analytical solutions. This methodology is implemented in a freely available extension of the Clawpack software.

Our goals rule out many common specialized approaches, such as those based on operator- or dimension-splitting, which can significantly complicate both convergence beyond first-order in time and the use of general boundary conditions. Consequently, the methodology we present is based on a standard, "textbook" approach to nonlinear solvers, identifying ways to keep per-problem complexity modest when linearizing in the course of Newton's method and when enforcing boundary conditions.

In addition to mathematical issues, we highlight concerns of interface and testing, both of which are crucial to software development. In considering interface, we ensure that all generic logic is contained in common library routines, and that code that must be written by the end user corresponds to meaningful abstractions. Thorough testing, in the meantime, ensures that both the methodology and implementation work as intended, and provide scaffolding for all future incremental improvements.

In the long term, we hope this contributes to the use of standardized, public numerical code in published research, whether through the methodology, implementation, or both.

Key words. Explicit machine computation and programs, finite difference methods, finite volume methods, packaged methods

AMS subject classifications. 35-04, 65M06, 65M08, 65Y15

1. Introduction.

The process of developing code for application-specific PDE solvers is by no means a standardized process. Individual research groups typically maintain independent code bases, and this code frequently is not made publicly available, even when published research depends on it. This model of development has been called into question with increasing frequency in recent years from the standpoint of reproducibility in research [3, 13, 20]. Furthermore, it places a heavy burden on individual researchers, as development and maintenance of code is time-consuming and difficult.

Our original aim for this project was to provide open source code for solving thin film problems, such as those found in [4, 14, 18], by finite volume methods. However, these problems involve fourth-order nonlinear systems of PDEs, and the associated complications to code development turn out to be very general in nature. Consequently, the scope of the project broadened considerably to the development of implicit finite volume solvers for generic nonlinear PDE systems in a way that can be easily reused, with a particular eye to the nuances of fourth-order problems.

Much of the methodology that is common in practice for nonlinear implicit solvers is accompanied by significant barriers to implementation and usability. Hence, we arrive at two closely-related goals. First, we identify and address the major conflicts between mathematical generality and code complexity that occur in the development of such solvers. Second, we provide an implementation based on these ideas that will

[*]This work was supported by grants NSF-DMS-0914942, NSF-FRG-0968154, Research Corp. Award 19788, and the ARCS Foundation.

[†]Google, Inc. (`claridge@gmail.com`).

[‡]Mathematics Department, Harvey Mudd College (`levy@g.hmc.edu`).

[§]Mathematics Department, UCLA (`jtwong@math.ucla.edu`).

hopefully be of direct use to other researchers. The implementation expressly supports problems up to fourth-order with a large class of supported boundary conditions that is designed to be extensible.

This paper itself focuses largely on the mathematical issues and on general aspects of the code's development, which should carry over to other implementations as well. Meanwhile, our code, which is an extension of the Clawpack software [12], is freely available as a GitHub repository at

`https://github.com/claridge/implicit_solvers`.

Version 1.0 was used to produce the results in this paper, but further improvements may be available at the head of the repository. Note to reviewers: This draft is based on v0.2, and v1.0 will be established following review.

At a high level, the blueprint for nonlinear implicit solvers is well-established. Following typical developments in numerical analysis textbooks [5, 11], the first step is to convert the PDE system to a system of ODEs. Next, by way of a discrete integration scheme, one converts the ODE system to a sequence of nonlinear algebraic systems. Then for each algebraic system one applies Newton's method: form a linear approximation to the algebraic operator, solve the resulting linear system of equations, and repeat until convergence is achieved. Tucked away in this blueprint, however, are two major sources of problem-specific human effort and code complexity: calculation of the Jacobian and the implementation of boundary conditions.

The Jacobian calculation requires that all discretized spatial differential operators be expanded in terms of stencil coefficients. Then all partial derivatives of these stencil coefficients must be evaluated, and the results must be carefully encoded into the computational version of the Jacobian operator. This process is easy enough to carry out for 1-dimensional second-order scalar problems, but progressing to higher dimensions, higher orders, and systems of PDEs can be prohibitive.

Within thin films literature, at least, techniques are frequently employed that reduce the complexity of the Jacobian calculation. Sequential integration of the equations in a system, which is justifiable by operator splitting, removes coupling between components. Meanwhile, the approximate factorization scheme of [19] replaces the Jacobian of a 2-dimensional fourth-order operator with a sequence of 1-dimensional operators.

Boundary conditions introduce a level of complexity on par with the Jacobian. First, outside of simple cases, determining the appropriate boundary conditions to use with the intermediate quantities of Newton's method requires careful consideration. Second, boundary conditions must be incorporated into any discretized differential operators, typically introducing irregular stencils near the boundary of the domain and adding many more distinct terms to the already formidable Jacobian. Third, techniques based on operator- or dimension-splitting, such as those mentioned above to simplify Jacobian calculations, complicate the use of any nontrivial boundary conditions. Obtaining accuracy any better than first-order in time requires complicated correction procedures [6, 9].

Fortunately, there are good solutions to these problems. Though the underlying mathematical ideas are not new, they are sometimes off the beaten path, and the challenge is to locate and select them from numerous available options. In the end, we are able to produce a solver library that requires a modest problem-specific investment of human labor, sacrifices no generality of the PDE system or boundary conditions for simplicity, and whose implementation as a whole is quite tractable. After covering background issues in §2, we address linearization in §3 and boundary conditions in

§4. For the impatient, though, here is a short list of suggestions: use iterative linear solvers for major interface benefits; use an approximate Jacobian if possible, and linearize in continuous space if not; and implement boundary conditions using ghost cells.

With the mathematical methodology in place, solver implementation is simplified greatly, but it still requires a fair amount of organization and detail management. Section 5 covers both the interface — the means by which a user leverages pre-existing code — and the implementation of internal routines.

Finally, we turn our attention to the question of "Does all of this work?" Section 6 addresses testing, which is crucial for proof of principle, debugging, and long-term maintenance. We provide a large set of automated tests on problems with analytical solutions, as well as a reproduction of results from a complex problem in literature with excellent qualitative agreement.

Throughout, we pay little direct attention to matters of performance, focusing instead on the flexibility to solve a breadth of problems. As concisely stated by [1], "get it right before you make it faster" is a worthy guideline that pushes execution time down our list of priorities. Even so, "getting it right" can be tremendously beneficial — in §6.3 we obtain results visually identical to an application study elsewhere in literature, but with time steps 10,000 times larger than originally used.

Furthermore, significant portions of the code will run in parallel on a multicore machine via OpenMP, and the code is accompanied by some simple guidelines for optimizing performance in user-written routines. For longer term development, we have done nothing that will fundamentally compromise subtler performance optimizations, such as preconditioning.

2. Mathematical background.

Before diving into the mathematical details, a bit of setup is necessary. Hopefully the reader is already familiar with the topics we briefly cover in this section, or at least one of their close relatives (e.g. finite differences rather than finite volumes).

2.1. Notation.

The remainder of the paper concerns a generic PDE on domain Ω. We denote the PDE, along with its boundary conditions, as

$$q_t = \mathbf{g}(q) \text{ on } \Omega, \qquad \mathbf{b}(t,q) = 0, \text{ on } \partial\Omega. \tag{2.1}$$

Here $q(x,t)$ is the solution, and $\mathbf{g}$ and $\mathbf{b}$ are both spatial differential operators. For example, $\mathbf{g} = \partial_{xx}$ and $\mathbf{b} = \partial_x$ specify a diffusion problem with homogeneous Neumann boundary conditions. In general $\mathbf{g}$ may depend on time, but we suppress this dependence for simplicity. It is retained for $\mathbf{b}$ because time-dependence is immediately of interest when we consider boundary conditions in §4.

We allow for the possibility that q, x, $\mathbf{g}$, $\mathbf{b}$, and all derived quantities, may be either scalar- or vector-valued. In this way, (2.1) can represent a system of PDEs in several spatial variables. The same convention applies to indices; for example, in 2-dimensional space, we use x_i to express what might otherwise be denoted (x_{i_1}, y_{i_2}). Most of the abstract discussion, however, should be perfectly readable with a 1-dimensional scalar-valued PDE in mind.

Superscripts are used to indicate approximations that occur at discrete points in time, such as $q^n(x) \approx q(x, t_n)$. Capitalization, on the other hand, corresponds to discrete-space approximations, with subscripts used as spatial indices. For example, $Q_i^n \approx q(x_i, t_n)$. A capital letter without any scripting denotes a vector of values on the spatial grid.

The rule for capitalization applies to operators as well, so $\mathbf{G}$ corresponds to a spatial discretization of $\mathbf{g}$. (Technically $\mathbf{G}$ must incorporate a discretization of $\mathbf{b}$ as well, but we postpone discussion of this complication until §4.) On a few occasions, specific notation for the act of spatial discretization is helpful, and for this we we use disc, e.g. $\mathbf{G} = \mathrm{disc}(\mathbf{g})$. Strictly speaking, a well-defined notion of disc would require choices of derivative stencils, but as part of our goal is to leave these choices to the end user, the ambiguity is appropriate.

2.2. Finite volume methods. The mathematical methodology we present is not particular to any discretization scheme, but for examples and implementation, we have to choose a specific one. We work with finite volume methods [10], in which the spatial domain is subdivided into cells, $\mathcal{C}_i$. To each cell $\mathcal{C}_i$, we associate an approximation to its cell average,

$$Q_i^n \approx \int_{\mathcal{C}_i} q(x, t_n)\, dx. \tag{2.2}$$

We restrict our attention to regular rectangular grids, for which Q_i^n approximates the value at the cell's center up to $\mathcal{O}(\Delta x^2)$.

We do not use finite volume methods in any deep way, and for our purposes they are largely interchangeable with finite differences. Their most notable influences, in terms of this project, is that we refer to grid cells rather than grid points, and sample applications commonly compute fluxes at the interfaces between grid cells. The finite volume framework is of much greater importance if one intends to accompany the implicit solvers we address with explicit solvers for hyperbolic terms.

2.3. Integration schemes. Our discussion of integration schemes focuses on backward Euler. It is the simplest implicit method, allowing exposition to be as straightforward as possible. Furthermore, it is the key component of second-order integration via Crank–Nicolson, which is produced by chaining together half-steps of forward and backward Euler. This chaining process may even be extended to implement an L-stable, second-order scheme as derived in [16].

2.4. Iterative linear solvers. Given a system of linear equations $\mathbf{A}(X) = B$, iterative linear solvers are notable in that the user needs only to supply them with a way to compute $\mathbf{A}(X)$ given X. This feature is extremely valuable for PDE solvers, as $\mathbf{A}$ is frequently composed of much simpler operators that are far easier to implement independently. It removes the need to represent $\mathbf{A}$ as a matrix, eliminating a major source of confusion when dealing with multiple spatial indices. Additionally, though also more subtly, it lets us decouple boundary conditions from the discretization of differential operators, as described in §4.2.

We use iterative solvers exclusively for the linear systems we encounter. In doing so, we forego the efficiency of banded solvers for 1-dimensional problems. But for problems of higher dimensions, iterative solvers are the state of the art in addition to carrying the benefits noted above.

In general, the linear systems we encounter are not symmetric positive definite, ruling out the otherwise exceptional conjugate gradient algorithm. Instead, we use the stabilized biconjugate gradient algorithm, or BiCGStab [17]. BiCGStab is easy to implement and requires far less work per iteration than the other natural choice, GMRES, but it carries the complication that it may fail to converge to the desired degree of accuracy. The fact that we are solving nonlinear problems proves to be an advantage in this case. Upon a convergence failure, BiCGStab still typically provides an approximate solution that is better than our initial guess, and we can use that approximation to proceed with Newton's method. No doubt this approach will still fail under certain improbable circumstances, but it works quite well in practice.

2.5. Computational platform. Our implementation is as an extension of the Clawpack library for developing PDE solvers [12]. Clawpack was originally designed to solve hyperbolic problems by explicit methods, but it provides an access point to apply other integration schemes at each time step.

By building upon Clawpack, we ensure that anybody using our code will have easy access to its hyperbolic integration schemes, and we provide existing Clawpack users with an extension for implicit integration. We furthermore save a significant amount of effort by leveraging Clawpack's build procedure, core PDE drivers, and means for defining configuration parameters.

The code is written in a fairly simple dialect of modern Fortran, notably using free-format source and dynamic memory allocation (not to mention names longer than six letters). Other newer features, such as pointers, derived data types, and modules, are currently avoided to maintain consistency with the current Clawpack release and to stay within a feature set that is almost certainly familiar to a majority of application developers. The set of features used is likely to grow in time, though, as they are embraced by forthcoming releases of Clawpack.

3. Linearization techniques.

When solving a nonlinear system of equations, we face the important task of linearizing the corresponding nonlinear operator for use with Newton's method. The traditional means of linearization, as observed in the introduction, can be notoriously hard to implement. Hence, it is helpful to replace direct linearization with a scheme that is approximate but much easier to encode. Here we review the direct approach and then present two approximation techniques that are good enough to have little impact on the convergence of Newton's method.

Throughout this section, we rely on the decoupling of boundary conditions via ghost cells, as described in §4.2. That is, we ignore boundary conditions, assuming that they will be handled independently. A reader who encounters this statement with a healthy dose of skepticism may wish to read that section in advance.

3.1. Discrete space linearization. In the context of PDE solvers, Newton's method most commonly appears after the problem has been fully discretized in both time and space. Beginning with the generic PDE (2.1), integration over a time step with backward Euler proceeds as follows.

First we discretize in time, letting $q^n(x) \approx q(x, t_n)$. Then we set

$$q^{n+1} = q^n + \Delta t\, \mathbf{g}(q^{n+1}). \tag{3.1}$$

Next we discretize in space, replacing q^n with the fully discrete vector Q^n, for which $Q_i^n \approx q^n(x_i)$, and we replace $\mathbf{g}$ with a discretized version $\mathbf{G}$. Doing so, we obtain

$$Q^{n+1} = Q^n + \Delta t\, \mathbf{G}(Q^{n+1}). \tag{3.2}$$

Hence, Q^{n+1} solves the nonlinear equation

$$\mathbf{M}(R) \equiv R - Q^n - \Delta t\, \mathbf{G}(R) = 0. \tag{3.3}$$

To solve (3.3), we apply Newton's method, for which we generate a sequence of iterates $R^{[0]}, R^{[1]}, R^{[2]}, \ldots$ that converges to Q^{n+1}. Given $R^{[k]}$, we form a linear approximation of $\mathbf{M}$ about $R^{[k]}$ and determine the perturbation $P^{[k]}$ such that the approximation vanishes at $R^{[k]} + P^{[k]}$. We then define $R^{[k+1]} \equiv R^{[k]} + P^{[k]}$. (The terms *iterate* and *perturbation* will refer exclusively to these intermediate quantities of Newton's method from now on.)

The linear approximation procedure yields the equation

$$\mathbf{M}'[R^{[k]}](P^{[k]}) = -\mathbf{M}(R^{[k]}), \tag{3.4}$$

where $\mathbf{M}'[R^{[k]}]$ is the linearization (Jacobian) of $\mathbf{M}$ about $R^{[k]}$. Our concern, then, becomes the solution of (3.4). The right-hand side is as straightforward to evaluate as one could hope for. By (3.3), it reduces to evaluation of the operator $\mathbf{G}$. A user-defined implementation of $\mathbf{G}$ is a natural component of any PDE solver, as it is the most basic way to express the operator $\mathbf{g}$ appearing in the original problem.

In order to use an iterative solver with (3.4), we must be able to evaluate $\mathbf{M}'[R](P)$ given any iterate R and perturbation P. Direct calculation shows that

$$\begin{aligned}\mathbf{M}'[R](P) &\equiv \lim_{\epsilon\to 0}\frac{\mathbf{M}(R+\epsilon P)-\mathbf{M}(R)}{\epsilon} &(3.5)\\ &= P - \Delta t \lim_{\epsilon\to 0}\frac{\mathbf{G}(R+\epsilon P)-\mathbf{G}(R)}{\epsilon} &(3.6)\\ &= P - \Delta t \mathbf{G}'[R](P). &(3.7)\end{aligned}$$

Hence, computing $\mathbf{M}'[R](P)$ reduces to the evaluation of $\mathbf{G}'[R](P)$. Typically, an implementation of $\mathbf{G}'[R](P)$ is another step that must be relegated to the end user.

In some situations, this job is not too difficult. For example, in the case of the 1-dimensional porous medium equation $q_t = (q^\alpha)_{xx}$, a centered difference approximation for ∂_{xx} gives us

$$\mathbf{G}(R)_i = \frac{R_{i-1}^\alpha - 2R_i^\alpha + R_{i+1}^\alpha}{\Delta x^2}. \tag{3.8}$$

Using the definition of the linearization directly, we obtain

$$\mathbf{G}'[R](P)_i = \lim_{\epsilon\to 0}\frac{(R_{i-1}+\epsilon P_{i-1})^\alpha - R_{i-1}^\alpha - 2[(R_i+\epsilon P_i)^\alpha - R_i^\alpha] + \cdots}{\epsilon}. \tag{3.9}$$

Grouping the terms for each index produces 3 scalar derivatives in ϵ, and we are left with

$$\mathbf{G}'[R](P)_i = \frac{\alpha R_{i-1}^{\alpha-1}P_{i-1} - 2\alpha R_i^{\alpha-1}P_i + \alpha R_{i+1}^{\alpha-1}P_{i+1}}{\Delta x^2}. \tag{3.10}$$

Unfortunately, the linearization procedure is often far more complicated than this. Suppose that instead, we wish to solve the fourth-order problem $q_t = \nabla\cdot(q^\alpha\nabla\Delta q)$ in two dimensions. If we need to calculate only $\mathbf{G}(Q)$, then it would be natural to proceed in steps: first approximate Δq on each cell, then take its gradient on cell edges, multiply by q, and finally compute the divergence. Throughout this process, we use only small, easily-implemented stencils.

However, to determine $\mathbf{G}'[R]$ explicitly as above, we would have to write out $\mathbf{G}(R)$ in terms of its components. With a typical discretization based on a 5-point Laplacian stencil, the value $\mathbf{G}(R)_i$ depends on R's value in 13 different cells. Calculating the Jacobian in this case requires formidable bookkeeping skills, both on paper and when translating results into code. The situation rapidly worsens if we include another spatial dimension, incorporate a stronger nonlinearity, or expand to a system of PDEs.

Furthermore, any results we obtain are tethered to particular choices of derivative stencils.

The amount of problem-specific work required to determine the Jacobian is impractical and well beyond any estimates for what is reasonable to pass along to the end user. Fortunately, there are much better approaches to the linearization process.

3.2. The Jacobian-free Newton–Krylov method. In many cases, $\mathbf{G}'[R]$ can be approximated by

$$\mathbf{G}'[R](P) \approx \frac{\mathbf{G}(R+\epsilon P) - \mathbf{G}(R)}{\epsilon} \tag{3.11}$$

sufficiently well for use in Newton's method. This approach is typically referred to as the Jacobian-free Newton–Krylov (JFNK) method. It is reviewed in detail in [7], which among other things addresses suitable automated procedures for choosing ϵ.

In terms of per-problem human effort, JFNK is essentially optimal. No additional information needs to be supplied for the linearization process, as the approximation of $\mathbf{G}'$ is determined entirely in terms of $\mathbf{G}$. It also avoids the issue of determining boundary conditions for $\mathbf{G}'$. Hence, JFNK should be attempted as the first option for any new problem.

The major weakness of JFNK arises from the fact that division by ϵ in (3.11) typically incurs a loss of roughly $\log_{10} \epsilon$ digits of precision. (As a simple example of this principle, one can examine the accuracy of $\cos(1) \approx (\sin(1+\epsilon) - \sin(1))/\epsilon$ as ϵ varies.)

This loss is incurred in addition to losses arising from discrete differential operators appearing in $\mathbf{G}$, and the combined effects can be overwhelming on fine grids, particularly when high-order derivatives are involved. For example, if $\Delta x = 10^{-2}$ for a fourth-order problem with $\mathcal{O}(1)$ coefficients and $\epsilon = 10^{-6}$, then we expect to lose about 8 digits of precision when evaluating $\mathbf{G}$ and another 6 when evaluating (3.11), eliminating 14 of the possible 16 digits available at double precision.

Consequently, it is important to have an option other than JFNK at our disposal.

3.3. Continuous-space linearization. At a very high level, the integration strategy of §3.1 separates into three steps: discretize in time, discretize in space, and linearize (by way of Newton's method). If we change the order of the first two steps, the linearization procedure remains unchanged, but what if we interchange the second and third?

We return to (3.1), which immediately follows discretization in time in §3.1. Rearranging shows that q^{n+1} solves the equation

$$\mathbf{m}(q^{n+1}) \equiv q^{n+1} - q^n - \Delta t \mathbf{g}(q^{n+1}) = 0. \tag{3.12}$$

Although $\mathbf{m}$ in this case is a nonlinear differential operator, we can at least formally apply Newton's method in the exact same way as before. In fact, the only difference in terms of notation is that capital letters become lowercase.

To be specific, however, we generate a sequence of iterates $r^{[0]}, r^{[1]}, r^{[2]}, \dots$ that will (hopefully) converge to q^{n+1}. To obtain $r^{[k+1]}$, we first determine the perturbation $p^{[k]}$ by forming a linear approximation of $\mathbf{m}$ about $r^{[k]}$ and determining $r^{[k]} + p^{[k]}$ at which it vanishes. We then set $r^{[k+1]} \equiv r^{[k]} + p^{[k]}$. The equation we need to solve at each iteration is

$$\mathbf{m}'[r^{[k]}](p^{[k]}) = -\mathbf{m}(r^{[k]}). \tag{3.13}$$

Here $\mathbf{m}'[r^{[k]}]$ is the linearization of $\mathbf{m}$ about $r^{[k]}$. Although $\mathbf{m}$ is a continuous operator in this case, the usual definition of a linearization applies just the same:

$$\mathbf{m}'[r](p) \equiv \lim_{\epsilon \to 0} \frac{\mathbf{m}(r + \epsilon p) - \mathbf{m}(r)}{\epsilon}. \tag{3.14}$$

Only now do we discretize in space. The equation (3.13) becomes

$$\operatorname{disc}(\mathbf{m}')[R^{[k]}](P^{[k]}) = -\mathbf{M}(R^{[k]}), \tag{3.15}$$

which is a fully discrete linear system of equations. Compared to (3.4), the only difference is that $\mathbf{M}'$ has been replaced by $\operatorname{disc}(\mathbf{m}')$.

As before, in order to solve (3.15) with an iterative method, we must be able to evaluate $\operatorname{disc}(\mathbf{m}')[R](P)$ for any iterate and perturbation. From (3.12), it follows that

$$\mathbf{m}'[r](p) = p - \Delta t \mathbf{g}'[r](p). \tag{3.16}$$

Based on any reasonable notion of discretization,

$$\operatorname{disc}(\mathbf{m}')[R](P) = P - \Delta t \operatorname{disc}(\mathbf{g}')[R](P), \tag{3.17}$$

and the implementation of $\operatorname{disc}(\mathbf{m}')$ reduces to that of $\operatorname{disc}(\mathbf{g}')$.

The implementation of $\operatorname{disc}(\mathbf{g}')$ must be provided by the user and replaces that of $\mathbf{G}'$ from the previous section. The difference is that the linearization in continuous space is much easier to carry out by hand, as we demonstrate below with several examples. For a more rigorous but theoretical perspective, one may consult [2].

3.3.1. Example: Porous medium equation. Consider the porous medium equation once again, only this time in its dimension-independent form

$$q_t = \Delta(q^\alpha). \tag{3.18}$$

Then

$$\mathbf{g}'[r](p) = \lim_{\epsilon \to 0} \frac{\Delta(r + \epsilon p)^\alpha - \Delta(r^\alpha)}{\epsilon}. \tag{3.19}$$

Pulling the Laplacian to the front and expanding $(r + \epsilon p)^\alpha$ in a Taylor series in ϵ, we obtain

$$\begin{aligned} \mathbf{g}'[r](p) &= \Delta \left(\lim_{\epsilon \to 0} \frac{r^\alpha + \epsilon \alpha r^{\alpha-1} p + \mathcal{O}(\epsilon^2) - r^\alpha}{\epsilon} \right) && (3.20) \\ &= \alpha \Delta(r^{\alpha-1} p). && (3.21) \end{aligned}$$

In the 1-dimensional case, using a centered difference to discretize (3.21) produces an operator $\operatorname{disc}(\mathbf{g}')$ that agrees exactly with the $\mathbf{G}'$ that we determined in (3.10); changing the order of spatial discretization and linearization leads to no difference in the end result. But unlike (3.10), (3.21) is suitable both for the multidimensional case and for the task of changing derivative stencils.

One can argue that for an equation this simple, the discrete-space form of Newton's method can also be adapted to more general situations without too much difficulty. The advantage of the continuous-space formulation becomes more evident as we deal with problems of greater complexity.

3.3.2. Example: A fourth-order problem. For the next example, consider

$$q_t = \nabla \cdot (q^\alpha \nabla \Delta q), \tag{3.22}$$

a fourth-order analogue to the porous medium equation. (To keep the correspondence closer, we might have used the alternate form of the porous medium equation $q_t = \nabla \cdot (q^\alpha \nabla q)$, but at the expense of more detailed hand calculations.)

The linearization process is more involved than it was for the porous medium equation, but still not difficult:

$$\mathbf{g}'[r](p) = \lim_{\epsilon \to 0} \frac{\nabla \cdot ((r+\epsilon p)^\alpha \nabla \Delta (r+\epsilon p)) - \nabla \cdot (r \nabla \Delta r)}{\epsilon} \tag{3.23}$$

$$= \nabla \cdot \lim_{\epsilon \to 0} \frac{\epsilon(\alpha r^{\alpha-1} p \nabla \Delta r + r^\alpha \nabla \Delta p) + \mathcal{O}(\epsilon^2)}{\epsilon} \tag{3.24}$$

$$= \nabla \cdot (\alpha r^{\alpha-1} p \nabla \Delta r + r^\alpha \nabla \Delta p). \tag{3.25}$$

When evaluating disc$(\mathbf{g}')[R](P)$ computationally, we are free to evaluate (3.25) in the same sequence of steps (Laplacian, gradient, multiplication, and divergence) described at the end of §3.1. Furthermore, this implementation can be as robust to changes in derivative stencils as we choose to make it.

3.3.3. Example: A system of equations. The procedure we have demonstrated so far works without modification for systems of PDEs. We demonstrate this for a simplified model of thin film/surfactant interaction, as explored in [14], from which we will use exact solutions in §6.

The model system, in vector form, is

$$q_t = \begin{pmatrix} q_1 \\ q_2 \end{pmatrix}_t = \begin{pmatrix} \nabla \cdot (\frac{1}{2} q_1^2 \nabla q_2) \\ \nabla \cdot (q_1 q_2 \nabla q_2) \end{pmatrix} = \begin{pmatrix} \mathbf{g}_1(q) \\ \mathbf{g}_2(q) \end{pmatrix} = \mathbf{g}(q). \tag{3.26}$$

The components of $\mathbf{g}$ may be linearized independently, the only new complication being that their arguments are now vectors.

Working through the details for $\mathbf{g}_1$, we find that

$$\mathbf{g}_1'[r](p) = \lim_{\epsilon \to 0} \frac{\mathbf{g}_1(r+\epsilon p) - \mathbf{g}_1(r)}{\epsilon} \tag{3.27}$$

$$= \nabla \cdot \lim_{\epsilon \to 0} \frac{\frac{1}{2}(r_1 + \epsilon p_1)^2 \nabla (r_2 + \epsilon p_2) - \frac{1}{2} r_1^2 \nabla r_2}{\epsilon} \tag{3.28}$$

$$= \nabla \cdot \left(\frac{1}{2} r_1^2 \nabla p_2 + r_1 p_1 \nabla r_2 \right). \tag{3.29}$$

The calculations for $\mathbf{g}_2$ are similar, resulting in

$$\mathbf{g}_2'[r](p) = \nabla \cdot ((p_1 r_2 + r_1 p_2) \nabla r_2 + r_1 r_2 \nabla p_2). \tag{3.30}$$

4. Boundary conditions. So far, we have glossed over the fact that any discretized differential operator must incorporate boundary conditions. This complication applies to $\mathbf{G} = \text{disc}(\mathbf{g})$ most obviously, which must take into account the boundary conditions of the problem (2.1). But it also applies to $\mathbf{G}'$ with the added complication that the appropriate boundary conditions are not immediately obvious.

In this section, we determine the proper boundary conditions for use with Newton's method. We also illustrate a strategy for implementing all the necessary boundary conditions in a way that decouples from operator discretization and allows common boundary conditions to be implemented in a reusable fashion.

4.1. Boundary conditions in Newton's method. Throughout the course of Newton's method, we must evaluate differential operators applied to iterates and perturbations, as we see in (3.4) or its close relatives (3.13) and (3.15). The continuous-space form (3.13) is easier to work with than the others, keeping us closer to the boundary conditions of the original problem.

On the right side of (3.13), we see $-\mathbf{m}(r)$. From its definition (3.12), $\mathbf{m}$ corresponds to the evaluation of $\mathbf{g}$ at time t_{n+1}. Therefore, it acquires the boundary condition $\mathbf{b}(t_{n+1}, r) = 0$ from the original problem (2.1). By another line of reasoning, each iterate r is itself a guess at q^{n+1}, so it should inherit the same boundary conditions as q^{n+1}.

On the left side of (3.13), we see $\mathbf{m}'[r](p)$. In order to determine the appropriate boundary condition for $\mathbf{m}'[r]$, we return to its definition in (3.14). Both terms in the numerator are applications of $\mathbf{m}$, so $\mathbf{b}(t_{n+1}, r + \epsilon p) = 0$ and $\mathbf{b}(t_{n+1}, r) = 0$ independently. Defining $\tilde{\mathbf{b}} = \mathbf{b}(t_{n+1}, \cdot)$ for simplicity, we find that

$$\lim_{\epsilon \to 0} \frac{\tilde{\mathbf{b}}(r + \epsilon p) - \tilde{\mathbf{b}}(r)}{\epsilon} = 0. \tag{4.1}$$

And therefore $\mathbf{m}'[r](p)$ carries the boundary condition

$$\mathbf{b}'[r](p) = 0. \tag{4.2}$$

This is likewise the boundary condition associated with $\mathbf{g}'[r]$.

4.2. Ghost cells. The question remains of how to implement boundary conditions in a way that is suitably flexible, while minimizing the work relegated to the end user. Based on §3, the user will need to supply routines that directly evaluate $\mathbf{G}(R)$ and $\mathbf{G}'[R](P)$. From the previous section, we need to determine how to incorporate the boundary conditions $\mathbf{b}(t, q) = 0$ into $\mathbf{G}$ and $\mathbf{b}'[r](p) = 0$ into $\mathbf{G}'$. In the discussion that follows, we focus on applying $\mathbf{b}(t, q) = 0$ to $\mathbf{G}$, but the argument is general enough address both cases.

Sufficiently far from the boundary, $\mathbf{G}$ need not depend on boundary conditions at all. We may use identical derivative stencils on all such cells, resulting in discrete formulae that are not too complex. However, discretizing near the boundary requires asymmetric derivative stencils to incorporate boundary values and avoid cells that lie outside of the domain.

As an example, consider a diffusion equation with constant Dirichlet boundary conditions: $\mathbf{g}(q) = q_{xx}$ and $\mathbf{b}(t, q) = \beta$. At any interior cell $\mathcal{C}_i$, we may use a centered difference to produce

$$\mathbf{G}(Q)_i = \frac{Q_{i-1} - 2Q_i + Q_{i+1}}{\Delta x^2}. \tag{4.3}$$

However, at the leftmost cell of the domain (say it is $\mathcal{C}_1$), we must incorporate the boundary condition and use an irregular stencil. One such option, which we might derive from Taylor expansions, is

$$\mathbf{G}(Q)_1 = \frac{16\beta - 25Q_1 + 10Q_2 - Q_3}{5\Delta x^2}. \tag{4.4}$$

As presented, though, the derivation of (4.4) depends on the operator $\mathbf{g}$ itself and will introduce special cases into any implementation of $\mathbf{G}$.

Ghost cells, or equivalently ghost points for finite differences, provide a better solution. Rather than considering irregular derivative stencils, we introduce a cell $\mathcal{C}_0$ to the left of the domain and extrapolate a value for Q there. Using a cubic extrapolant (suitable to maintain a $\mathcal{O}(\Delta x^2)$ approximation to q_{xx}) we find that

$$Q_0 \equiv \frac{16\beta - 15Q_1 + 5Q_2 - Q_3}{5}. \tag{4.5}$$

Remarkably, when we evaluate $\mathbf{G}(Q)_1$ using the interior stencil (4.3) and this value of Q_0 it reduces to (4.4). Not only is the extrapolation procedure independent of $\mathbf{g}$, but it allows $\mathbf{G}$ to be implemented in a way that only considers the simpler, interior case.

Summarizing in more generality, we consider $\mathbf{b}(t, q)$ to define a set of conditions for an extrapolating polynomial at the boundary of the domain. Before applying $\mathbf{G}$, we use these conditions and some number of interior cell values — whatever is suitable to obtain the required polynomial degree — to extrapolate values of Q into the ghost cells. Afterwards, we implement $\mathbf{G}$ treating all cells as though they are fully interior.

4.3. Practical use. The boundary condition (4.2) is more general than is necessary for most use cases. More often than not, boundary conditions may be expressed in the simpler form

$$\mathbf{b}(t, q) = \mathbf{b}_{\text{linear}}(q) - h(x, t) \tag{4.6}$$

for some function h. For example, $\mathbf{b}_{\text{linear}}(q) = q$ will specify Dirichlet conditions, while $\mathbf{b}_{\text{linear}}(q) = q_x$ specifies Neumann conditions. We can likewise specify Robin conditions in this fashion, or by using vector-valued $\mathbf{b}_{\text{linear}}$ and h, we may specify multiple normal derivatives as required for fourth-order problems.

In this case, the boundary condition $\mathbf{b}'[r](p)$ that corresponds to $\mathbf{g}'$ is simply $\mathbf{b}_{\text{linear}}(p) = 0$. That is, p satisfies the homogeneous version of the boundary conditions for q.

Altogether, this situation lets us provide an especially simple interface for boundary conditions. The user only needs to specify the type of boundary conditions used on each side of the grid, selecting from a list of those supported, and provide a subroutine that supplies the values for $h(x, t)$. Boundary conditions for $\mathbf{G}'$ require no special consideration, as they use the exact same type of conditions as $\mathbf{G}$ with $h(x, t)$ replaced by zero.

5. Implementation and interface. With the mathematical details in place, we now turn to assembling them into a computational implementation. The focus at this stage is to maintain a clean separation into library routines and interface routines.

The library routines are fully implemented and carry out as much generic logic as possible. They let the user reuse large amounts of code thereby saving work, and they can be tested across a variety of different problems, ensuring that they are as bug-free as possible.

The interface routines, on the other hand, are defined only by their calling signature; they must be written by the user. They carry out concise, well-defined tasks that are closely related to the mathematical problem being solved, such as applying the discrete version of the PDE's spatial differential operator or providing boundary data at each time step.

Carefully distinguishing between the two is important, both to ensure that generic logic need not be reimplemented on a per-problem basis, and to establish that user-developed code will correspond to meaningful abstractions with clear relations to the underlying PDE problem. Naming is specific to our implementation, but the central ideas of what code is exposed and what is not are broadly relevant, even to languages in which the code's structure could be very different.

5.1. Interface routines. Below is an overview of the routines that must be provided by the user in order to solve a particular problem. Writing these all from scratch may be a daunting task, and one of the goals of the test cases in §6.1 is to provide a wide range of sample implementations, such that much of the work can be handled by copy-and-paste from a similar problem.

The first few routines are a part of Clawpack itself, while the remainder are specific to our implicit integration package.

`setrun` (Clawpack): The Python routine `setrun` is the means of passing configuration parameters to the program. It produces two objects: `clawdata`, which contains base Clawpack parameters, and `probdata`, which contains user-defined parameters. These objects are equipped with methods to write themselves to data files of particular formats, and those files are read by the Fortran program (see `setprob`, below) that implements the PDE solver.

The base Clawpack parameters consist of quantities fundamental to the problem (grid coordinates, number of cells, initial and final times, etc.) and configuration parameters for its hyperbolic solvers, though the latter are provided with default values behind the scenes and can largely be ignored if unused.

We use `probdata` to define a number of core parameters that are central to the implicit integration library. The user is free to add additional parameters here as needed.

`setprob` (Clawpack): This is a Fortran routine used to read the user-defined parameters written by `setrun`. The parameters are communicated to subroutines by way of common blocks.

`qinit` (Clawpack): This routine sets the initial conditions for the problem. It executes after `setprob`, so it may access any configuration parameters that have been placed in common blocks.

`apply_pde_operator`: Given an array of solution values, this routine computes the discretized PDE operator $\mathbf{G}$. This routine executes after ghost cells have been filled as in §4.2, so no special treatment of boundary cases is required.

`apply_linearized_pde_operator`: Given arrays of values for a Newton iterate R and perturbation P, this routine computes $\mathbf{G}'[R](P)$. Either of the approximation procedures discussed in §3 produce a suitable implementation of this method, and a library implementation of this is provided for use with JFNK. As with `apply_pde_operator`, ghost cell values for the iterate and perturbation have been filled prior to this routine's execution.

`apply_bcs`: Given the array of solution values and the current time, this routine fills ghost cell values in accordance with the boundary conditions. A pre-implemented version is provided for use with the simpler `set_implicit_boundary_data`, described below.

`apply_linearized_bcs`: Given arrays of iterate and perturbation values for Newton's method, this routine fills the ghost cells of the perturbation array based on the linearized boundary conditions. It also has a pre-implemented version for use with `set_implicit_boundary_data`.

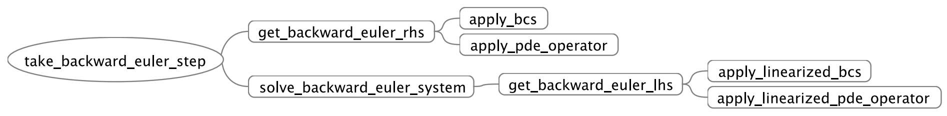

FIG. 5.1. *Calling tree for backward Euler integration.*

`set_implicit_boundary_data`: This routine defines boundary data on each cell interface on the boundary of the domain. For each component of the PDE's solution, two values may be provided at each interface. One or both of these values may be ignored (and hence need not be specified) depending on the type of boundary condition used.

In order to use this routine, one must select a predefined type of boundary condition for each edge of the grid. The types are specified by configuration parameters in `setrun`: `x_low_bc_type` and `x_high_bc_type`, and additionally `y_low_bc_type` and `y_high_bc_type` in 2 dimensions.

Each of these parameters is a list of strings, one for each equation in the PDE system. The types of boundary conditions currently supported are:

(i) `"p"`: Periodic boundary conditions. Must be coordinated between lower and upper ends of the domain.

(ii) `"n"`: None, e.g. for outflow conditions. Ghost cells are filled with extrapolated data.

(iii) `"0"` or `"1"`: One derivative is specified at each boundary interface, with the string specifying the order of the derivative. Order 0 refers to direct function values.

(iv) `"01"`, `"02"`, `"03"`, `"12"`, `"13"`, or `"23"`: Two derivatives are specified at each boundary interface. For example, `"03"` means that q is specified, as well as q_{xxx} (for the left or right edge of the domain) or q_{yyy} (for the top or bottom).

5.2. Integration library routines. For the routines that perform integration over each time step, we focus on showing how their logic decomposes to the interface routines described above. In keeping with the reasoning of §2.3, we discuss backward Euler only. However, library implementations of forward Euler and Crank–Nicolson are also provided, and any of the three schemes may be selected with a configuration parameter.

For reference, the tree of subroutine calls is depicted in Figure 5.1. The four internal routines are outlined below. The notation $x \leftarrow y$ indicates that a computational variable x is overwritten by the value of y.

`take_backward_euler_step`$(t_n, \Delta t, Q)$: Takes $Q = Q^n$ upon input, and returns with $Q = Q^{n+1}$ obtained from backward Euler integration.

Pseudocode:

```
$R \leftarrow Q^n$
**for** $k = 1, maxiterations$ **do**
    $P \leftarrow -\mathbf{M}(R)$, via get_backward_euler_rhs
    $P \leftarrow$ solution of $\mathbf{M}'[R](P) = -\mathbf{M}(R)$, via solve_backward_euler_system
    $R \leftarrow R + P$
    **if** $\|P\| < tolerance$ **then**
        $Q^{n+1} \leftarrow R$
        **return**
    **end if**
**end for**
```

`get_backward_euler_rhs`$(t_n, \Delta t, Q^n, R, Z)$: Sets $Z = -\mathbf{M}(R)$.

Pseudocode:

```
Fill ghost cells of $R$ at $t_{n+1}$, via apply_bcs
$Z \leftarrow \mathbf{G}(R)$, via apply_pde_operator
$Z \leftarrow -(R - Q^n - \Delta t Z)$
```

`solve_backward_euler_system`$(t_n, \Delta t, R, P)$: On input, P stores $-\mathbf{M}(R)$. Overwrites P with the solution of $\mathbf{M}'[R](P) = -\mathbf{M}(R)$.

Implementation is heavily dependent on the iterative solver being used. However, all problem-specific details reduce to calculation of the left-hand side of the system via `get_backward_euler_lhs`.

BiCGStab is the only solver currently implemented. Unfortunately, use of ghost cells and OpenMP parallelization significantly complicate the use of preexisting implementations. We currently use a customized version, adapted from [15].

`get_backward_euler_lhs`$(t_n, \Delta t, R, P, Z)$: Sets $Z = \mathbf{M}'[R](P)$.

Pseudocode:

```
Fill ghost cells of $P$ at $t_{n+1}$, via apply_linearized_bcs
$Z \leftarrow \mathbf{G}'[R](P)$, via apply_linearized_pde_operator
$Z \leftarrow P - \Delta t Z$
```

5.3. Boundary conditions. The two boundary condition interface routines `apply_bcs` and `apply_linearized_bcs` are sufficiently general to support any conditions one might wish to apply, but they are lacking in specificity. As mentioned earlier, we provide a number of pre-specified boundary conditions for which the interface is reduced to a simpler combination of configuration parameters and the subroutine `set_implicit_boundary_data`.

The boundary conditions we support are notable in that they are of the form (4.6) and $\mathbf{b}_{\text{linear}}$ is fixed. Furthermore, the procedure for filling ghost cells operates independently on each solution component and on each 1-dimensional slice of data. In these situations, like (4.5), the value of Q in a ghost cell may be calculated explicitly as a linear combination of boundary values at the nearest boundary interface and of interior solution values along the same 1-dimensional slice.

By way of `set_implicit_boundary_data`, the user specifies up to two boundary values β_1 and β_2 on each boundary interface. Then for each ghost cell $\mathcal{C}_k$, the corresponding ghost value may be computed as

$$Q_k = \sum_{i=1}^{2} a_i \beta_i + \sum_j c_j Q_j, \tag{5.1}$$

where the summation includes a small number of interior cells from the same 1-dimensional slice as $\mathcal{C}_k$. (This formulation accommodates periodic boundary conditions by using cells on the opposite side of the domain, though in practice the computational handling of that case involves its own fairly simple logic.) The case-specific logic consists of determining the appropriate coefficients a_i and c_j. The only important factor for implementation is that a linear system of equations must be solved to determine each set of coefficients, and the code should be structured to solve any given system no more than once.

Ghost cells in the corner of the domain are not filled in by this scheme. However, they may be filled by using polynomials fit to the data in nearby cells.

6. Testing. For both 1- and 2-dimensional versions of the library, we provide a suite of five sample applications as test cases. These are problems with known analytical solutions, allowing us to test specific expectations for convergence under refinement.

The tests are designed to be run continuously. They ensure not only that the code functions as intended at any given point in time, but that future changes do not introduce new bugs. They are built on top of Python's `unittest` module, making it easy to run them in batches before any new changes are submitted to the repository.

Furthermore, the tests serve as a form of implicit documentation. Each one incorporates a fully-implemented application code, and the corresponding instances of the interface routines may serve as templates for developing new applications.

6.1. Test problems. Test cases have been chosen to ensure that fourth-order problems, nonlinear problems, and systems of equations have all been addressed, while including simpler linear cases to facilitate debugging and feature development (most notably boundary conditions). The cases are as follows.

Second-order, linear:

$$q_t = \Delta q, \qquad q(x,t) = e^{-Nt} \prod_{i=1}^{N} \sin(x_i) \tag{6.1}$$

Second-order, nonlinear:

$$q_t = \Delta(q^2), \qquad q(x,t) = \max\left\{0, t^{-\alpha}\left(\Gamma - \frac{\alpha}{4N}\frac{\|x\|^2}{t^{2\alpha/N}}\right)\right\}, \tag{6.2}$$

where $\alpha = 1/(1+2/N)$.

Mixed first- and second-order, nonlinear system:

$$\begin{pmatrix} q_1 \\ q_2 \end{pmatrix}_t = \begin{pmatrix} \nabla\cdot(\frac{1}{2}q_1^2\nabla q_2) \\ \nabla\cdot(q_1 q_2 \nabla q_2) \end{pmatrix}, \tag{6.3}$$

$$N = 1 \ : \ q(x,t) = \begin{pmatrix} 2\rho \\ \frac{1}{6t^{1/3}}(1-\rho) \end{pmatrix}, \text{ where } \rho = x/t^{1/3}, \tag{6.4}$$

$$N = 2 \ : \ q(x,t) = \begin{pmatrix} 2\rho^2 \\ -\frac{1}{8\tau^{1/2}}\log\rho \end{pmatrix}, \text{ where } \rho = \|x\|/t^{1/4}. \tag{6.5}$$

Fourth-order, linear:

$$q_t = \nabla\cdot(\nabla\Delta q), \qquad q(x,t) = e^{-N^2 t}\prod_{i=1}^{N}\sin(x_i) \tag{6.6}$$

Fourth-order, nonlinear:

$$q_t = \nabla\cdot(q\nabla\Delta q), \qquad q(x,t) = \max\left\{0, \frac{1}{8(N+2)\tau^N}\left(\Gamma^2 - \frac{\|x\|^2}{\tau^2}\right)\right\}, \tag{6.7}$$

where $\tau = ((N+4)t)^{1/(N+4)}$.

For the nonlinear problems, care must be taken to stay away from corners and potential negative values for any value that serves as an effective diffusion coefficient.

Table 6.1
Convergence results for test cases, to two decimal places.

	$\mathbb{R}^1$			$\mathbb{R}^2$		
Application	L^1	L^2	L^∞	L^1	L^2	L^∞
Second-order, linear	1.93	1.93	1.93	2.01	2.00	1.99
Second-order, nonlinear	1.99	1.98	1.98	2.00	2.00	2.00
Second-order, nonlinear system	1.99	1.99	1.91	2.00	2.00	1.92
Fourth-order, linear	1.99	1.99	1.98	2.01	2.01	2.00
Fourth-order, nonlinear	1.98	1.98	1.98	2.00	2.01	2.00

6.2. Methodology. For each test case, we implement operators using approximations that are second-order in space. We separately test problem-specific versions of `apply_linearized_pde_operator` and a central version that uses approximate linearization.

To define a test, we select a fixed end time t_{final} and choose a set of successively refined Δt-values that evenly subdivide it. We pick Δx corresponding to the largest value of Δt and then generate a set of $(\Delta t, \Delta x)$ pairs that keep $\Delta x/\Delta t$ as nearly constant as possible.

We then perform test runs up to a fixed t_{final} for each $(\Delta t, \Delta x)$ pair using Crank–Nicolson integration. For each run, we compare the numerical solution at t_{final} to $q_{\text{exact}}(x, t_{\text{final}})$ and measure error in the L^1, L^2 and L^∞ norms. The error in each norm is fitted to a curve of the form $\alpha \Delta t^{-\beta}$, and we require that $\beta \geq 1.9$ for consistency with the second-order accuracy of Crank–Nicolson. A snapshot of results is provided in Table 6.1, confirming that this expectation is met.

6.3. An example from literature. We conclude our tests with a look at a problem from literature, a study of fingering phenomena in a thin film/surfactant model [18]. We can make only a qualitative comparison of results, so in particular this problem does not yield another automated test. However, the underlying PDE problem is much more complicated than the automated test problems, and it is more indicative of the types of models one would encounter in application studies.

The fingering study is based on a PDE system for film height $h(x,t)$ and surfactant concentration $\Gamma(x,t)$. For brevity we skip to the nondimensionalized form:

$$h_t + \nabla \cdot \left(\frac{1}{2}\nabla\sigma\, h^2\right) = \nabla \cdot \left(\frac{1}{3}h^3(\beta\nabla h - \kappa\Delta\nabla h)\right) \tag{6.8}$$

$$\Gamma_t + \nabla \cdot (h\Gamma\nabla\sigma) = \frac{1}{2}h^2\Gamma(\beta\nabla h - \kappa\Delta\nabla h) + \delta\Delta\Gamma. \tag{6.9}$$

The parameter β incorporates strength of gravitational effects, κ specifies strength of capillarity, and δ is the scaled diffusivity of the surfactant.

Initial conditions are specified by

$$h(x,y,0) = (1-x^2+b)H(1-x) + bH(x-1) + P(x,y), \quad \Gamma(x,y,0) = H(1-x), \tag{6.10}$$

where $H(x) = \frac{1}{2}(1+\tanh(20x))$ is a smooth approximation to the Heaviside function, and the perturbation of the film is of the form

$$P(x,y) = Ae^{-B(x-1)^2}\sum_{i=1}^{4} C_i\cos(k_i y). \tag{6.11}$$

The scalar parameters are $b = 0.05$, $A = 0.01$, $B = 5$; $C_1 = C_2 = C_3 = 1$ and $C_4 = 0.5$; and $k_1 = 2$, $k_2 = 5$, $k_3 = 7$, and $k_4 = 20$.

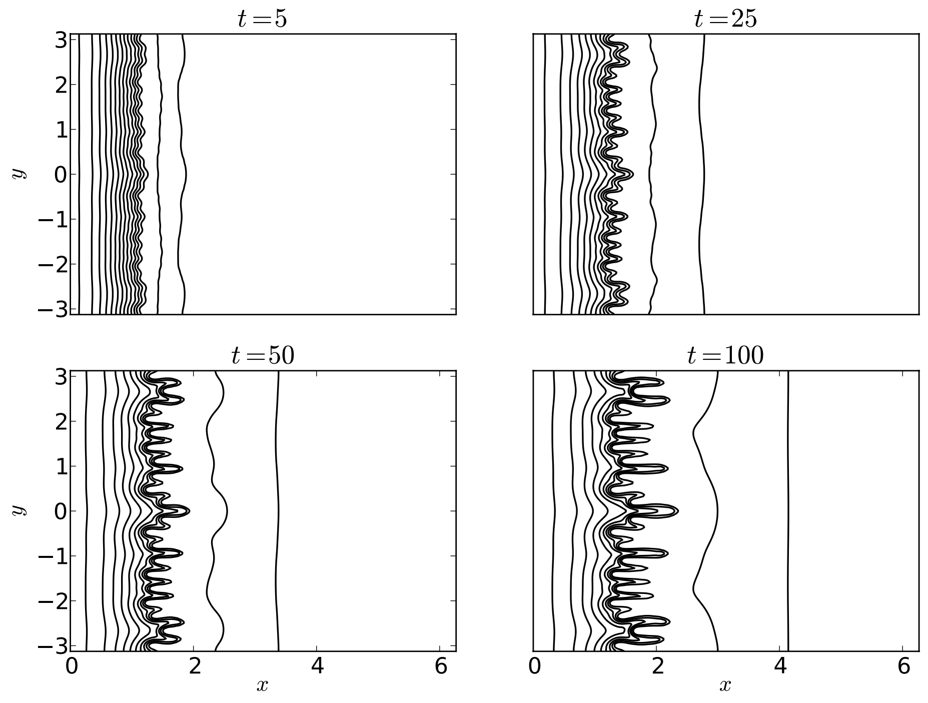

FIG. 6.1. *Contour plots of simulated results from the problem described in* §*6.3. Contour values are multiples of* $1/15$ *on the interval* $[0,1]$*.*

The authors report that their calculations are performed on the rectangle $[0,2\pi]\times[0,2\pi]$, but in order to reproduce their results, we found that we needed to shift the y-interval to $[-\pi,\pi]$. The computational grid size is 200×200.

Periodic boundary conditions are imposed in y, zero-flux conditions at $x=0$, and the conditions $h(2\pi,y,t)=b$, $h_x(2\pi,y,t)=0$, $\Gamma(2\pi,y,t)=0$ at $x=2\pi$.

Results from our code, using JFNK for linearization, are depicted in Figure 6.3. Similarity to Figure 19 of [18] is exceptionally strong. Furthermore, we have achieved these results using $\Delta t=0.1$, based on an estimate of the desired accuracy rather than any restrictions due to stability. By contrast, [18] used a customized numerical scheme and reported stability issues that constrained time step size to the order of 10^{-5}.

7. Conclusions and future directions. We have demonstrated that the process of developing nonlinear implicit solvers need not be overly complicated, even while maintaining full support for PDE systems. Furthermore, common boundary conditions can be implemented in a way that makes them largely reusable, requiring that a solver for a particular problem need only provide information that truly is problem-specific.

We are hopeful that the implementation made available alongside this paper is not only suitable to illustrate its concepts and establish their validity, but to fill the ongoing needs of application developers as well. It is important to keep in mind the less obvious benefits of using a pre-existing implementation — things like the build procedure, data format, test infrastructure, and cumulative debugging. These are all issues that one can handle independently given enough attention, but they are more time-consuming and subtler than one generally expects from the outset. At the same time, we have hopefully presented the underlying ideas clearly enough to bring the development of these sorts of solvers into the realm of "you could do this yourself if you really wanted to."

Moving forward, the most obvious target for further development is performance. A handful of simple performance considerations are outlined in documentation that accompanies the code, but there is definitely room for more sophisticated work. In particular, based on experience with the test cases, speed on fourth-order problems with even a modest amount of refinement leaves much to be desired. Preconditioning is the most obvious approach to pursue — indeed something that should come to mind whenever iterative solvers are mentioned — and multigrid techniques are especially appealing in this regard. The challenge is to implement them in such a way that they require no additional work to the end user.

A number of other improvements are easy to pick out, more in the spirit of incremental development. Support for more boundary conditions would be useful, as would a suitable implementation of GMRES, serving as backup when BiCGStab fails to converge. (It is worth noting, however, that every convergence failure of BiCGStab so far in this code has been the result of a programming error.) Meanwhile, loftier goals include a symbolic interface, or built-in support for the modified boundary conditions that should be used with operator-splitting methods (e.g. if using Clawpack's hyperbolic solvers) or high-accuracy integration schemes.

In any case, future development will likely depend on outside interest and feedback. If you think the code may suit your needs, then download it and give it a try. If it helps you solve your problem, please let us know. And if it doesn't, please let us know that too, and why.

8. Acknowledgements. Many thanks are due to Randall LeVeque, both for valuable feedback and much of the education that made this work possible. We are also grateful to Cameron Conti, Eric Autry, and Gregory Kronmiller for testing an earlier (and far less polished) version of the Clawpack library [8].

REFERENCES

[1] A.M. Davis, *Fifteen principles of software engineering*, Software, IEEE, 11 (1994), pp. 94–96.

[2] P. Deuflhard, *Newton methods for nonlinear problems: Affine invariance and adaptive algorithms*, vol. 35, Springer, Berlin, Heidelberg, 2011.

[3] D.L. Donoho, A. Maleki, I.U. Rahman, M. Shahram, and V. Stodden, *Reproducible research in computational harmonic analysis*, Computing in Science & Engineering, 11 (2009), pp. 8–18.

[4] B.D. Edmonstone, O.K. Matar, and R.V. Craster, *Flow of surfactant-laden thin films down an inclined plane*, Journal of Engineering Mathematics, 50 (2004), pp. 141–156.

[5] A. Iserles, *A First Course in the Numerical Analysis of Differential Equations*, Cambridge University Press, Cambridge, UK, 2008.

[6] L.A. Khan and P.L.F. Liu, *Intermediate Dirichlet boundary conditions for operator splitting algorithms for the advection-diffusion equation*, Computers and Fluids, 24 (1995), pp. 447–458.

[7] D.A. Knoll and D.E. Keyes, *Jacobian-free Newton–Krylov methods: A survey of approaches and applications*, Journal of Computational Physics, 193 (2004), pp. 357–397.

[8] G. Kronmiller, E. Autry, and C. Conti, *The effects of spatial and temporal grids on simulations of thin films with surfactant*, SIAM Undergraduate Research Online, 6 (2013).

[9] R.J. LeVeque, *Intermediate boundary conditions for LOD, ADI and approximate factorization methods*, ICASE Report No. 85-21, NASA Langley Research Center, (1985).

[10] R.J. LeVeque, *Finite Volume Methods for Hyperbolic Problems*, Cambridge University Press, Cambridge, UK, 2002.

[11] R.J. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*, SIAM, Philadelphia, PA, 2007.

[12] R.J. LeVeque, M.J. Berger, et al., *Clawpack Software 4.6*, http://www.clawpack.org, 2011.

[13] R.J. Leveque, I. Mitchell, and V. Stodden, *Reproducible research for scientific computing: Tools and strategies for changing the culture*, Computing in Science and Engineering, (2012), pp. 13–17.

[14] E.R. Peterson, *Flow of Thin Liquid Films with Surfactant: Analysis, Numerics, and Experiment*, PhD thesis, North Carolina State University, 2010.

[15] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, PA, 2003.

[16] E.H. Twizell, A.B. Gumel, and M.A. Arigu, *Second-order, L_0-stable methods for the heat equation with time-dependent boundary conditions*, Advances in Computational Mathematics, 6 (1996), pp. 333–352.

[17] H.A. Van der Vorst, *BI-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 13 (1992), p. 631.

[18] M.R.E. Warner, R.V. Craster, and O.K. Matar, *Fingering phenomena associated with insoluble surfactant spreading on thin liquid films*, Journal of Fluid Mechanics, 510 (2004), pp. 169–200.

[19] T.P. Witelski and M. Bowen, *ADI schemes for higher-order nonlinear diffusion equations*, Applied Numerical Mathematics, 45 (2003), pp. 331–351.

[20] Yale Law School Roundtable on Data and Code Sharing, *Reproducible research: Addressing the need for data and code sharing in computational science*, Computing in Science and Engineering, 12 (2010), pp. 8–13.