CHAPTER 8

Initial and Boundary Value Problems of Differential Equations

Our ultimate goal is to solve very general nonlinear partial differential equations of elliptic, hyperbolic, parabolic or mixed type. However, a variety of basic techniques are required from the solutions of ordinary differential equations. By understanding the basic ideas for computationally solving initial and boundary value problems for differential equations, we can solve more complicated partial differential equations. The development of numerical solution techniques for initial and boundary value problems originates from the simple concept of the Taylor expansion. Thus the building blocks for scientific computing are rooted in concepts from freshman calculus. Implementation, however, often requires ingenuity, insight and clever application of the basic principles. In some sense, our numerical solution techniques reverse our understanding of calculus. Whereas calculus teaches us to take a limit in order to define a derivative or integral, in numerical computations we take the derivative or integral of the governing equation and go backwards to define it as the difference.

1. Initial Value Problems: Euler, Runge–Kutta and Adams Methods

The solutions of general partial differential equations rely heavily on the techniques developed for ordinary differential equations. Thus we begin by considering systems of differential equations of the form

$$\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y}) \tag{1}$$

where $\mathbf{y}$ represents the solution vector of interest and the general function $f(t, \mathbf{y})$ models the specific system of interest. Indeed, the function $f(t, \mathbf{y})$ is the primary quantity required for calculating the dynamical evolution of a given system. As such, it will be the critical part of building python codes for solving differential equations. In addition to the evolution dynamics, the initial conditions are given by

$$\mathbf{y}(0) = \mathbf{y}_0 \tag{2}$$

with $t \in [0, T]$. Although very simple in appearance, this equation cannot be solved analytically in general. Of course, there are certain cases for which the problem can be solved analytically, but it will generally be important to rely on numerical solutions for insight. For an overview of analytic techniques, see Boyce and DiPrima [**29**]. Note that the function $f(\mathbf{y}, t)$ is what is ultimately required by python to solve a given differential equation system.

The simplest algorithm for solving this system of differential equations is known as the *Euler method*. The Euler method is derived by making use of the definition of the derivative:

$$\frac{d\mathbf{y}}{dt} = \lim_{\Delta t \to 0} \frac{\Delta \mathbf{y}}{\Delta t}. \tag{3}$$

Thus over a time-span $\Delta t = t_{n+1} - t_n$ we can approximate the original differential equation by

$$\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y}) \quad \Rightarrow \quad \frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{\Delta t} \approx f(t_n, \mathbf{y}_n). \tag{4}$$

The approximation can easily be rearranged to give

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(t_n, \mathbf{y}_n). \tag{5}$$

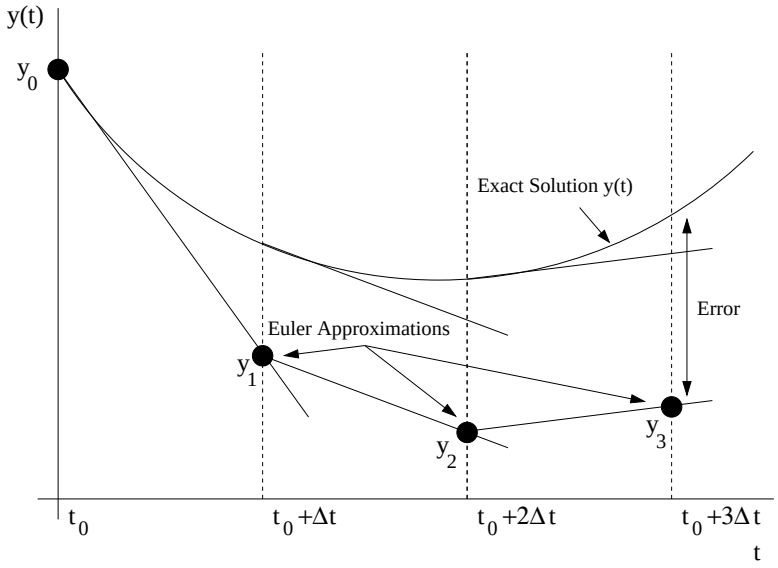

FIGURE 1. Graphical description of the iteration process used in the Euler method. Note that each subsequent approximation is generated from the slope of the previous point. This graphical illustration suggests that smaller steps Δt should be more accurate.

Thus the Euler method gives an iterative scheme by which the future values of the solution can be determined. Generally, the algorithm structure is of the form

$$\mathbf{y}(t_{n+1}) = F(\mathbf{y}(t_n)) \tag{6}$$

where $F(\mathbf{y}(t_n)) = \mathbf{y}(t_n) + \Delta t \cdot f(t_n, \mathbf{y}(t_n))$. The graphical representation of this iterative process is illustrated in Fig. 1 where the slope (derivative) of the function is responsible for generating each subsequent approximation to the solution $\mathbf{y}(t)$. Note that the Euler method is exact as the step-size decreases to zero: $\Delta t \to 0$.

The Euler method can be generalized to the following iterative scheme:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot \phi \tag{7}$$

where the function ϕ is chosen to reduce the error over a single time-step Δt and $\mathbf{y}_n = \mathbf{y}(t_n)$. The function ϕ is no longer constrained, as in the Euler scheme, to make use of the derivative at the left end point of the computational step. Rather, the derivative at the midpoint of the time-step and at the right end of the time-step may also be used to possibly improve accuracy. In particular, by generalizing to include the slope at the left and right ends of the time-step Δt, we can generate an iteration scheme of the following form:

$$\mathbf{y}(t + \Delta t) = \mathbf{y}(t) + \Delta t \left[A f(t, \mathbf{y}(t)) + B f\left(t + P \cdot \Delta t, \mathbf{y}(t) + Q \Delta t \cdot f(t, \mathbf{y}(t))\right)\right] \tag{8}$$

where A, B, P and Q are arbitrary constants. Upon Taylor expanding the last term, we find

$$\begin{aligned} f(t + P \cdot \Delta t, \mathbf{y}(t) + Q \Delta t \cdot f(t, \mathbf{y}(t))) &= f(t, \mathbf{y}(t)) + P \Delta t \cdot f_t(t, \mathbf{y}(t)) \\ &\quad + Q \Delta t \cdot f_{\mathbf{y}}(t, \mathbf{y}(t)) \cdot f(t, \mathbf{y}(t)) + O(\Delta t^2) \end{aligned} \tag{9}$$

where f_t and $f_{\mathbf{y}}$ denote differentiation with respect to t and $\mathbf{y}$, respectively, use has been made of (1), and $O(\Delta t^2)$ denotes all terms that are of size Δt^2 and smaller. Plugging in this last result into

the original iteration scheme (8) results in the following:

$$\begin{aligned}\mathbf{y}(t+\Delta t) &= \mathbf{y}(t)+\Delta t(A+B)f(t,\mathbf{y}(t)) \\ &\quad + PB\Delta t^2\cdot f_t(t,\mathbf{y}(t)) \\ &\quad + BQ\Delta t^2\cdot f_{\mathbf{y}}(t,\mathbf{y}(t))\cdot f(t,\mathbf{y}(t))+O\left(\Delta t^3\right)\end{aligned} \tag{10}$$

which is valid up to $O(\Delta t^2)$.

To proceed further, we simply note that the Taylor expansion for $\mathbf{y}(t+\Delta t)$ gives:

$$\begin{aligned}\mathbf{y}(t+\Delta t) &= \mathbf{y}(t)+\Delta t\cdot f(t,\mathbf{y}(t))+\frac{1}{2}\Delta t^2\cdot f_t(t,\mathbf{y}(t)) \\ &\quad +\frac{1}{2}\Delta t^2\cdot f_{\mathbf{y}}\left(t,\mathbf{y}(t)\right)f\left(t,\mathbf{y}(t)\right)+O\left(\Delta t^3\right).\end{aligned} \tag{11}$$

Comparing this Taylor expansion with (10) gives the following relations:

$$A+B=1 \tag{12a}$$

$$PB=\frac{1}{2} \tag{12b}$$

$$BQ=\frac{1}{2} \tag{12c}$$

which yields three equations for the four unknows A, B, P and Q. Thus one degree of freedom is granted, and a wide variety of schemes can be implemented. Two of the more commonly used schemes are known as *Heun's method* and *modified Euler–Cauchy* (second-order Runge–Kutta). These schemes assume $A=1/2$ and $A=0$, respectively, and are given by:

$$\mathbf{y}(t+\Delta t)=\mathbf{y}(t)+\frac{\Delta t}{2}\left[f(t,\mathbf{y}(t))+f(t+\Delta t,\mathbf{y}(t)+\Delta t\cdot f(t,\mathbf{y}(t)))\right] \tag{13a}$$

$$\mathbf{y}(t+\Delta t)=\mathbf{y}(t)+\Delta t\cdot f\left(t+\frac{\Delta t}{2},\mathbf{y}(t)+\frac{\Delta t}{2}\cdot f(t,\mathbf{y}(t))\right). \tag{13b}$$

Generally speaking, these methods for iterating forward in time given a single initial point are known as *Runge–Kutta methods*. By generalizing the assumption (8), we can construct stepping schemes which have arbitrary accuracy. Of course, the level of algebraic difficulty in deriving these higher accuracy schemes also increases significantly from Heun's method and modified Euler–Cauchy.

1.1. Fourth-order Runge–Kutta. Perhaps the most popular general stepping scheme used in practice is known as the *fourth-order Runge–Kutta method*. The term "fourth-order" refers to the fact that the Taylor series local truncation error is pushed to $O(\Delta t^5)$. The total cumulative (global) error is then $O(\Delta t^4)$ and is responsible for the scheme name of "fourth-order". The scheme is as follows:

$$\mathbf{y}_{n+1}=\mathbf{y}_n+\frac{\Delta t}{6}\left[f_1+2f_2+2f_3+f_4\right] \tag{14}$$

where

$$f_1=f(t_n,\mathbf{y}_n) \tag{15a}$$

$$f_2=f\left(t_n+\frac{\Delta t}{2},\mathbf{y}_n+\frac{\Delta t}{2}f_1\right) \tag{15b}$$

$$f_3=f\left(t_n+\frac{\Delta t}{2},\mathbf{y}_n+\frac{\Delta t}{2}f_2\right) \tag{15c}$$

$$f_4=f\left(t_n+\Delta t,\mathbf{y}_n+\Delta t\cdot f_3\right). \tag{15d}$$

This scheme gives a local truncation error which is $O(\Delta t^5)$. The cumulative (global) error in this case is fourth order so that for $t\sim O(1)$ the error is $O(\Delta t^4)$. The key to this method, as well as any of the other Runge–Kutta schemes, is the use of intermediate time-steps to improve accuracy.

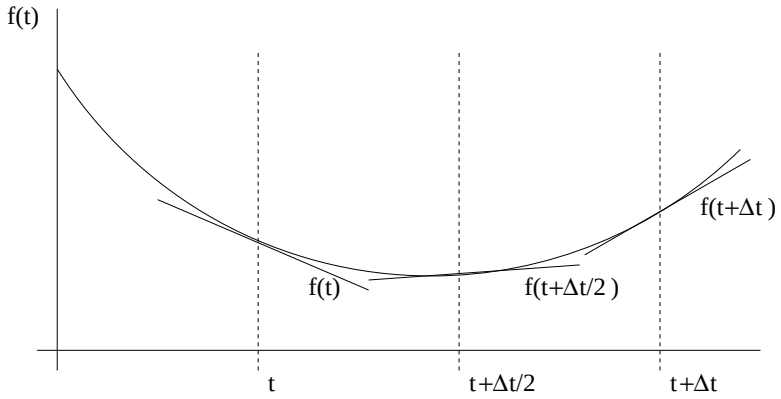

FIGURE 2. Graphical description of the initial, intermediate, and final slopes used in the 4th-order Runge-Kutta iteration scheme over a time Δt.

For the fourth-order scheme presented here, a graphical representation of this derivative sampling at intermediate time-steps is shown in Fig. 2.

1.2. Adams method: Multi-stepping techniques. The development of the Runge–Kutta schemes relies on the definition of the derivative and Taylor expansions. Another approach to solving (1) is to start with the fundamental theorem of calculus [**30**]. Thus the differential equation can be integrated over a time-step Δt to give

$$\frac{d\mathbf{y}}{dt} = f(t,\mathbf{y}) \quad \Rightarrow \quad \mathbf{y}(t+\Delta t) - \mathbf{y}(t) = \int_t^{t+\Delta t} f(t,\mathbf{y})dt. \tag{16}$$

And once again using our iteration notation we find

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} f(t,\mathbf{y})dt. \tag{17}$$

This integral iteration relation is simply a restatement of (7) with $\Delta t \cdot \phi = \int_{t_n}^{t_{n+1}} f(t,\mathbf{y})dt$. However, at this point, no approximations have been made and (17) is exact. The numerical solution will be found by approximating $f(t,\mathbf{y}) \approx p(t,\mathbf{y})$ where $p(t,\mathbf{y})$ is a polynomial. Thus the iteration scheme in this instance will be given by

$$\mathbf{y}_{n+1} \approx \mathbf{y}_n + \int_{t_n}^{t_{n+1}} p(t,\mathbf{y})dt. \tag{18}$$

It only remains to determine the form of the polynomial to be used in the approximation.

The *Adams–Bashforth* suite of computational methods uses the current point and a determined number of past points to evaluate the future solution. As with the Runge–Kutta schemes, the order of accuracy is determined by the choice of ϕ. In the Adams–Bashforth case, this relates directly to the choice of the polynomial approximation $p(t,\mathbf{y})$. A first-order scheme can easily be constructed by allowing

$$p_1(t) = \text{constant} = f(t_n,\mathbf{y}_n), \tag{19}$$

where the present point and no past points are used to determine the value of the polynomial. Inserting this first-order approximation into (18) results in the previously found Euler scheme

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(t_n,\mathbf{y}_n). \tag{20}$$

Alternatively, we could assume that the polynomial used both the current point and the previous point so that a second-order scheme resulted. The linear polynomial which passes through these

two points is given by

$$p_2(t) = f_{n-1} + \frac{f_n - f_{n-1}}{\Delta t}(t - t_{n-1}). \tag{21}$$

When inserted into (18), this linear polynomial yields

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} \left(f_n + \frac{f_n - f_{n-1}}{\Delta t}(t - t_{n-1}) \right) dt. \tag{22}$$

Upon integration and evaluation at the upper and lower limits, we find the following second-order Adams–Bashforth scheme

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2} \left[3f\left(t_n, \mathbf{y}_n\right) - f\left(t_{n-1}, \mathbf{y}_{n-1}\right) \right]. \tag{23}$$

In contrast to the Runge–Kutta method, this is a *two-step algorithm* which requires two initial conditions. This technique can be easily generalized to include more past points and thus higher accuracy. However, as accuracy is increased, so are the number of initial conditions required to step forward one time-step Δt. Aside from the first-order accurate scheme, any implementation of Adams–Bashforth will require a *bootstrap* to generate a second "initial condition" for the solution iteration process.

The Adams–Bashforth scheme uses current and past points to approximate the polynomial $p(t, \mathbf{y})$ in (18). If instead a future point, the present, and the past is used, then the scheme is known as an *Adams–Moulton method*. As before, a first-order scheme can easily be constructed by allowing

$$p_1(t) = \text{constant} = f(t_{n+1}, \mathbf{y}_{n+1}), \tag{24}$$

where the future point and no past and present points are used to determine the value of the polynomial. Inserting this first-order approximation into (18) results in the *backward Euler scheme*

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(t_{n+1}, \mathbf{y}_{n+1}). \tag{25}$$

Alternatively, we could assume that the polynomial used both the future point and the current point so that a second-order scheme resulted. The linear polynomial which passes through these two points is given by

$$p_2(t) = f_n + \frac{f_{n+1} - f_n}{\Delta t}(t - t_n). \tag{26}$$

Inserted into (18), this linear polynomial yields

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} \left(f_n + \frac{f_{n+1} - f_n}{\Delta t}(t - t_n) \right) dt. \tag{27}$$

Upon integration and evaluation at the upper and lower limits, we find the following second-order Adams–Moulton scheme

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2} \left[f(t_{n+1}, \mathbf{y}_{n+1}) + f(t_n, \mathbf{y}_n) \right]. \tag{28}$$

Once again this is a two-step algorithm. However, it is categorically different from the Adams–Bashforth methods since it results in an *implicit scheme*, i.e. the unknown value $\mathbf{y}_{n+1}$ is specified through a nonlinear equation (28). The solution of this nonlinear system can be very difficult, thus making *explicit schemes* such as Runge–Kutta and Adams–Bashforth, which are simple iterations, more easily handled. However, implicit schemes can have advantages when considering stability issues related to time-stepping.

One way to circumvent the difficulties of the implicit stepping method while still making use of its power is to use a *predictor–corrector method*. This scheme draws on the power of both the Adams–Bashforth and Adams–Moulton schemes. In particular, the second-order implicit scheme given by (28) requires the value of $f(t_{n+1}, \mathbf{y}_{n+1})$ in the right-hand side. If we can predict (approximate) this value, then we can use this predicted value to solve (28) explicitly. Thus we begin with

a predictor step to estimate $\mathbf{y}_{n+1}$ so that $f(t_{n+1}, \mathbf{y}_{n+1})$ can be evaluated. We then insert this value into the right-hand side of (28) and explicitly find the corrected value of $\mathbf{y}_{n+1}$. The second-order predictor–corrector steps are then as follows:

$$\text{Predictor (Adams–Bashforth): } \mathbf{y}^P_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2}\left[3f_n - f_{n-1}\right] \tag{29a}$$

$$\text{Corrector (Adams–Moulton): } \mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2}\left[\, f\big(t_{n+1}, \mathbf{y}^P_{n+1}\big) + f(t_n, \mathbf{y}_n)\right]. \tag{29b}$$

Thus the scheme utilizes both explicit and implicit time-stepping schemes without having to solve a system of nonlinear equations.

1.3. Higher order differential equations. Thus far, we have considered systems of first-order equations. Higher order differential equations can be put into this form and the methods outlined here can be applied. For example, consider the third-order, nonhomogeneous, differential equation

$$\frac{d^3u}{dt^3} + u^2\frac{du}{dt} + \cos t \cdot u = g(t). \tag{30}$$

By defining

$$y_1 = u \tag{31a}$$

$$y_2 = \frac{du}{dt} \tag{31b}$$

$$y_3 = \frac{d^2u}{dt^2}, \tag{31c}$$

we find that $dy_3/dt = d^3u/dt^3$. Using the original equation along with the definitions of y_i we find that

$$\frac{dy_1}{dt} = y_2 \tag{32a}$$

$$\frac{dy_2}{dt} = y_3 \tag{32b}$$

$$\frac{dy_3}{dt} = \frac{d^3u}{dt^3} = -u^2\frac{du}{dt} - \cos t \cdot u + g(t) = -y_1^2 y_2 - \cos t \cdot y_1 + g(t) \tag{32c}$$

which results in the original differential equation (1) considered previously

$$\frac{d\mathbf{y}}{dt} = \frac{d}{dt}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_2 \\ y_3 \\ -y_1^2 y_2 - \cos t \cdot y_1 + g(t) \end{pmatrix} = f(t, \mathbf{y}). \tag{33}$$

At this point, all the time-stepping techniques developed thus far can be applied to the problem. It is imperative to write any differential equation as a first-order system before solving it numerically with the time-stepping schemes developed here.

1.4. python commands. The time-stepping schemes considered here are all available in the python suite of differential equation solvers. The following are a few of the most common solvers:

- **ode23:** second-order Runge–Kutta routine;
- **ode45:** fourth-order Runge–Kutta routine;
- **ode113:** variable-order predictor–corrector routine;
- **ode15s:** variable-order Gear method for stiff problems [**31, 32**].

scheme	local error ϵ_k	global error E_k
Euler	$O(\Delta t^2)$	$O(\Delta t)$
2nd-order Runge-Kutta	$O(\Delta t^3)$	$O(\Delta t^2)$
4th-order Runge-Kutta	$O(\Delta t^5)$	$O(\Delta t^4)$
2nd-order Adams-Bashforth	$O(\Delta t^3)$	$O(\Delta t^2)$

TABLE 1. Local and global discretization errors associated with various time-stepping schemes.

2. Error Analysis for Time-Stepping Routines

Accuracy and *stability* are fundamental to numerical analysis and are the key factors in evaluating any numerical integration technique. Therefore, it is essential to evaluate the accuracy and stability of the time-stepping schemes developed. Rarely does it occur that both accuracy and stability work in concert. In fact, they are often offsetting and work directly against each other. Thus a highly accurate scheme may compromise stability, whereas a low-accuracy scheme may have excellent stability properties.

We begin by exploring accuracy. In the context of time-stepping schemes, the natural place to begin is with Taylor expansions. Thus we consider the expansion

$$\mathbf{y}(t+\Delta t) = \mathbf{y}(t) + \Delta t \cdot \frac{d\mathbf{y}(t)}{dt} + \frac{\Delta t^2}{2} \cdot \frac{d^2\mathbf{y}(c)}{dt^2} \tag{1}$$

where $c \in [t, t+\Delta t]$. Since we are considering $d\mathbf{y}/dt = f(t,\mathbf{y})$, the above formula reduces to the Euler iteration scheme

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f\left(t_n, \mathbf{y}_n\right) + O\left(\Delta t^2\right). \tag{2}$$

It is clear from this that the truncation error is $O(\Delta t^2)$. Specifically, the truncation error is given by $\Delta t^2/2 \cdot d^2\mathbf{y}(c)/dt^2$.

Of importance is how this truncation error contributes to the overall error in the numerical solution. Two types of error are important to identify: *local* and *global error*. Each is significant in its own right. However, in practice we are only concerned with the global (cumulative) error. The *global discretization error* is given by

$$E_k = \mathbf{y}(t_k) - \mathbf{y}_k \tag{3}$$

where $\mathbf{y}(t_k)$ is the exact solution and $\mathbf{y}_k$ is the numerical solution. The *local discretization error* is given by

$$\epsilon_{k+1} = \mathbf{y}(t_{k+1}) - (\mathbf{y}(t_k) + \Delta t \cdot \phi) \tag{4}$$

where $\mathbf{y}(t_{k+1})$ is the exact solution and $\mathbf{y}(t_k) + \Delta t \cdot \phi$ is a one-step approximation over the time interval $t \in [t_n, t_{n+1}]$.

For the Euler method, we can calculate both the local and global error. Given a time-step Δt and a specified time interval $t \in [a,b]$, we have after K steps that $\Delta t \cdot K = b - a$. Thus we find

$$\text{local:} \quad \epsilon_k = \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(c_k)}{dt^2} \sim O(\Delta t^2) \tag{5a}$$

$$\begin{aligned} \text{global:} \quad E_k &= \sum_{j=1}^{K} \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(c_j)}{dt^2} \approx \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(\mathbf{c})}{dt^2} \cdot K \\ &= \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(\mathbf{c})}{dt^2} \cdot \frac{b-a}{\Delta t} = \frac{b-a}{2}\Delta t \cdot \frac{d^2\mathbf{y}(\mathbf{c})}{dt^2} \sim O(\Delta t) \end{aligned} \tag{5b}$$

which gives a local error for the Euler scheme which is $O(\Delta t^2)$ and a global error which is $O(\Delta t)$. Thus the cumulative error is large for the Euler scheme, i.e. it is not very accurate.

A similar procedure can be carried out for all the schemes discussed thus far, including the multi-step Adams schemes. Table 1 illustrates various schemes and their associated local and global errors. The error analysis suggests that the error will always decrease in some power of Δt. Thus it is tempting to conclude that higher accuracy is easily achieved by taking smaller time-steps Δt. This would be true if not for round-off error in the computer.

2.1. Round-off and step-size. An unavoidable consequence of working with numerical computations is round-off error. When working with most computations, *double precision* numbers are used. This allows for 16-digit accuracy in the representation of a given number. This round-off has a significant impact upon numerical computations and the issue of time-stepping.

As an example of the impact of round-off, we consider the Euler approximation to the derivative

$$\frac{d\mathbf{y}}{dt} \approx \frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{\Delta t} + \epsilon(\mathbf{y}_n, \Delta t) \tag{6}$$

where $\epsilon(\mathbf{y}_n, \Delta t)$ measures the truncation error. Upon evaluating this expression in the computer, round-off error occurs so that

$$\mathbf{y}_{n+1} = \mathbf{Y}_{n+1} + \mathbf{e}_{n+1}. \tag{7}$$

Thus the combined error between the round-off and truncation gives the following expression for the derivative:

$$\frac{d\mathbf{y}}{dt} = \frac{\mathbf{Y}_{n+1} - \mathbf{Y}_n}{\Delta t} + E_n(\mathbf{y}_n, \Delta t) \tag{8}$$

where the total error, E_n, is the combination of round-off and truncation such that

$$E_n = E_{\text{round}} + E_{\text{trunc}} = \frac{\mathbf{e}_{n+1} - \mathbf{e}_n}{\Delta t} - \frac{\Delta t}{2}\frac{d^2\mathbf{y}(c)}{dt^2}. \tag{9}$$

We now determine the maximum size of the error. In particular, we can bound the maximum value of round-off and the second derivative to be

$$|\mathbf{e}_{n+1}| \le e_r \tag{10a}$$

$$|-\mathbf{e}_n| \le e_r \tag{10b}$$

$$M = \max_{c\in[t_n, t_{n+1}]} \left\{ \left| \frac{d^2\mathbf{y}(c)}{dt^2} \right| \right\}. \tag{10c}$$

This then gives the maximum error to be

$$|E_n| \le \frac{e_r + e_r}{\Delta t} + \frac{\Delta t}{2}M = \frac{2e_r}{\Delta t} + \frac{\Delta t M}{2}. \tag{11}$$

To minimize the error, we require that $\partial|E_n|/\partial(\Delta t) = 0$. Calculating this derivative gives

$$\frac{\partial|E_n|}{\partial(\Delta t)} = -\frac{2e_r}{\Delta t^2} + \frac{M}{2} = 0, \tag{12}$$

so that

$$\Delta t = \left(\frac{4e_r}{M}\right)^{1/2}. \tag{13}$$

This gives the step-size resulting in a minimum error. Thus the smallest step-size is not necessarily the most accurate. Rather, a balance between round-off error and truncation error is achieved to obtain the optimal step-size.

2.2. Stability. The accuracy of any scheme is certainly important. However, it is meaningless if the scheme is not stable numerically. The essense of a stable scheme: the numerical solutions do not blow up to infinity. As an example, consider the simple differential equation

$$\frac{dy}{dt} = \lambda y \tag{14}$$

with

$$y(0) = y_0. \tag{15}$$

The analytic solution is easily calculated to be $y(t) = y_0 \exp(\lambda t)$. However, if we solve this problem numerically with a forward Euler method we find

$$y_{n+1} = y_n + \Delta t \cdot \lambda y_n = (1 + \lambda \Delta t) y_n. \tag{16}$$

After N steps, we find this iteration scheme yields

$$y_N = (1 + \lambda \Delta t)^N y_0. \tag{17}$$

Given that we have a certain amount of round-off error, the numerical solution would then be given by

$$y_N = (1 + \lambda \Delta t)^N (y_0 + e). \tag{18}$$

The error then associated with this scheme is given by

$$E = (1 + \lambda \Delta t)^N e. \tag{19}$$

At this point, the following observations can be made. For $\lambda > 0$, the solution $y_N \to \infty$ in Eq. (18) as $N \to \infty$. So although the error also grows, it may not be significant in comparison to the size of the numerical solution.

In contrast, Eq. (18) for $\lambda < 0$ is markedly different. For this case, $y_N \to 0$ in Eq. (18) as $N \to \infty$. The error, however, can dominate in this case. In particular, we have the following two cases for the error given by (19):

$$\text{I:} \quad |1 + \lambda \Delta t| < 1 \ \text{ then } \ E \to 0 \tag{20a}$$

$$\text{II:} \quad |1 + \lambda \Delta t| > 1 \ \text{ then } \ E \to \infty. \tag{20b}$$

In case I, the scheme would be considered stable. However, case II holds and is unstable provided $\Delta t > -2/\lambda$.

A general theory of stability can be developed for any one-step time-stepping scheme. Consider the one-step recursion relation for an $M \times M$ system

$$\mathbf{y}_{n+1} = \mathbf{A}\mathbf{y}_n. \tag{21}$$

After N steps, the algorithm yields the solution

$$\mathbf{y}_N = \mathbf{A}^N \mathbf{y}_0, \tag{22}$$

where $\mathbf{y}_0$ is the initial vector. A well-known result from linear algebra is that

$$\mathbf{A}^N = \mathbf{S}\mathbf{\Lambda}^N \mathbf{S}^{-1} \tag{23}$$

where $\mathbf{S}$ is the matrix whose columns are the eigenvectors of $\mathbf{A}$, and

$$\mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \lambda_M \end{pmatrix} \ \rightarrow \ \mathbf{\Lambda}^N = \begin{pmatrix} \lambda_1^N & 0 & \cdots & 0 \\ 0 & \lambda_2^N & 0 & \cdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \lambda_M^N \end{pmatrix} \tag{24}$$

is a diagonal matrix whose entries are the eigenvalues of $\mathbf{A}$. Thus upon calculating $\mathbf{\Lambda}^N$, we are only concerned with the eigenvalues. In particular, instability occurs if $|\lambda_i| > 1$ for $i = 1, 2, \ldots, M$. This

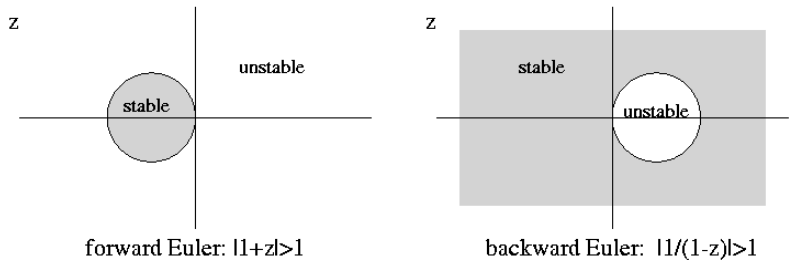

FIGURE 3. Regions for stable stepping (shaded) for the forward Euler and backward Euler schemes. The criteria for instability is also given for each stepping method.

method can be easily generalized to two-step schemes (Adams methods) by considering $\mathbf{y}_{n+1} = \mathbf{A}\mathbf{y}_n + \mathbf{B}\mathbf{y}_{n-1}$.

Lending further significance to this stability analysis is its connection with practical implementation. We contrast the difference in stability between the forward and backward Euler schemes. The forward Euler scheme has already been considered in (16)–(19). The backward Euler displays significant differences in stability. If we again consider (14) with (15), the backward Euler method gives the iteration scheme

$$y_{n+1} = y_n + \Delta t \cdot \lambda y_{n+1}, \tag{25}$$

which after N steps leads to

$$y_N = \left(\frac{1}{1-\lambda\Delta t}\right)^N y_0. \tag{26}$$

The round-off error associated with this scheme is given by

$$E = \left(\frac{1}{1-\lambda\Delta t}\right)^N e. \tag{27}$$

By letting $z = \lambda\Delta t$ be a complex number, we find the following criteria to yield unstable behavior based upon (19) and (27):

$$\text{Forward Euler:} \qquad |1+z| > 1 \tag{28a}$$

$$\text{Backward Euler:} \qquad \left|\frac{1}{1-z}\right| > 1. \tag{28b}$$

Figure 3 shows the regions of stable and unstable behavior as a function of z. It is observed that the forward Euler scheme has a very small range of stability whereas the backward Euler scheme has a large range of stability. This large stability region is part of what makes implicit methods so attractive. Thus stability regions can be calculated. However, control of the accuracy is also essential.

3. Advanced Time-Stepping Algorithms

Before closing our analysis on time-stepping algorithms, a few alternative methods are considered in the interest of achieving better performance. Specifically, the commonly used adaptive time-stepping algorithm will be outlined. In addition, the exponential time-stepper will be developed for numerically stiff problems.

3.1. Adaptive time-stepping algorithm. Adaptive time-stepping algorithms are tremendously important in practice. The premise of the adaptive stepping technique is to take as large a time-step as possible while retaining a prescribed accuracy. All of the time-stepping algorithms used in python have a built-in adaptive stepper. Indeed, the fundamental premise of the standard

differential equation solver in python is to guarantee a solution with a prescribed absolute and relative tolerance. The time-step Δt is chosen so that the tolerance constraints are met.

The following algorithm outlines the method employed in the adaptive stepping method routine.

(1) Start with a default time-step Δt_0.
(2) Use one of the iteration algorithms (such as fourth-order Runge–Kutta) to take a time-step Δt into the future. Represent the solution by $f_1(t + \Delta t)$. The initial $\Delta t = \Delta t_0$.
(3) Now cut the time-step in half ($\Delta t/2$) and use the iteration algorithm to advance Δt into the future. This would require two iterative steps. Represent the solution by $f_2(t + \Delta t)$.
(4) Compare the solutions using Δt and $\Delta t/2$. For instance, one may measure the L^2 norm between the two solutions: $E = \|f_1 - f_2\|$.
(5) If the difference is above a prescribed tolerance, i.e. $E >$ tolerance, then cut the time-step in half again to $\Delta t/4$ and compare again. Continue cutting the time-step in half until the tolerance is achieved.
(6) If the comparison is already below the prescribed tolerance, then the time-step can be doubled in size to $2\Delta t$. The comparison can be made again until the tolerance condition is violated.

This algorithm is a shell of what the algorithm might look like. Certainly a more sophisticated version can be constructed, but this illustrates the key concept of either making the time-step bigger or smaller as needed.

The advantages of such a scheme are enormous. It allows the iterative method for advancing the differential equation solution into the future to be maximally efficient. When the solution is changing slowly, the time-steps will be quite large, whereas when the solution changes rapidly, the time-step will automatically adjust and shorten in order to preserve the accuracy.

In python, the time-step can be adjusted by modifying the tolerance settings associated with the time-stepping algorithm. The following code adjusts the time-step so that a 10^{-4} accuracy is achieved both for relative tolerance and absolute tolerance.

```
TOL=1e-4; OPTIONS = odeset('RelTol',TOL,'AbsTol',TOL);
[t,y] = ode45('F',tspan,y0,OPTIONS);
```

The speed of the code is largely determined by the accuracy setting as determined from the **odeset** command. The default is a 10^{-6} tolerance for both relative and absolute error.

3.2. Exponential time-steppers. A nice example of a time-stepping technique that removes numerical stiffness generated from either large linear terms (or linear, high-order derivative terms) is the exponential time-stepping technique [**33, 34**]. This is the only stiff time-stepper that will be considered in detail in this book. As with any other stiff-stepping technique, advantage is taken of certain properties of the differential equation considered in order to make the algorithm faster, i.e. in order to maximize the step-size while keeping a fixed accuracy.

The prototype equation to be considered is the differential equation of the form [**33**]

$$\frac{d\mathbf{y}}{dt} = c\mathbf{y} + F(\mathbf{y}, t) \tag{1}$$

where $|c| \gg 1$. When c is large in magnitude, it dominates the selection of the time-step Δt. In fact, it forces the time-step Δt to be quite small in order to accurately resolve the future solution $\mathbf{y}(t+\Delta t)$. It should be noted that the large c term often arises from high-order and linear derivatives in problems involving partial differential equations. This will be considered in future sections.

One method of dealing with numerical stiffness induced by c is to attempt a solution via the integrating factor method. Multiplying Eq. (1) by the factor $\exp(-ct)$ gives the following set of

algebraic reductions:

$$\begin{aligned}\frac{d\mathbf{y}}{dt}\exp(-ct) &= c\mathbf{y}\exp(-ct) + F(\mathbf{y},t)\exp(-ct)\\ \frac{d\mathbf{y}}{dt}\exp(-ct) - c\mathbf{y}\exp(-ct) &= F(\mathbf{y},t)\exp(-ct)\\ \frac{d}{dt}\left(\mathbf{y}\exp(-ct)\right) &= F(\mathbf{y},t)\exp(-ct). \end{aligned} \tag{2}$$

Integrating both sides from time t to time $t+\Delta t$ yields

$$\mathbf{y}(t+\Delta t)\exp\left(-c(t+\Delta t)\right) - \mathbf{y}(t)\exp(-ct) = \int_t^{t+\Delta t} F(\mathbf{y}(\tau),\tau)\exp(-c\tau)d\tau. \tag{3}$$

Finally, by multiplying both sides by $\exp(c(t+\Delta t))$ and making a change of variables in the integral, the following formula is achieved:

$$\mathbf{y}(t+\Delta t) = \mathbf{y}(t)\exp(c\Delta t) + \exp(c\Delta t)\int_0^{\Delta t} F(\mathbf{y}(t+\tau),t+\tau)\exp(-c\tau)d\tau. \tag{4}$$

This formula is exact. Moreover, it has some of the basic characteristics of the Adams–Bashforth and Adams–Moulton schemes considered earlier where the integral approximation determines the accuracy and iteration of the scheme. But unlike the Adams methods, the linear term that is scaled with the parameter c is explicitly accounted for by the integrating factor.

The approximation of the integral yields the exponential steppers of interest. The simplest approximation is to assume that the function F takes on a constant value so that $F(\mathbf{y}(t+\tau),t+\tau) \approx F(\mathbf{y}(t),t)$. Integrating the integral now with respect to τ yields

$$\mathbf{y}_{n+1} = \mathbf{y}_n\exp(c\Delta t) + F_n\left(\exp(c\Delta t) - 1\right)/c \tag{5}$$

where $\mathbf{y}_n = \mathbf{y}(t_n)$ and $F_n = F(\mathbf{y}_n, t_n)$. Note that in the limit as $c \ll 1$, this formula asymptotically approaches the Euler stepping formula.

As with the Adams–Bashforth method, a higher order approximation can be used for evaluating the integral. In particular, the following can be used

$$F = F_n + \tau\left(F_n - F_{n-1}\right)/\Delta t + O\left(\Delta t^2\right). \tag{6}$$

This approximation to the integrand gives an improved accuracy to the time-stepping method. When inserted into the formula (4), the following time-stepping algorithm is derived

$$\begin{aligned}\mathbf{y}_{n+1} = \quad & \mathbf{y}_n\exp(c\Delta t) + F_n\left[(1+c\Delta t)\exp(c\Delta t) - 1 - 2c\Delta t\right]/\left(c^2\Delta t\right)\\ & + F_{n-1}\left[1 + c\Delta t - \exp(c\Delta t)\right]/\left(c^2\Delta t\right). \end{aligned} \tag{7}$$

In the limit as $c \to 0$, this reduces to the second-order Adams–Bashforth scheme. But recall that the purpose of this scheme is to consider problems for which $c \gg 1$.

Cox and Matthews [**33**] continue with this idea in order to derive the equivalent of the fourth-order Runge–Kutta scheme with the exponential time-stepping explicitly accounted for. The claim is that this derivation is nontrivial and requires careful manipulation and the aid of symbolic computing. Regardless, the following exponential, fourth-order Runge–Kutta scheme is developed

$$\begin{aligned}\mathbf{y}_{n+1} \quad = \quad & e^{c\Delta t}\mathbf{y}_n + \left[-4 - c\Delta t + e^{c\Delta t}\left(4 - 3c\Delta t + (c\Delta t)^2\right)\right]F(\mathbf{y}_n,t_n)/\left(c^3\Delta t^2\right)\\ & + 2\left[2 + c\Delta t + e^{c\Delta t}(-2 + c\Delta t)\right]F(\mathbf{a}_n,t_n+\Delta t/2) + F(\mathbf{b}_n,t_n+\Delta t/2)\\ & + \left[-4 - 3c\Delta t - (c\Delta t)^2 + e^{c\Delta t}\left(4 - c\Delta t\right)\right]F(\mathbf{c}_n,t_n+\Delta t) \end{aligned} \tag{8}$$

where

$$\mathbf{a}_n = \mathbf{y}_n e^{c\Delta t/2} + \left(e^{c\Delta t/2} - 1\right) F(\mathbf{y}_n, t_n)/c \tag{9a}$$

$$\mathbf{b}_n = \mathbf{y}_n e^{c\Delta t/2} + \left(e^{c\Delta t/2} - 1\right) F(\mathbf{a}_n, t_n + \Delta t/2)/c \tag{9b}$$

$$\mathbf{c}_n = \mathbf{a}_n e^{c\Delta t/2} + \left(e^{c\Delta t/2} - 1\right) \left[2F(\mathbf{b}_n, t_n + \Delta t/2) - F(\mathbf{y}_n, t_n)\right]/c. \tag{9c}$$

To implement this in practice, Kassam and Trefethen [**34**] show that special care must be taken in order to evaluate the coefficients $\mathbf{a}_n, \mathbf{b}_n$ and $\mathbf{c}_n$. This evaluation is intimately related to the well-known numerical difficulty in evaluating the function $(e^z - 1)/z$. However, using the method of Kassam and Trefethen [**34**], the exponential time-stepping algorithm becomes a tremendously efficient tool when $c \gg 1$. Indeed, Kassam and Trefethen [**34**] illustrate that an entire order or magnitude in step-size can be gained for higher order partial differential equations such as the Kuramoto–Sivashinky equation. In this case, the high c value is effectively created by linear, four-order diffusion. The order of magnitude in increased step-size Δt makes the implementation of the scheme above a must.

As a final comment, one can easily proceed with other schemes of this type. All that is needed is the starting point of the definition of the derivative or the fundamental theorem of calculus. Approximations are generated from there. The exponential scheme outlined above clearly takes advantage of the linear term by folding it into the integrating factor. All specialty stepping schemes typically do something of this form, or utilize a semi-implicit method, to maximize time-stepping performance and error reduction.

4. Boundary Value Problems: The Shooting Method

To this point, we have only considered the solutions of differential equations for which the initial conditions are known. However, many physical applications do not have specified initial conditions, but rather some given boundary (constraint) conditions. A simple example of such a problem is the second-order boundary value problem

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \tag{1}$$

on $t \in [a, b]$ with the general boundary conditions

$$\alpha_1 y(a) + \beta_1 \frac{dy(a)}{dt} = \gamma_1 \tag{2a}$$

$$\alpha_2 y(b) + \beta_2 \frac{dy(b)}{dt} = \gamma_2. \tag{2b}$$

Thus the solution is defined over a specific interval and must satisfy the relations (2) at the end points of the interval. Figure 4 gives a graphical representation of a generic boundary value problem solution. We discuss the algorithm necessary to make use of the time-stepping schemes in order to solve such a problem.

4.1. The shooting method. The boundary value problems constructed here require information at the present time ($t = a$) and a future time ($t = b$). However, the time-stepping schemes developed previously only require information about the starting time $t = a$. Some effort is then needed to reconcile the time-stepping schemes with the boundary value problems presented here.

We begin by reconsidering the generic boundary value problem

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \tag{3}$$

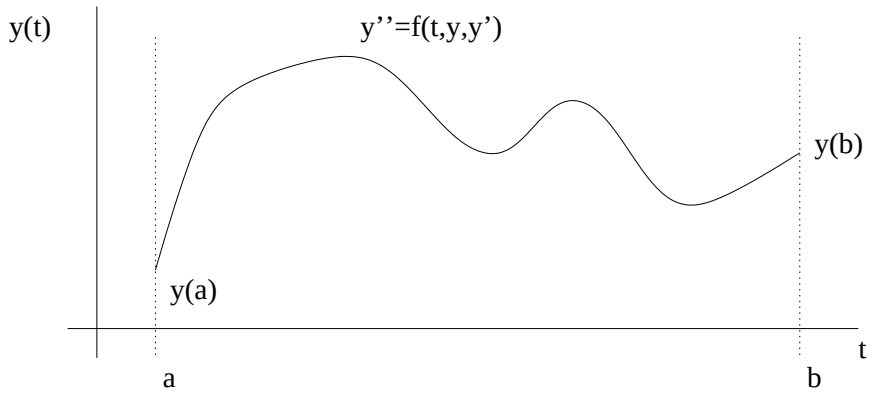

FIGURE 4. Graphical depiction of the structure of a typical solution to a boundary value problem with constraints at $t = a$ and $t = b$.

on $t \in [a, b]$ with the boundary conditions

$$y(a) = \alpha \tag{4a}$$

$$y(b) = \beta. \tag{4b}$$

The stepping schemes considered thus far for second-order differential equations involve a choice of the initial conditions $y(a)$ and $y'(a)$. We can still approach the boundary value problem from this framework by choosing the "initial" conditions

$$y(a) = \alpha \tag{5a}$$

$$\frac{dy(a)}{dt} = A, \tag{5b}$$

where the constant A is chosen so that as we advance the solution to $t = b$ we find $y(b) = \beta$. The shooting method gives an iterative procedure with which we can determine this constant A. Figure 5 illustrates the solution of the boundary value problem given two distinct values of A. In this case, the value of $A = A_1$ gives a value for the initial slope which is too low to satisfy the boundary conditions (4), whereas the value of $A = A_2$ is too large to satisfy (4).

4.2. Computational algorithm. The above example demonstrates that adjusting the value of A in (5b) can lead to a solution which satisfies (4b). We can solve this using a self-consistent algorithm to search for the appropriate value of A which satisfies the original problem. The basic algorithm is as follows:

(1) Solve the differential equation using a time-stepping scheme with the initial conditions $y(a) = \alpha$ and $y'(a) = A$.
(2) Evaluate the solution $y(b)$ at $t = b$ and compare this value with the target value of $y(b) = \beta$.
(3) Adjust the value of A (either bigger or smaller) until a desired level of tolerance and accuracy is achieved. A bisection method for determining values of A, for instance, may be appropriate.
(4) Once the specified accuracy has been achieved, the numerical solution is complete and is accurate to the level of the tolerance chosen and the discretization scheme used in the time-stepping.

We illustrate graphically a bisection process in Fig. 6 and show the convergence of the method to the numerical solution which satisfies the original boundary conditions $y(a) = \alpha$ and $y(b) = \beta$. This process can occur quickly so that convergence is achieved in a relatively low number of iterations provided the differential equation is well behaved.

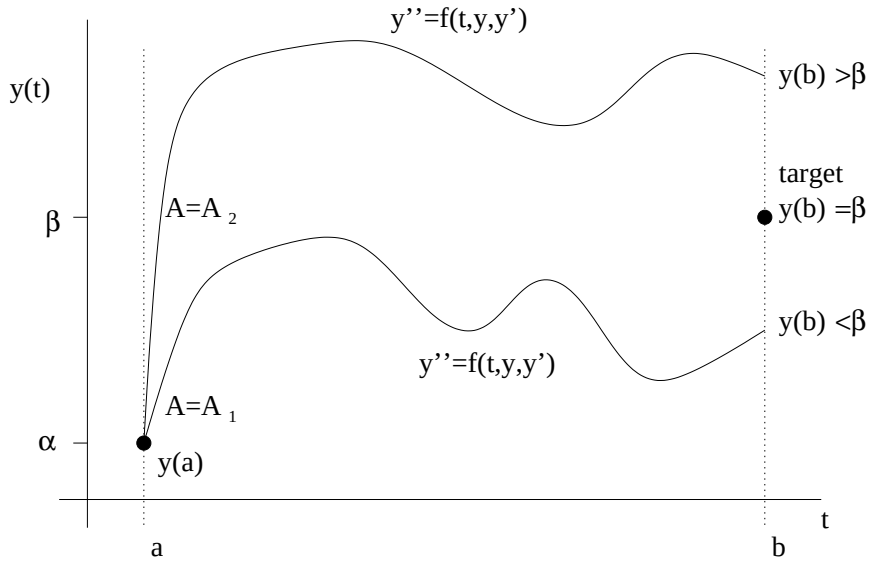

FIGURE 5. Solutions to the boundary value problem with $y(a) = \alpha$ and $y'(a) = A$. Here, two values of A are used to illustrate the solution behavior and its lack of matching the correct boundary value $y(b) = \beta$. However, the two solutions suggest that a bisection scheme could be used to find the correct solution and value of A.

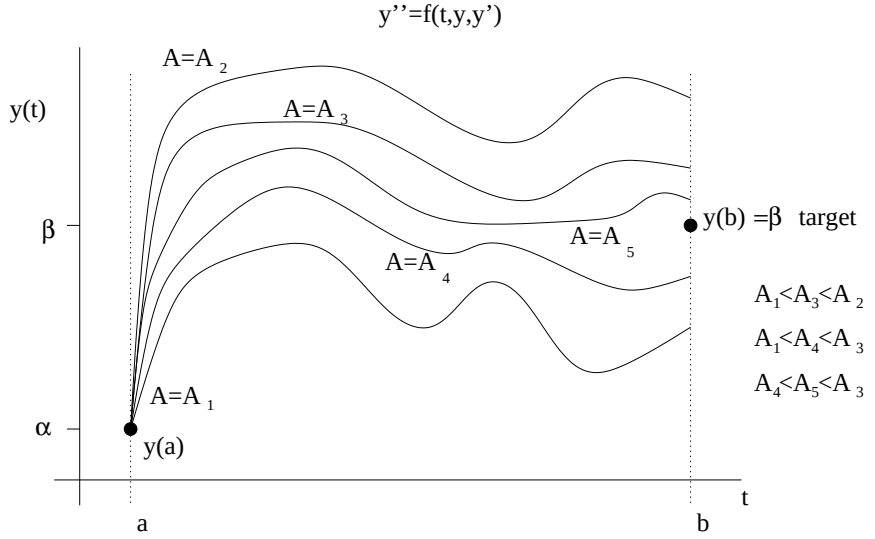

FIGURE 6. Graphical illustration of the shooting process which uses a bisection scheme to converge to the appropriate value of A for which $y(b) = \beta$.

4.3. Shooting example. To illustrate the implementation of the shooting method, consider the following boundary value problem

$$y'' + (x^2 - \sin x)y' - (\cos^2 x)y = 5 \qquad x \in [0, 1] \tag{6}$$

with the boundary conditions

$$y(0) = 3 \tag{7a}$$

$$y'(1) = 5. \tag{7b}$$

The first step is to write the above boundary value problem as an equivalent system of equations by defining $y_1 = y$ and $y_2 = y'$. This yields

$$\begin{aligned} y_1' &= y_2 && \text{(8a)} \\ y_2' &= -(x^2 - \sin x)y_2 + (\cos^2 x)y_1 + 5 && \text{(8b)} \\ y_1(0) &= 3 \;\; y_2(1) = 5\,. && \text{(8c)} \end{aligned}$$

The idea is to replace the second boundary condition above, $y_2(1) = 5$, with $y_1'(0) = A$ where A is to be determined in the shooting algorithm. From exploring the differential equations with different value of A, it is found that $A > -3$ in order for the derivate at $x = 1$ to go from below the value of 5 to above the value of 5. The following code finds the solution in a fairly straightforward manner.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def bvpexam_rhs(y, x):
    return [y[1], -(x**2 - np.sin(x)) * y[1] + np.cos(x)**2 * y[0] + 5]

xspan = [0, 1]  # x range
A = -3  # initial derivative value
dA = 0.5  # step size for derivative adjustment

for j in range(100):
    y0 = [3, A]  # initial condition
    x = np.linspace(xspan[0], xspan[1], 100)  # grid for odeint
    ysol = odeint(bvpexam_rhs, y0, x)  # solve ODE

    if abs(ysol[-1, 1] - 5) < 10**(-6):  # check convergence
        break

    if ysol[-1, 1] < 5:  # adjust launch angle
        A += dA  # if below five, make A bigger
    else:
        A -= dA  # if above five, make A smaller
        dA /= 2  # refine search now
```

Figure 7 depicts the final solution $y(x) = y_1(x)$ along with the derivative $y'(x) = y_2(x)$. The dotted line is the initial guess ($A = -3$) for a solution and the starting point of the shooting algorithm. Note that the algorithm uses something like a bisection algorithm for refining the search for the appropriate value of A.

4.4. Eigenvalues and eigenfunctions: The infinite domain. Boundary value problems often arise as eigenvalue systems for which the eigenvalue and eigenfunction must both be determined. As an example of such a problem, we consider the second-order differential equation on the infinite line

$$\frac{d^2\psi_n}{dx^2} + [n(x) - \beta_n]\,\psi_n = 0 \qquad (9)$$

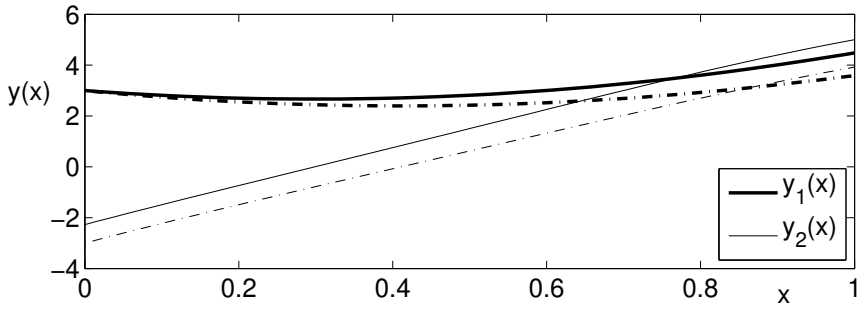

FIGURE 7. Solution of the boundary value problem depicting the solution $y(x) = y_1(x)$ (bolded lines) and its derivative $y'(x) = y_2(x)$. The dotted lines are the initial guess of the shooting algorithm for which $y'(1) = y_2(1) = A = -3$.

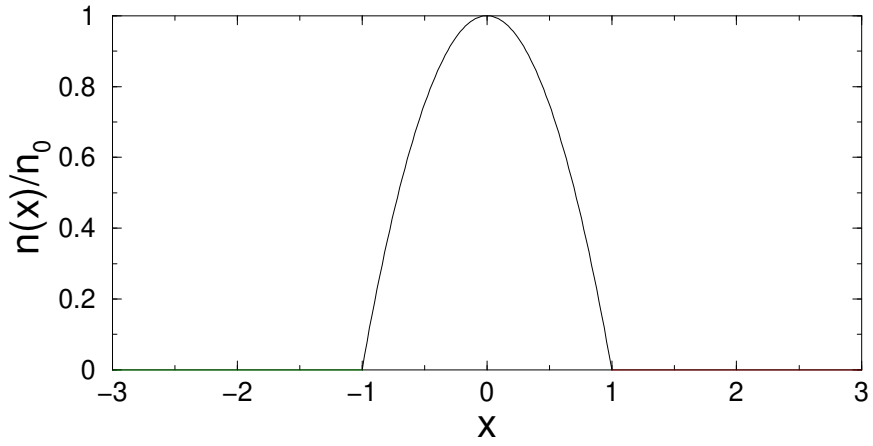

FIGURE 8. Plot of the spatial function $n(x)$.

with the boundary conditions $\psi_n(x) \to 0$ as $x \to \pm\infty$. For this example, we consider the spatial function $n(x)$ which is given by

$$n(x) = n_0 \begin{cases} 1 - |x|^2 & 0 \le |x| \le 1 \\ 0 & |x| > 1 \end{cases} \tag{10}$$

with n_0 being an arbitrary constant. Figure 8 shows the spatial dependence of $n(x)$. The parameter β_n in this problem is the eigenvalue. For each eigenvalue, we can calculate a normalized eigenfunction ψ_n. The standard normalization requires $\int_{-\infty}^{\infty} |\psi_n|^2 dx = 1$.

Although the boundary conditions are imposed as $x \to \pm\infty$, computationally we require a finite domain. We thus define our computational domain to be $x \in [-L, L]$ where $L \gg 1$. Since $n(x) = 0$ for $|x| > 1$, the governing equation reduces to

$$\frac{d^2\psi_n}{dx^2} - \beta_n \psi_n = 0 \quad |x| > 1 \tag{11}$$

which has the general solution

$$\psi_n = c_1 \exp\left(\sqrt{\beta_n} x\right) + c_2 \exp\left(-\sqrt{\beta_n} x\right) \tag{12}$$

for $\beta_n \ge 0$. Note that we can only consider values of $\beta_n \ge 0$ since for $\beta_n < 0$, the general solution becomes $\psi_n = c_1 \cos(\sqrt{|\beta_n|}x) + c_2 \sin(-\sqrt{|\beta_n|}x)$ which does not decay to zero as $x \to \pm\infty$. In

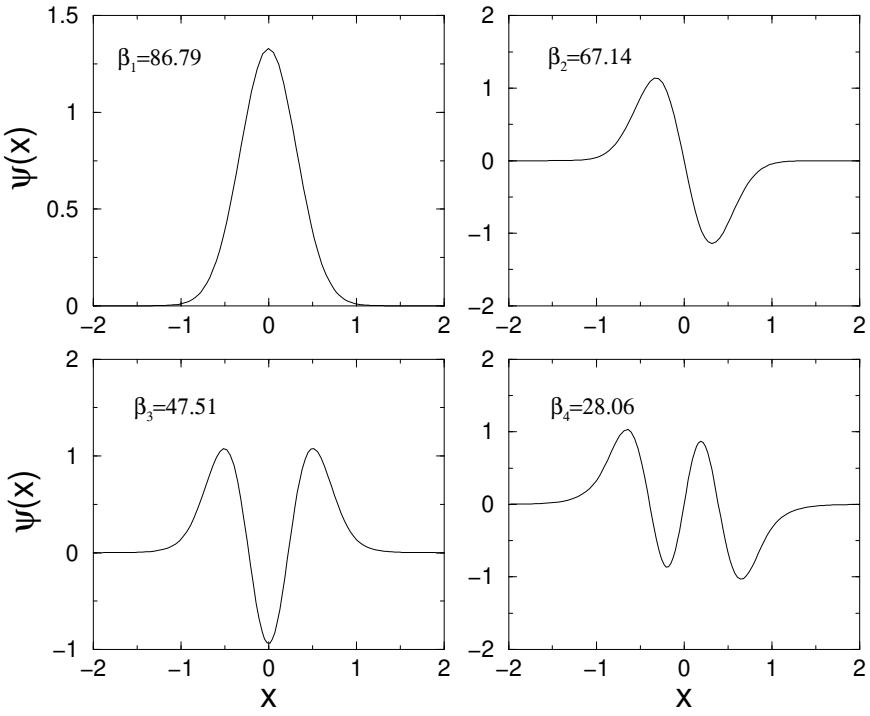

FIGURE 9. Plot of the first four eigenfunctions along with their eigenvalues β_n. For this example, $L = 2$ and $n_0 = 100$. These eigenmode structures are typical of those found in quantum mechanics and electromagnetic waveguides.

order to ensure that the decay boundary conditions are satisfied, we must eliminate one of the two linearly independent solutions of the general solution. In particular, we must have

$$x \to \infty : \qquad \psi_n = c_2 \exp\left(-\sqrt{\beta_n} x\right) \tag{13a}$$

$$x \to -\infty : \qquad \psi_n = c_1 \exp\left(\sqrt{\beta_n} x\right). \tag{13b}$$

Thus the requirement that the solution decays at infinity eliminates one of the two linearly independent solutions. Alternatively, we could think of this situation as being a case where only one linearly independent solution is allowed as $x \to \pm\infty$. But a single linearly independent solution corresponds to a first-order differential equation. Therefore, the decay solutions (13) can equivalently be thought of as solutions to the following first-order equations:

$$x \to \infty : \qquad \frac{d\psi_n}{dx} + \sqrt{\beta_n}\psi_n = 0 \tag{14a}$$

$$x \to -\infty : \qquad \frac{d\psi_n}{dx} - \sqrt{\beta_n}\psi_n = 0. \tag{14b}$$

From a computational viewpoint then, the effective boundary conditions to be considered on the computational domain $x \in [-L, L]$ are the following

$$x = L : \qquad \frac{d\psi_n(L)}{dx} = -\sqrt{\beta_n}\psi_n(L) \tag{15a}$$

$$x = -L : \qquad \frac{d\psi_n(-L)}{dx} = \sqrt{\beta_n}\psi_n(-L). \tag{15b}$$

In order to solve the problem, we write the governing differential equation as a system of equations. Thus we let $x_1 = \psi_n$ and $x_2 = d\psi_n/dx$ which gives

$$x_1' = \psi_n' = x_2 \tag{16a}$$
$$x_2' = \psi_n'' = [\beta_n - n(x)]\,\psi_n = [\beta_n - n(x)]\,x_1. \tag{16b}$$

In matrix form, we can write the governing system as

$$\mathbf{x}' = \begin{pmatrix} 0 & 1 \\ \beta_n - n(x) & 0 \end{pmatrix} \mathbf{x} \tag{17}$$

where $\mathbf{x} = (x_1 \;\; x_2)^T = (\psi_n \;\; d\psi_n/dx)^T$. The boundary conditions (15) are

$$x = L: \qquad x_2 = -\sqrt{\beta_n} x_1 \tag{18a}$$
$$x = -L: \qquad x_2 = \sqrt{\beta_n} x_1. \tag{18b}$$

The formulation of the boundary value problem is thus complete. It remains to develop an algorithm to find the eigenvalues β_n and corresponding eigenfunctions ψ_n. Figure 9 illustrates the first four eigenfunctions and their associated eigenvalues for $n_0 = 100$ and $L = 2$.

5. Implementation of Shooting and Convergence Studies

The implementation of the shooting scheme relies on the effective use of a time-stepping algorithm along with a root finding method for choosing the appropriate initial conditions which solve the boundary value problem. The specific system to be considered is similar to that developed in the last section. We consider

$$\mathbf{x}' = \begin{pmatrix} 0 & 1 \\ \beta_n - n(x) & 0 \end{pmatrix} \mathbf{x} \tag{1}$$

where $\mathbf{x} = (x_1 \;\; x_2)^T = (\psi_n \;\; d\psi_n/dx)^T$. The boundary conditions are simplified in this case to be

$$x = 1: \qquad \psi_n(1) = x_1(1) = 0 \tag{2a}$$
$$x = -1: \qquad \psi_n(-1) = x_1(-1) = 0. \tag{2b}$$

At this stage, we will also assume that $n(x) = n_0$ for simplicity.

With the problem thus defined, we turn our attention to the key aspects in the computational implementation of the boundary value problem solver. These are

- **FOR** loops
- **IF** statements
- time-stepping algorithms: **ode23, ode45, ode113, ode15s**
- step-size control
- code development and flow.

Every code will be controlled by a set of FOR loops and IF statements. It is imperative to have proper placement of these control statements in order for the code to operate successfully.

5.1. Convergence. In addition to developing a successful code, it is reasonable to ask whether your numerical solution is actually correct. Thus far, the premise has been that discretization should provide an accurate approximation to the true solution provided the time-step Δt is small enough. Although in general this philosophy is correct, every numerical algorithm should be carefully checked to determine if it indeed converges to the true solution. The time-stepping schemes considered previously already hint at how the solutions should converge: fourth-order Runge–Kutta converges like Δt^4, second-order Runge–Kutta converges like Δt^2, and second-order predictor–corrector schemes converge like Δt^2. Thus the algorithm for checking convergence is as follows:

(1) Solve the differential equation using a time-step $\Delta t*$ which is very small. This solution will be considered the exact solution. Recall that we would in general like to take Δt as large as possible for efficiency purposes.

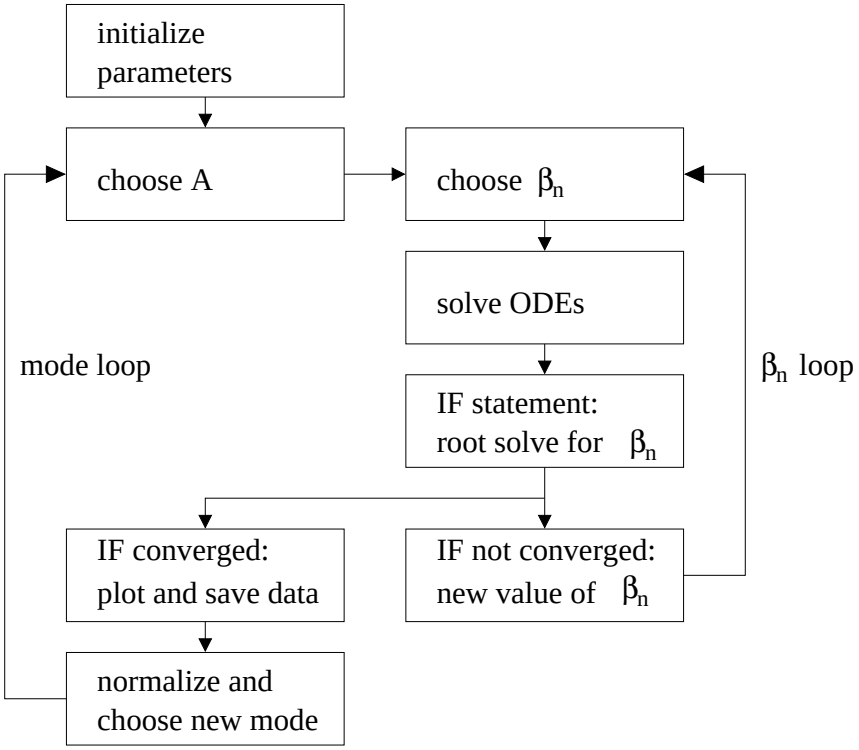

FIGURE 10. Basic algorithm structure for solving the boundary value problem. Two FOR loops are required to step through the values of β_n and A along with a single IF statement block to check for convergence of the solution

(2) Using a much larger time-step Δt, solve the differential equation and compare the numerical solution with that generated from $\Delta t*$. Cut this time-step in half and compare once again. In fact, continue cutting the time-step in half: $\Delta t, \Delta t/2, \Delta t/4, \Delta t/8, \ldots$ in order to compare the difference in the exact solution to this hierarchy of solutions.
(3) The difference between any run $\Delta t*$ and Δt is considered the error. Although there are many definitions of error, a practial error measurement is the root-mean square error $E = \left[(1/N)\sum_{i=1}^{N} |y_{\Delta t*} - y_{\Delta t}|^2\right]^{1/2}$. Once calculated, it is possible to verify the convergence law of Δt^2, for instance, with a second-order Runge–Kutta.

5.2. Flow control. In order to begin coding, it is always prudent to construct the basic structure of the algorithm. In particular, it is good to determine the number of FOR loops and IF statements which may be required for the computations. What is especially important is determining the hierarchic structure for the loops. To solve the boundary value problem proposed here, we require two FOR loops and one IF statement block. The outermost FOR loop of the code should determine the number of eigenvalues and eigenmodes to be searched for. Within this FOR loop there exists a second FOR loop which iterates the shooting method so that the solution converges to the correct boundary value solution. This second FOR loop has a logical IF statement which needs to check whether the solution has indeed converged to the boundary value solution, or whether adjustment of the value of β_n is necessary and the iteration procedure needs to be continued. Figure 10 illustrates the backbone of the numerical code for solving the boundary value problem. It includes the two FOR loops and logical IF statement block as the core of its algorithmic structure. For a nonlinear problem, a third FOR loop would be required for A in order to achieve the normalization of the eigenfunctions to unity.

The various pieces of the code are constructed here using the python programming language. We begin with the initialization of the parameters.

Initialization

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

tol = 1e-4  # define a tolerance level
col = ['r', 'b', 'g', 'c', 'm', 'k']  # eigenfunc colors
n0 = 100; A = 1; x0 = [0, A]; xp = [-1, 1]
xshoot =  np.linspace(xp[0], xp[1],1000)
```

Upon completion of the initialization process for the parameters which are not involved in the main loop of the code, we move into the main FOR loop which searches out a specified number of eigenmodes. Embedded in this FOR loop is a second FOR loop which attempts different values of β_n until the correct eigenvalue is found. An IF statement is used to check the convergence of values of β_n to the appropriate value.

Main program

```
def shoot2(x, dummy, n0, beta):
    return [x[1], (beta - n0) * x[0]]

beta_start = n0  # beginning value of beta
for modes in range(1, 6):  # begin mode loop
    beta = beta_start  # initial value of eigenvalue beta
    dbeta = n0 / 100  # default step size in beta
    for _ in range(1000):  # begin convergence loop for beta
        y = odeint(shoot2, x0, xshoot, args=(n0,beta))

        if abs(y[-1, 0] - 0) < tol:  # check for convergence
            print(beta)  # write out eigenvalue
            break  # get out of convergence loop

        if (-1) ** (modes + 1) * y[-1, 0] > 0:
            beta -= dbeta
        else:
            beta += dbeta / 2
            dbeta /= 2

    beta_start = beta - 0.1  # after finding eigenvalue, pick new start
    norm = np.trapz(y[:, 0] * y[:, 0], xshoot)  # calculate the normalization
    plt.plot(xshoot, y[:, 0] / np.sqrt(norm), col[modes - 1])  # plot modes
```

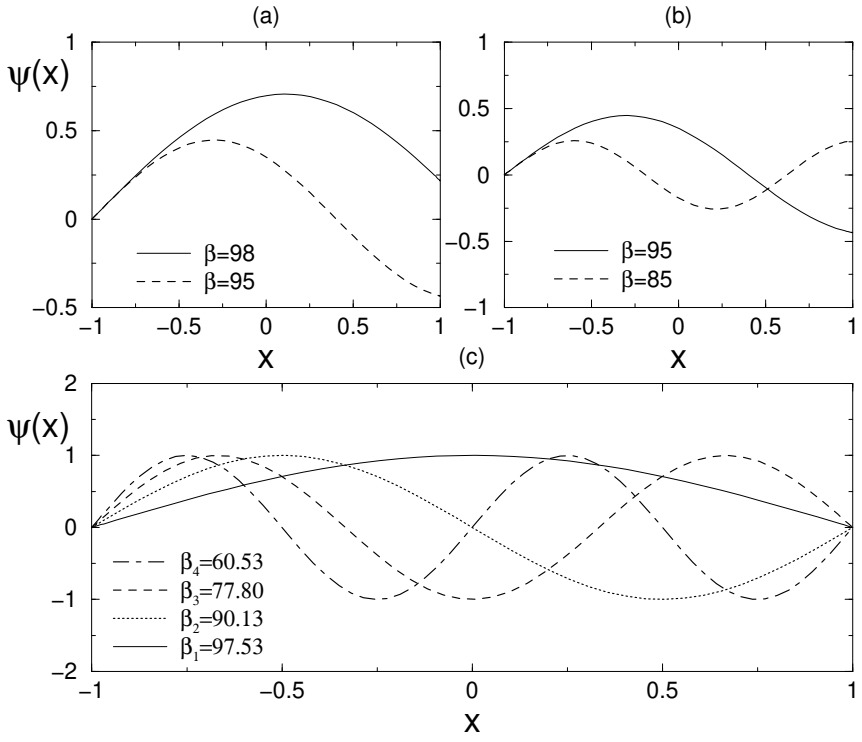

FIGURE 11. In (a) and (b) the behavior of the solution near the first even and first odd solution is depicted. Note that for the even modes increasing values of β bring the solution from $\psi_n(1) > 0$ to $\psi_n(1) < 0$. In contrast, odd modes go from $\psi_n(1) < 0$ to $\psi_n(1) > 0$ as β is increased. In (c) the first four normalized eigenmodes along with their corresponding eigenvalues are illustrated for $n_0 = 100$.

The code uses *ode45*, which is a fourth-order Runge–Kutta method, to solve the differential equation and advance the solution. The function *shoot2.m* is called in this routine.

This code will find the first five eigenvalues and plot their corresponding normalized eigenfunctions. The bisection method implemented to adjust the values of β_n to find the boundary value solution is based upon observations of the structure of the even and odd eigenmodes. In general, it is always a good idea to first explore the behavior of the solutions of the boundary value problem before writing the shooting routine. This will give important insights into the behavior of the solutions and will allow for a proper construction of an accurate and efficient bisection method. Figure 11 illustrates several characteristic features of this boundary value problem. In Figs. 11(a) and 11(b), the behavior of the solution near the first even and first odd solution is exhibited. From Fig. 11(a) it is seen that for the even modes increasing values of β bring the solution from $\psi_n(1) > 0$ to $\psi_n(1) < 0$. In contrast, odd modes go from $\psi_n(1) < 0$ to $\psi_n(1) > 0$ as β is increased. This observation forms the basis for the bisection method developed in the code. Figure 11(c) illustrates the first four normalized eigenmodes along with their corresponding eigenvalues.

6. Boundary Value Problems: Direct Solve and Relaxation

The shooting method is not the only method for solving boundary value problems. The direct method of solution relies on Taylor expanding the differential equation itself. For linear problems, this results in a matrix problem of the form $\mathbf{Ax} = \mathbf{b}$. For nonlinear problems, a nonlinear system of equations must be solved using a relaxation scheme, i.e. a Newton or secant method. The

prototypical example of such a problem is the second-order boundary value problem

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \tag{1}$$

on $t \in [a, b]$ with the general boundary conditions

$$\alpha_1 y(a) + \beta_1 \frac{dy(a)}{dt} = \gamma_1 \tag{2a}$$

$$\alpha_2 y(b) + \beta_2 \frac{dy(b)}{dt} = \gamma_2. \tag{2b}$$

Thus the solution is defined over a specific interval and must satisfy the relations (2) at the end points of the interval.

Before considering the general case, we simplify the method by considering the linear boundary value problem

$$\frac{d^2y}{dt^2} = p(t)\frac{dy}{dt} + q(t)y + r(t) \tag{3}$$

on $t \in [a, b]$ with the simplified boundary conditions

$$y(a) = \alpha \tag{4a}$$

$$y(b) = \beta. \tag{4b}$$

Taylor expanding the differential equation and boundary conditions will generate the linear system of equations which solve the boundary value problem.

To see how the Taylor expansions are useful, consider the following two Taylor series:

$$f(t + \Delta t) = f(t) + \Delta t \frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2f(t)}{dt^2} + \frac{\Delta t^3}{3!}\frac{d^3f(c_1)}{dt^3} \tag{5a}$$

$$f(t - \Delta t) = f(t) - \Delta t \frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2f(t)}{dt^2} - \frac{\Delta t^3}{3!}\frac{d^3f(c_2)}{dt^3} \tag{5b}$$

where $c_1 \in [t, t+\Delta t]$ and $c_2 \in [t, t-\Delta t]$. Subtracting these two expressions gives

$$f(t + \Delta t) - f(t - \Delta t) = 2\Delta t \frac{df(t)}{dt} + \frac{\Delta t^3}{3!}\left(\frac{d^3f(c_1)}{dt^3} + \frac{d^3f(c_2)}{dt^3}\right). \tag{6}$$

By using the mean value theorem of calculus, we find $f'''(c) = (f'''(c_1) + f'''(c_2))/2$. Upon dividing the above expression by $2\Delta t$ and rearranging, we find the following expression for the first derivative:

$$\frac{df(t)}{dt} = \frac{f(t + \Delta t) - f(t - \Delta t)}{2\Delta t} - \frac{\Delta t^2}{6}\frac{d^3f(c)}{dt^3} \tag{7}$$

where the last term is the truncation error associated with the approximation of the first derivative using this particular Taylor series generated expression. Note that the truncation error in this case is $O(\Delta t^2)$. We could improve on this by continuing our Taylor expansion and truncating it at higher orders in Δt. This would lead to higher accuracy schemes. Further, we could also approximate the second, third, fourth and higher derivatives using this technique. It is also possible to generate backward and forward difference schemes by using points only behind or in front of the current point, respectively. Tables 1–3 summarize the second-order and fourth-order central difference schemes along with the forward- and backward-difference formulas which are accurate to second order.

To solve the simplified linear boundary value problem above, which is accurate to second order, we use Table 1 for the second and first derivatives. The boundary value problem then becomes

$$\frac{y(t+\Delta t) - 2y(t) + y(t-\Delta t)}{\Delta t^2} = p(t)\frac{y(t+\Delta t) - y(t-\Delta t)}{2\Delta t} + q(t)y(t) + r(t) \tag{8}$$

with the boundary conditions $y(a) = \alpha$ and $y(b) = \beta$. We can rearrange this expression to read

$$\left[1 - \frac{\Delta t}{2}p(t)\right] y(t + \Delta t) - \left[2 + \Delta t^2 q(t)\right] y(t) + \left[1 + \frac{\Delta t}{2}p(t)\right] y(t - \Delta t) = \Delta t^2 r(t). \tag{9}$$

We discretize the computational domain and denote $t_0 = a$ to be the left boundary point and $t_N = b$ to be the right boundary point. This gives the boundary conditions

$$y(t_0) = y(a) = \alpha \tag{10a}$$
$$y(t_N) = y(b) = \beta. \tag{10b}$$

The remaining $N - 1$ points can be recast as a matrix problem $\mathbf{Ax} = \mathbf{b}$ where

$$\mathbf{A} = \begin{bmatrix} 2+\Delta t^2 q(t_1) & -1+\frac{\Delta t}{2}p(t_1) & 0 & \cdots & & 0 \\ -1-\frac{\Delta t}{2}p(t_2) & 2+\Delta t^2 q(t_2) & -1+\frac{\Delta t}{2}p(t_2) & 0 & \cdots & \vdots \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & & & \ddots & \ddots & -1+\frac{\Delta t}{2}p(t_{N-2}) \\ 0 & \cdots & & 0 & -1-\frac{\Delta t}{2}p(t_{N-1}) & 2+\Delta t^2 q(t_{N-1}) \end{bmatrix} \tag{11}$$

and

$$\mathbf{x} = \begin{bmatrix} y(t_1) \\ y(t_2) \\ \vdots \\ y(t_{N-2}) \\ y(t_{N-1}) \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} -\Delta t^2 r(t_1) + (1 + \Delta t p(t_1)/2) y(t_0) \\ -\Delta t^2 r(t_2) \\ \vdots \\ -\Delta t^2 r(t_{N-2}) \\ -\Delta t^2 r(t_{N-1}) + (1 - \Delta t p(t_{N-1})/2) y(t_N) \end{bmatrix}. \tag{12}$$

Thus the solution can be found by a direct solve of the linear system of equations.

6.1. Nonlinear systems. A similar solution procedure can be carried out for nonlinear systems. However, difficulties arise from solving the resulting set of nonlinear algebraic equations. We can once again consider the general differential equation and expand with second-order accurate schemes:

$$y'' = f(t, y, y') \;\rightarrow\; \frac{y(t+\Delta t) - 2y(t) + y(t-\Delta t)}{\Delta t^2} = f\left(t, y(t), \frac{y(t+\Delta t) - y(t-\Delta t)}{2\Delta t}\right). \tag{13}$$

We discretize the computational domain and denote $t_0 = a$ to be the left boundary point and $t_N = b$ to be the right boundary point. Considering again the simplified boundary conditions $y(t_0) = y(a) = \alpha$ and $y(t_N) = y(b) = \beta$ gives the following nonlinear system for the remaining $N - 1$ points.

$$\begin{aligned} 2y_1 - y_2 - \alpha + \Delta t^2 f(t_1, y_1, (y_2 - \alpha)/2\Delta t) &= 0 \\ -y_1 + 2y_2 - y_3 + \Delta t^2 f(t_2, y_2, (y_3 - y_1)/2\Delta t) &= 0 \\ \vdots & \\ -y_{N-3} + 2y_{N-2} - y_{N-1} + \Delta t^2 f(t_{N-2}, y_{N-2}, (y_{N-1} - y_{N-3})/2\Delta t) &= 0 \\ -y_{N-2} + 2y_{N-1} - \beta + \Delta t^2 f(t_{N-1}, y_{N-1}, (\beta - y_{N-2})/2\Delta t) &= 0. \end{aligned}$$

This $(N-1)\times(N-1)$ nonlinear system of equations can be very difficult to solve and imposes a severe constraint on the usefulness of the scheme. However, there may be no other way of solving the problem and a solution to this system of equations must be computed. Further complicating the issue is the fact that for nonlinear systems such as these, there are no guarantees about the existence or uniqueness of solutions. The best approach is to use a *relaxation* scheme which is based upon Newton or secant method iterations.

7. Implementing python for Boundary Value Problems

Both a shooting technique and a direct discretization method have been developed here for solving boundary value problems. More generally, one would like to use a high-order method that is robust and capable of solving general, nonlinear boundary value problems. python provides a convenient and easy to use routine, known as **bvp4c**, that is capable of solving fairly sophisticated problems. The algorithm relies on an iteration structure for solving nonlinear systems of equations. In particular, **bvp4c** is a finite difference code that implements the three-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a C^1-continuous solution that is fourth-order accurate uniformly in $x\in[a,b]$. Mesh selection and error control are based on the residual of the continuous solution. Since it is an iteration scheme, its effectiveness will ultimately rely on your ability to provide the algorithm with an initial guess for the solution.

Two example codes will be demonstrated here. The first is a simple linear, constant coefficient second-order equation with fairly standard boundary conditions. The second example is nonlinear with an undetermined parameter (eigenvalue) that must also be determined. Both illustrate the power and ease of use of the built-in boundary value solver of python.

7.1. Linear boundary value problem. As a simple and particular example of a boundary value problem, consider the following:

$$y''+3y'+6y=5 \tag{1}$$

on the domain $x\in[1,3]$ and with boundary conditions

$$y(1) = 3 \tag{2a}$$
$$y(3)+2y'(3) = 5. \tag{2b}$$

This problem can be solved in a fairly straightforward manner using analytic techniques. However, we will pursue here a numerical solution instead.

As with any differential equation solver, the equation must first be put into the form of a system of first-order equations. Thus by introducing the variables

$$y_1 = y(x) \tag{3a}$$
$$y_2 = y'(x) \tag{3b}$$

we can rewrite the governing equations as

$$y_1' = y_2 \tag{4a}$$
$$y_2' = 5-3y_2-6y_1 \tag{4b}$$

with the transformed boundary conditions

$$y_1(1) = 3 \tag{5a}$$
$$y_1(3)+2y_2(3) = 5. \tag{5b}$$

In order to implement the boundary value problem in python, the boundary conditions need to be placed in the general form

$$f(y_1,y_2) = 0 \ \text{ at } \ x=x_L \tag{6a}$$
$$g(y_1,y_2) = 0 \ \text{ at } \ x=x_R \tag{6b}$$

where $f(y_1, y_2)$ and $g(y_1, y_2)$ are the boundary value functions at the left (x_L) and right (x_R) boundary points. This then allows us to rewrite the boundary conditions in (5) as the following:

$$f(y_1, y_2) = y_1 - 3 = 0 \qquad \text{at } x_L = 1 \tag{7a}$$

$$g(y_1, y_2) = y_1 + 2y_2 - 5 = 0 \qquad \text{at } x_R = 3\,. \tag{7b}$$

The formulation of the boundary value problem is then completely specified by the differential equation (4) and its boundary conditions (7).

The boundary value solver **bvp4c** requires three pieces of information: the equation to be solved, its associated boundary conditions, and your initial guess for the solution. The first two lines of the following code performs all three of these functions:

```
from scipy.integrate import solve_bvp
import matplotlib.pyplot as plt

def bvp_rhs(x, y):
    return np.vstack((y[1], 5 - 3 * y[1] - 6 * y[0]))

def bvp_bc(ya, yb):
    return np.array([ya[0] - 3, yb[0] + 2 * yb[1] - 5])

# Define initial mesh and guess
x_init = np.linspace(1, 3, 10)
y_init = np.zeros((2, x_init.size))

# Solve the boundary value problem
sol = solve_bvp(bvp_rhs, bvp_bc, x_init, y_init)

# Evaluate the solution on a finer mesh
x_eval = np.linspace(1, 3, 100)
BS = sol.sol(x_eval)
plt.plot(x_eval, BS[0])
```

We dissect this code by first considering the first line of code for generating a python data structure for use as the initial data. In this initial line of code, the **linspace** command is used to define the initial mesh that goes from $x = 1$ to $x = 3$ with 10 equally spaced points. In this example, the initial values of $y_1(x) = y(x)$ and $y_2(x) = y'(x)$ are zero. If you have no guess, then this is probably your best guess to start with. Thus you not only guess the initial guess, but the initial grid on which to find the solution.

Once the guess and initial mesh is generated, two functions are called with the **@bvp_rhs** and **@bvp_bc** function calls. These are functions representing the differential equation and boundary conditions (4) and (7),
Note that what is passed into the boundary condition function is the value of the vector **y** at the left (**yL**) and right (**yR**) of the computational domain, i.e. at the values of $x = 1$ and $x = 3$.

What is produced by **bvp4c** is a data structure **sol**. This data structure contains a variety of information about the problem, including the solution $y(x)$. To extract the solution, the final two lines of code in the main program are used. Specifically, the **deval** command evaluates the solution at the points specified by the vector **x**, which in this case is a linear space of 100 points between one and three. The solution can then simply be plotted once the values of $y(x)$ have been extracted. Figure 12 shows the solution and its derivative that are produced from the boundary value solver.

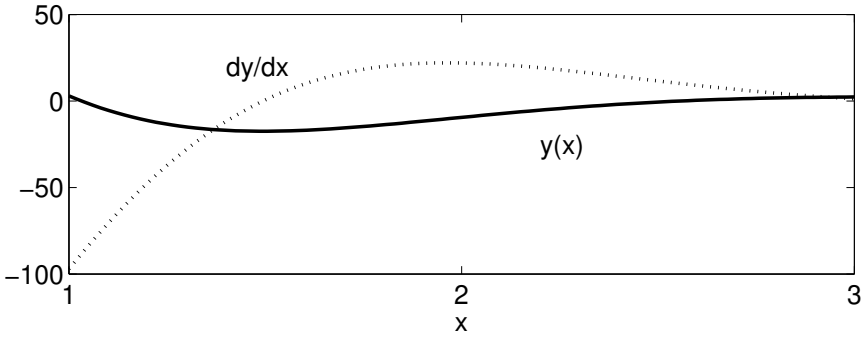

FIGURE 12. The solution of the boundary value problem (4) with (7). The solid line is $y(x)$ while the dotted line is its derivative $dy(x)/dx$.

7.2. Nonlinear eigenvalue problem. As a more sophisticated example of a boundary value problem, we will consider a fully nonlinear eigenvalue problem. Thus consider the following:

$$y'' + (100 - \beta)y + \gamma y^3 = 0 \tag{8}$$

on the domain $x \in [-1, 1]$ and with boundary conditions

$$y(-1) = 0 \tag{9a}$$
$$y(1) = 0. \tag{9b}$$

This problem cannot be solved using analytic techniques due to the complexity introduced by the nonlinearity. But a numerical solution can be fairly easily constructed. Note the similarity between this problem and that considered in the shooting section.

As before, the equation must first be put into the form of a system of first-order equations. Thus by introducing the variables

$$y_1 = y(x) \tag{10a}$$
$$y_2 = y'(x) \tag{10b}$$

we can rewrite the governing equations as

$$y_1' = y_2 \tag{11a}$$
$$y_2' = (\beta - 100)y_1 - \gamma y_1^3 \tag{11b}$$

with the transformed boundary conditions

$$y_1(-1) = 0 \tag{12a}$$
$$y_1(1) = 0. \tag{12b}$$

The formulation of the boundary value problem is then completely specified by the differential equation (11) and its boundary conditions (12). Note that unlike before, we do not know the parameter β. Thus it must be determined along with the solution. As with the initial conditions, we will also guess an initial value of β and let **bvp4c** converge to the appropriate value of β. Note that for such nonlinear problems, the effectiveness of **bvp4c** relies almost exclusively on providing a good initial guess.

As before, the boundary value solver **bvp4c** requires three pieces of information: the equation to be solved, its associated boundary conditions, and your initial guess for the solution and the parameter β. The first line of the code gives the initial guess for β while the next two lines of the following code perform the three remaining functions:

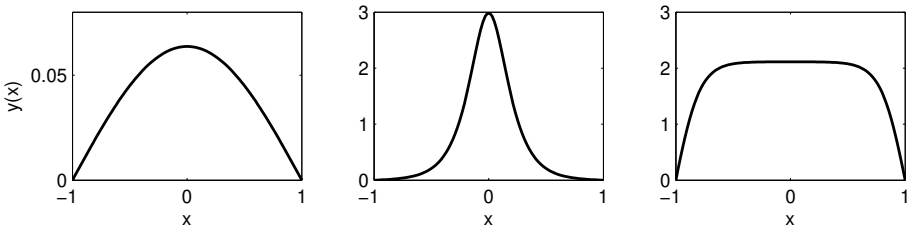

FIGURE 13. Three different nonlinear eigenvalue solutions to (11) and its boundary conditions (12). The left panel has the constraint condition $y'(-1) = 0.1$ with $\gamma = 1$, the middle panel has $y'(-1) = 0.1$ with $\gamma = 10$ and the right panel has $y'(-1) = 10$ with $\gamma = -10$. Such solutions are called *nonlinear ground states* of the eigenvalue problem.

```
import numpy as np
from scipy.integrate import solve_bvp
import matplotlib.pyplot as plt

def bvp_rhs2(x, y, beta):
    return np.vstack((y[1], (beta - 100) * y[0] - y[0]**3))

def bvp_bc2(yl, yr, beta):
    return np.array([yl[0], yl[1] - 0.1, yr[0]])

def mat4init(x):
    return np.array([np.cos((np.pi/2)*x),-(np.pi / 2)*np.sin((np.pi/2)*x)])

beta = 99
x_init = np.linspace(-1, 1, 50)
init2 = solve_bvp(bvp_rhs2, bvp_bc2, x_init, mat4init(x_init), p=[beta])

x2 = np.linspace(-1, 1, 100)
BS2 = init2.sol(x2)
plt.plot(x2, BS2[0])
```

In this case, there are now three function calls: **@bvp_rhs2**, **@bvp_bc2** and **@mat4init**. This calls the differential equation, its boundary values and its initial guess, respectively.

We begin with the initial guess. Our implementation example of the shooting method showed that the first linear solution behaved like a cosine function. Thus we can guess that $y_1(x) = y(x) = \cos[(\pi/2)x]$ where the factor of $\pi/2$ is chosen to make the solution zero at $x = \pm 1$. Note that initially 50 points are chosen between $x = -1$ and $x = 1$.

Again, it must be emphasized that the success of **bvp4c** relies almost exclusively on the above subroutine and guess. Without a good guess, especially for nonlinear problems, you may find a solution, but just not the one you want.

The equation itself is handled in the subroutine **bvp_rhs2.m**.

Finally, the implementation of the boundary conditions, or constraints, must be imposed. There is something very important to note here: we started with two constraints since we have a second-order differential equation, i.e. $y(\pm 1) = 0$. However, since we do not know the value of β, the system is currently an underdetermined system of equations. In order to remedy this, we need to

impose one more constraint on the system. This is somewhat arbitrary unless there is a natural constraint for you to choose in the system. Here, we will simply choose $dy(-1)/dx = 0.1$, i.e. we will impose the launch angle at the left.

With these three constraints, the **bvp4c** iteration routine not only finds the solution, but also the appropriate value of β that satisfies the above constraints.

Executing the routine and subroutines generates the solution to the boundary value problem. One can also experiment with different guesses and values of β to generate new and interesting solutions. Three such solutions are demonstrated in Fig. 13 as a function of the β value and the guess angle $dy(-1)/dx$.

8. Linear Operators and Computing Spectra

As a final application of boundary value problems, we will consider the ability to accurately compute the spectrum of linear operators. Linear operators often arise in the context of evaluating the stability of solutions to partial differential equations. Indeed, stability plays a key role in many branches of science and engineering, including aspects of fluid mechanics, pattern formation with reaction–diffusion models, high-speed transmission of optical information, and the feasibility of MHD fusion devices, to name a few. If one can find solutions for a given particular differential equation, then the stability of that solution becomes critical in determining the ultimate behavior of the system. Specifically, if a physical phenomenon is observable and persists, then the corresponding solution to a valid mathematical model should be stable. If, however, instability is established, the nature of the unstable modes suggest what patterns may develop from the unstable solutions. Finally, for many problems of physical interest, fundamental mathematical models are well established. However, in many cases these fundamental models are too complicated to allow for detailed analysis, thus leading to the study of simpler approximate (linear) models using reductive perturbation methods.

To further illustrate these concepts, consider a generic partial differential equation of the form

$$\frac{\partial u}{\partial t} = N\left(x, u, \frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}, \cdots\right) \tag{1}$$

where the function N is generically nonconstant coefficient in x and a nonlinear function of $u(x,t)$ and its derivatives. Here, we will assume that this function is well behaved so that solutions can exist by the basic Cauchy–Kovaleskaya theorem. In addition, there are boundary conditions associated with Eq. (1). Such boundary constraints may be imposed on either a finite or infinite domain, and an example of each case will be considered in what follows.

In what follows, only the stability of equilibrium solutions will be considered. Equilibrium solutions are found when $\partial u/\partial t = 0$. Thus, there would exist a time-independent solution $U(x)$ such that

$$N\left(x, U, \frac{\partial U}{\partial x}, \frac{\partial^2 U}{\partial x^2}, \cdots\right) = 0. \tag{2}$$

If one could find such a solution $U(x)$, then its stability could be computed. Again, if the solution is stable, then it may be observable in the physical system. If it is unstable, then it will not be observed in practice. However, the unstable modes of such a system may potentially give clues to the resulting behavior of the system.

To generate a linear stability problem for the equilibrium solution $U(x)$ of (1), one would *linearize* about the solution so that

$$u(x,t) = U(x) + \epsilon v(x,t) \tag{3}$$

where the parameter $\epsilon \ll 1$ so that the function $v(x,t)$ is only a small perturbation from the equilibrium solution. If the function $v(x,t) \to \infty$ as $t \to \infty$, the system is said to be unstable. If $v(x,t) \to 0$ as $t \to \infty$, the system is said to be asymptotically stable. If $v(x,t)$ remains $O(1)$ as

$t \to \infty$, the system is said to simply be stable. Thus a determination of what happens to $v(x,t)$ is required. Plugging in (3) into (1) and Taylor expanding using the fact that $\epsilon \ll 1$ gives the equation

$$\frac{\partial v}{\partial t} = \mathcal{L}\left[U(x)\right] v + O(\epsilon) \tag{4}$$

where the $O(\epsilon)$ represents all the terms from the Taylor expansion that are $O(\epsilon)$ or smaller. Finally, by assuming that the function $v(x,t)$ takes the following form:

$$v(x,t) = w(x)\exp(\lambda t) \tag{5}$$

the following *linear* eigenvalue problem (spectral problem) results

$$\mathcal{L}\left[U(x)\right] v = \lambda v \tag{6}$$

where $\mathcal{L}$ is the linear operator associated with the stability of the equilibrium solution $U(x)$. Note that by our definitions of stability above and the solution form (5), the equilibrium solution is unstable if any real part of the eigenvalue is positive: $\Re\{\lambda\} > 0$. Stability is established if $\Re\{\lambda\} \leq 0$, with asymptotic stability occurring if $\Re\{\lambda\} < 0$. Thus computing the eigenvalues of the linear operator is the critical step in evaluating stability.

8.1. Sturm–Liouville theory. Of course, anybody familiar with Sturm–Liouville theory [**29**] will recognize the importance of computing the spectra of the linearized operators. For Sturm–Liouville problems, the linear operator and its associated eigenvalues take on the form

$$\mathcal{L}v = -\frac{d}{dx}\left[p(x)\frac{dv}{dx}\right] + q(x)v = \lambda w(x)v \tag{7}$$

where $p(x) > 0$, $q(x)$ and $w(x) > 0$ are specified on the interval $x \in [a,b]$. In the simplest case, these are continuous on the entire interval. The associated boundary conditions are given by

$$\alpha_1 v(a) + \alpha_2 \frac{dv(a)}{dx} = 0 \quad \text{and} \quad \beta_1 v(b) + \beta_2 \frac{dv(b)}{dx} = 0. \tag{8}$$

It is known that the Sturm–Liouville problem has some very nice properties, including the fact that the eigenvalues are all real and distinct. Many classic physical systems of interest naturally fall within Sturm–Liouville theory, thus propagating a variety of special functions such as Bessel functions, Legendre functions, Laguerre polynomials, etc. Indeed, much of special-function theory involves Sturm–Liouville theory at its core. In certain cases, the ideas of Sturm–Liouville can be extended to finding solutions to nonlinear eigenvalue problems [**35**].

8.2. Derivate operators. Although much is known in the literature about Sturm–Liouville theory, the theory often relates properties about the eigenfunctions and eigenvectors in the abstract. To compute the actual spectra of (7), we will advocate a numerical procedure based upon finite difference discretization. Shooting methods can also be applied, but they will not be considered in what follows. What is immediately apparent from (7), and more generally (6), is the presence of derivative operators. From the finite difference perspective, these are no more than matrix operations on the discretized (vector) solution. Following the ideas of Section 1, we can construct a number of first and second derivate matrices that are continually used in practice. The domain $x \in [a,b]$ is discretized into $N+1$ intervals so that $x_0 = a$ and $x_{N+1} = b$.

To begin, we will consider constructing first-derivative matrices that are $O(\Delta x^2)$ accurate. In this case, Table 1 applies. The simplest case to consider is when *Dirichlet boundary conditions* are applied, i.e. $v(a) = v(b) = 0$. The Dirichlet boundaries imply that only the interior points $v(x_j)$

where $j = 1, 2, \cdots, N$ must be solved for. The resulting derivative matrix is given by

$$\frac{\partial}{\partial x} \text{ with } u(a) = u(b) = 0 \;\rightarrow\; \mathbf{A}_1 = \frac{1}{2\Delta x}\begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ -1 & 0 & 1 & 0 & \cdots & \\ 0 & -1 & 0 & 1 & \cdots & \\ & & \vdots & & & \\ & \cdots & & -1 & 0 & 1 \\ 0 & \cdots & & 0 & -1 & 0 \end{bmatrix}. \tag{9}$$

If the boundary conditions are changed to *Neumann boundary conditions* with $dv(a)/dx = dv(b)/dx = 0$, then the derivative matrix changes. Specifically, if use is made of the forward- and backward-difference formulas of Table 3, then it is easy to show that

$$\frac{dv_0}{dx} = \frac{-3v_0 + 4v_1 - v_2}{2\Delta x} = 0 \;\rightarrow\; v_0 = \frac{4}{3}v_1 - \frac{1}{3}v_2. \tag{10}$$

Similarly at the right boundary, one can find

$$\frac{dv_{N+1}}{dx} = \frac{-3v_{N+1} + 4v_N - v_{N-1}}{2\Delta x} = 0 \;\rightarrow\; v_{N+1} = \frac{4}{3}v_N - \frac{1}{3}v_{N-1}. \tag{11}$$

Thus both boundary values, v_0 and v_{N+1}, are determined from the interior points alone. The final piece of information necessary is the following:

$$\frac{dv_1}{dx} = \frac{v_2 - v_0}{2\Delta x} = \frac{v_2 - [(4/3)v_1 - (1/3)v_2]}{2\Delta x} = \frac{(4/3)(v_2 - v_1)}{2\Delta x}. \tag{12}$$

A similar calculation can be done for dv_N/dx to yield the derivative matrix

$$\frac{\partial}{\partial x} \text{ with } \frac{du(a)}{dx} = \frac{du(b)}{dx} = 0 \;\rightarrow\; \mathbf{A}_2 = \frac{1}{2\Delta x}\begin{bmatrix} -4/3 & 4/3 & 0 & 0 & \cdots & 0 \\ -1 & 0 & 1 & 0 & \cdots & \\ 0 & -1 & 0 & 1 & \cdots & \\ & & \vdots & & & \\ & \cdots & & -1 & 0 & 1 \\ 0 & \cdots & & 0 & -4/3 & 4/3 \end{bmatrix}. \tag{13}$$

Finally, we consider the case of *periodic boundary conditions* for which $v(a) = v(b)$. This implies that $v_0 = v_{N+1}$. Thus when considering this case, the unknowns include v_0 through v_N, i.e. there are $N + 1$ unknowns as opposed to N unknowns. The derivative matrix in this case is

$$\frac{\partial}{\partial x} \text{ with } u(a) = u(b) \;\rightarrow\; \mathbf{A}_3 = \frac{1}{2\Delta x}\begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & -1 \\ -1 & 0 & 1 & 0 & \cdots & \\ 0 & -1 & 0 & 1 & \cdots & \\ & & \vdots & & & \\ & \cdots & & -1 & 0 & 1 \\ 1 & \cdots & & 0 & -1 & 0 \end{bmatrix} \tag{14}$$

where the top right and bottom left corners come from the fact that, for instance, $dv_N/dx = (v_{N+1} - v_{N-1})/(2\Delta x) = (v_0 - v_{N-1})/(2\Delta x)$.

Second-derivative analogs can also be created for these matrices using the same basic ideas as for first derivatives. In this case, we find for Dirichlet boundary conditions

$$\frac{\partial^2}{\partial x^2} \text{ with } u(a)=u(b)=0 \;\rightarrow\; \mathbf{B}_1=\frac{1}{\Delta x^2}\begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & \\ 0 & 1 & -2 & 1 & \cdots & \\ & & \vdots & & & \\ & \cdots & & 1 & -2 & 1 \\ 0 & \cdots & & 0 & 1 & -2 \end{bmatrix}. \tag{15}$$

For Neumann boundary conditions, the following applies

$$\frac{\partial^2}{\partial x^2} \text{ with } \frac{du(a)}{dx}=\frac{du(b)}{dx}=0 \;\rightarrow\; \mathbf{B}_2=\frac{1}{\Delta x^2}\begin{bmatrix} -2/3 & 2/3 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & \\ 0 & 1 & -2 & 1 & \cdots & \\ & & \vdots & & & \\ & \cdots & & 1 & -2 & 1 \\ 0 & \cdots & & 0 & 2/3 & -2/3 \end{bmatrix}. \tag{16}$$

where the boundaries are again evaluated from the interior points through the relations (10) and (11). Finally, for periodic boundaries the matrix is

$$\frac{\partial^2}{\partial x^2} \text{ with } u(a)=u(b) \;\rightarrow\; \mathbf{B}_3=\frac{1}{\Delta x^2}\begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 1 \\ 1 & -2 & 1 & 0 & \cdots & \\ 0 & 1 & -2 & 1 & \cdots & \\ & & \vdots & & & \\ & \cdots & & 1 & -2 & 1 \\ 1 & \cdots & & 0 & 1 & -2 \end{bmatrix}. \tag{17}$$

The derivative operators shown here are fairly standard and can easily be implemented in python. The following code constructs matrices $\mathbf{A}_1$–$\mathbf{A}_3$ and $\mathbf{B}_1$ $\mathbf{B}_3$. First, there is the construction of the first-derivative matrices:

```
# Assume N and dx given
A = np.zeros((N, N))
for j in range(N - 1):
    A[j, j + 1] = 1
    A[j + 1, j] = -1
A1 = A / (2 * dx) # Dirichlet matrices

A2 = np.copy(A)
A2[0, 0] = -4 / 3
A2[0, 1] = 4 / 3
A2[N - 1, N - 1] = 4 / 3
A2[N - 1, N - 2] = -4 / 3
A2 = A2 / (2 * dx) # Neumann matrices

A3 = np.copy(A)
A3[N - 1, 0] = 1
A3[0, N - 1] = -1
A3 = A3 / (2 * dx) # Periodic BC matrices
```

The second-derivative matrices can be constructed similarly with the following python code

```
B = np.zeros((N, N))
for j in range(N):
    B[j, j] = -2
for j in range(N - 1):
    B[j, j + 1] = 1
    B[j + 1, j] = 1
B1 = B / (dx**2) # Dirichlet matrices for B

B2 = np.copy(B)
B2[0, 0] = -2 / 3
B2[0, 1] = 2 / 3
B2[N - 1, N - 1] = -2 / 3
B2[N - 1, N - 2] = 2 / 3
B2 = B2 / (dx**2) # Neumann matrices for B

B3 = np.copy(B)
B3[N - 1, 0] = 1
B3[0, N - 1] = -1
B3 = B3 / (dx**2) # Periodic BC matrices for B
```

Given how often such derivative matrices are used, the above code could be used as part of function call to simply pull out the derivative matrix required. In addition, we will later learn how to encode such matrices in a sparse manner since they are mostly zeros.

8.3. Example: The harmonic oscillator. To see an example of how to compute the spectra of a linear operator, consider the example of the harmonic oscillator which has the well-known Hermite function solutions. This is a Sturm–Liouville problem with $p(x) = w(x) = 1$ and $q(x) = x^2$. The $q(x)$ plays the role of a trapping potential in quantum mechanics. Typically, the harmonic oscillator is specified on the domain $x \in [-\infty, \infty]$ with solutions $v(\pm\infty) \to 0$. Instead of specifying that the domain is infinite, we will consider a finite domain of size $x \in [-a, a]$ with the Dirichlet boundary conditions $v(\pm L) = 0$. Thus we have the eigenvalue problem

$$\mathcal{L}v = -\frac{d^2v}{dx^2} + x^2v = \lambda v \quad \text{with} \quad v(\pm L) = 0. \tag{18}$$

Our objective is to construct the spectra and eigenfunctions of this linear operator. We can even compare them directly to the known solutions to see how accurate the reconstruction is. The following code discretizes the domain and computes the spectra.

```
L = 4  # domain size
N = 200  # discretization of interior
x = np.linspace(-L, L, N + 2)  # add boundary points
dx = x[1] - x[0]  # compute dx

P = np.zeros((N, N)) # Compute P matrix
for j in range(N):
    P[j, j] = x[j + 1] ** 2  # potential x^2
```

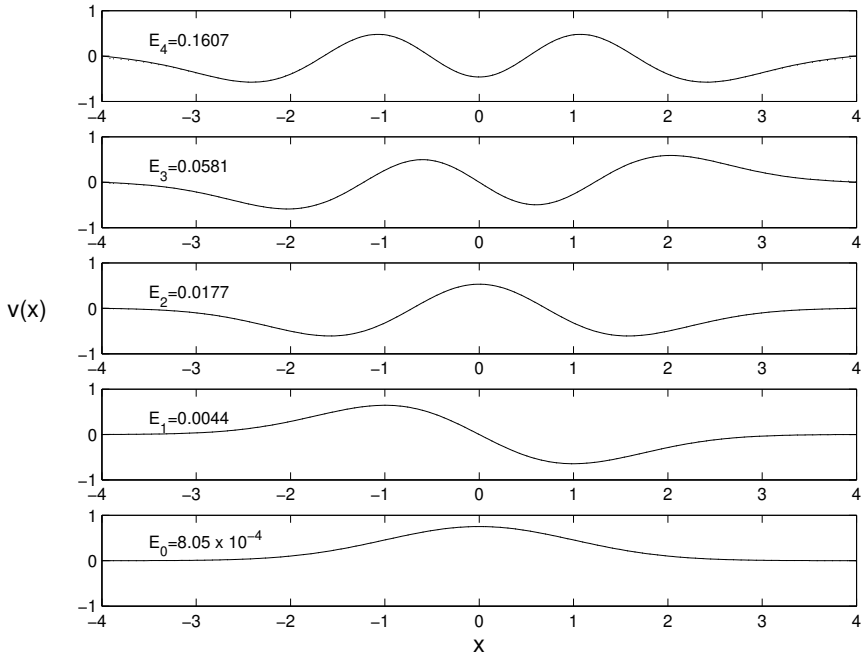

FIGURE 14. First five eigenfunctions of the harmonic oscillator computed using finite-difference discretization. Shown are the computed eigenfunctions (solid line) versus the exact solution (dotted line). The two lines are almost indistinguishable from each other. Thus the norm of the difference of the solutions is given by E_j for the different eigenfunctions.

```
linL = -B1 + P # Compute linear operator


D,V = eig(linL) # Compute eigenvalues/eigenvectors

sorted_indices = np.argsort(np.abs(D))[::-1]
Dsort = D[sorted_indices]
Vsort =V[:, sorted_indices]

D5 = Dsort[N-5:N]
V5 = Vsort[:,N-5:N]
```

This code computes the second-order accurate spectra of the harmonic oscillator. Here, $\Delta x \approx 0.04$ so that an accuracy of 10^{-3} is expected. Only the five smallest eigenvalues are returned in the code. These five eigenvalues are theoretically known to be given by $\lambda_n = (2n+1)$ with $n = 0, 1, 2, \cdots$. In our calculation we find the first five $\lambda_n = 1, 3, 5, 7, 9 \approx 0.9999, 2.9995, 4.9991, 7.0009, 9.0152$. Thus the accuracy is as prescribed. The first five eigenfunctions are shown in Fig. 14. As can be seen, the finite difference method is fairly easy to implement and yields fast and accurate results.

9. Neural Networks for Time Stepping

We return to the basic time-stepping algorithms introduced at the start of this chapter. The algorithms developed previously, including simple Euler stepping and Runge-Kutta, all involve approximating a Δt step into the future using Taylor series expansions. An alternative way to

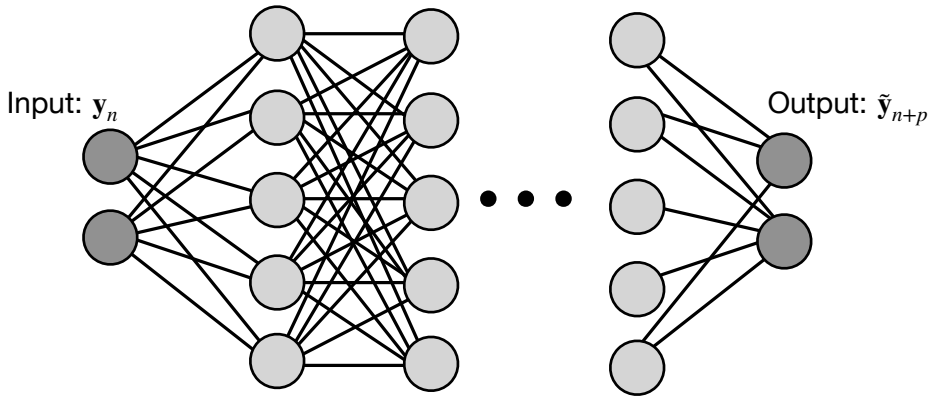

FIGURE 15. Feed forward neural network used as an approximation for time-stepping. The input $\mathbf{y}_n$ is mapped through a network to p steps into the future $\tilde{\mathbf{y}}_{n+p}$ by minimizing the error $\|\mathbf{y}_{n+p} - \tilde{\mathbf{y}}_{n+p}\|$.

think about a time-stepper is as a flow map from time t to time $t + \Delta t$ [**36, 37**]. Neural networks can learn such a flow map. Specifically, we can consider the basic input-output relationship of a neural network to be a flow map from time t to time $t + \Delta$:

$$\mathbf{Y} = f_{\boldsymbol{\theta}}(\mathbf{X}) \quad \rightarrow \quad \mathbf{y}_{n+1} = f_{\boldsymbol{\theta}}(\mathbf{y}_n) \tag{19}$$

where $f_{\boldsymbol{\theta}}(\cdot)$ is a trained neural network. Thus given trajectories of (1), the neural network weights $\boldsymbol{\theta}$ are updated to minimize the time-stepping error given a specific neural network architecture. Such an architecture effectively learns the right hand side function of (1) [**38, 39, 40, 41**].

As will all neural networks, training data is required in order to determine the weights of the activation functions. Figure 15 show an example feed forward architecture mapping the input $\mathbf{y}_n$ to $p\Delta t$ steps into the future $\tilde{\mathbf{y}}_{n+p}$ by minimizing the error $\|\mathbf{y}_{n+p} - \tilde{\mathbf{y}}_{n+p}\|$. In what follows, we train a neural network to take one step forward so that $p = 1$. The training data is comprised of 100 trajectories with randomly generated initial conditions. The data is mapped to input and output data which correspond to $\mathbf{y}_n$ and $\mathbf{y}_{n+1}$ respectively. The input-output data matrices are then made into torch objects.

```
def lorenz(t, x, sig=10, r=28, b=8/3):
    return [sig*(x[1]-x[0]), r*x[0]-x[1]-x[0]*x[2], x[0]*x[1]-b*x[2]]

dt = 0.01; T = 8; t = np.arange(0, T + dt, dt); n_trajectories = 100

input_data = []; output_data = []

for j in range(n_trajectories):
    x0 = 30 * (np.random.rand(3) - 0.5)
    sol = solve_ivp(lorenz, [0, T], x0, t_eval=t, atol=1e-11, rtol=1e-10)
    y = sol.y.T
    input_data.append(y[:-1])
```

```
        output_data.append(y[1:])

input_data = np.vstack(input_data)
output_data = np.vstack(output_data)

input_tensor = torch.tensor(input_data, dtype=torch.float32)
output_tensor = torch.tensor(output_data, dtype=torch.float32)
```

With the training data generated, a feed forward neural network is constructed. In this specific case, it is a three hidden layer network that takes the original three dimensional input space to 10 dimensions. The last layer is a linear layer that brings the output back to three dimensions. The activation functions are varied with both a sigmoid and ReLu layer being used, for instance. The following code defines the neural network class and structure.

```
class FeedforwardNN(nn.Module):
    def __init__(self):
        super(FeedforwardNN, self).__init__()
        self.fc1 = nn.Linear(3, 10)
        self.fc2 = nn.Linear(10, 10)
        self.fc3 = nn.Linear(10, 10)
        self.fc4 = nn.Linear(10, 3)
        self.sigmoid = nn.Sigmoid()
        self.radbas = nn.ReLU()
        self.linear = nn.Identity()

    def forward(self, x):
        x = self.sigmoid(self.fc1(x))
        x = self.radbas(self.fc2(x))
        x = self.linear(self.fc3(x))
        x = self.fc4(x)
        return x
```

Training now requires a loss function and various hyper parameter settings, which includes the number of epochs, batch size, loss rate, optimizer and error function. These can all be easily adjusted in the training.

```
net = FeedforwardNN()
criterion = nn.MSELoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)
n_epochs = 200; batch_size = 32

for epoch in range(n_epochs):
    permutation = torch.randperm(input_tensor.size()[0])
    for i in range(0, input_tensor.size()[0], batch_size):
        indices = permutation[i:i + batch_size]
        batch_input, batch_output = input_tensor[indices], output_tensor[indices]

        optimizer.zero_grad()
        outputs = net(batch_input)
```

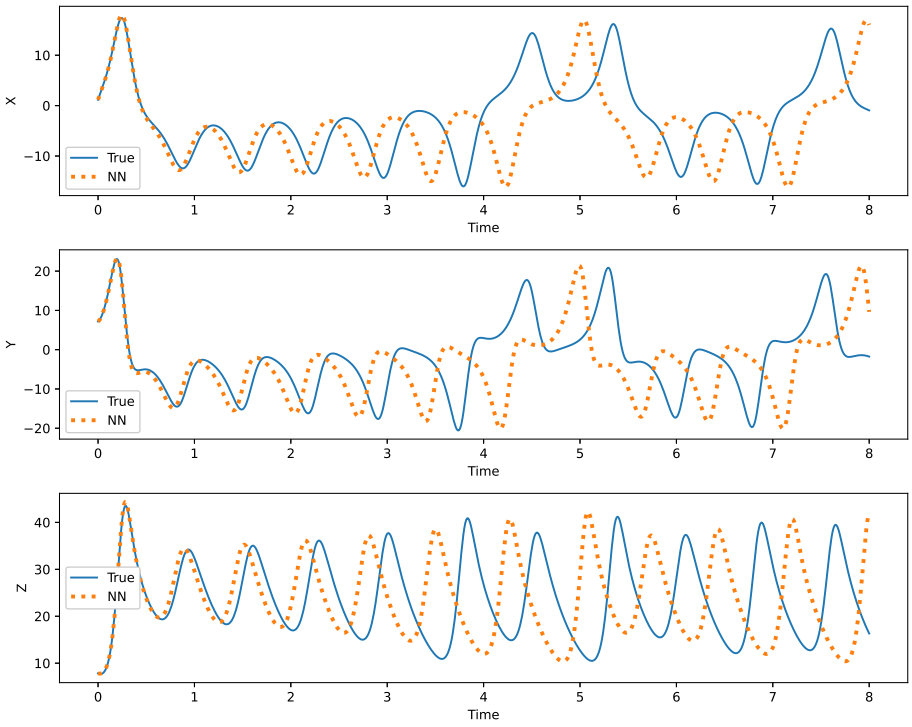

FIGURE 16. Learned neural network for time stepping. A new (test) initial condition is used to roll out the trajectory to the future. The salient features are learned. However, it is expected for Lorenz equation that the solution would diverge from the true trajectory due to sensitivity to initial conditions. More training data, deeper and wider networks, and often longer training times can greatly improve the neural networks capabilities.

```
loss = criterion(outputs, batch_output)
loss.backward()
optimizer.step()
```

Given the number of hyper parameters, it is often the case that hyper parameter sweeps are performed in order to increase the performance of the neural network. Although this particular neural network is small and can be easily trained in minutes, hyper parameter tuning on larger problems takes substantial amount of time, effort and patience.

Once the neural network is trained, it can be deployed with new initial data as shown in Fig. 16. The following code uses the neural network to *unroll* to future states. These can then be compared to actual simulations to verify their behavior.

```
ynn = [x0]
x0_tensor = torch.tensor(x0, dtype=torch.float32).unsqueeze(0)
for _ in range(1, len(t)):
    y0_tensor = net(x0_tensor)
    ynn.append(y0_tensor.detach().numpy().flatten())
    x0_tensor = y0_tensor

ynn = np.array(ynn)
```

The trained neural network can be deployed as a proxy for a numerical time-stepping algorithm such as Runge-Kutta. Unlike standard numerical steppers, however, neural networks rarely have rigorous estimates for convergence. Rather, cross-validation is used for determining the convergence properties. More sophisticated algorithms can also be applied to learn multiscale features for time-stepping [**36**]. Additionally, such techniques can be used in equation free architectures where no governing equations are known, but data is generated from sensors directly. This is something that is not possible with numerical steppers which require governing equations.

10. Problems and Exercises

10.1. Neuroscience and the Hodgkin–Huxley Model. In 1952, Alan Hodgkin and Andrew Huxley published a series of five papers detailing the mechanisms underlying the experimentally observed dynamics and propagation of voltage spikes in the giant squid axon (see Fig. 17). This seminal work eventually led them to receive the 1963 Nobel Prize in Physiology and Medicine. Not only were the experimental techniques and characterization of the ion channels unparalleled at the time, but Hodgkin and Huxley went further to develop the underpinnings of modern theoretical/computational neuroscience [**42, 43**]. Given its transformative role in neuroscience, the Hodgkin–Huxley model is regarded as one of the great achievements of twentieth-century biophysics.

At its core, the Hodgkin–Huxley model framed how the action potential in neurons behaves like some analogous electric circuit. This formulation was based upon their observations of the dynamics of ion conductances responsible for generating the nerve action potential. In physiology, an action potential is a short-lasting event in which the electrical membrane potential of a cell rapidly rises and falls, following a consistent, repeatable trajectory. Action potentials occur in several types of animal cells, called excitable cells, which include neurons, muscle cells, and endocrine cells, as well as in some plant cells.

Driving the dynamics was the fact that there was a potential difference in voltage between the outside and inside of a given cell. A typical neuron has a resting potential of -70 mV. The cell membrane separating these two regions consists of a lipid bilayer. The cell membrane also has channels, either gated or nongated, across which ions can migrate from one region to the other. Nongated channels are always open, whereas gated channels can open and close with the probability of opening often depending upon the membrane potential. Such gated channels, which often select for a single ion, are called *voltage-gated channels*.

The Hodgkin–Huxley model builds a dynamic model of the action potential based upon the ion flow across the membrane. The principal ions involved in the dynamics are sodium ions (Na^+), potassium ions (K^+), and chloride anions (Cl^-). Although the Cl^- does not flow across the membrane, it does play a fundamental role in the charge balance and voltage dynamics of the sodium and potassium. In animal cells, two types of action potential exist: a voltage-gated sodium channel and a voltage-gated calcium channel. Sodium-based action potentials usually last for under one millisecond, whereas calcium-based action potentials may last for 100 milliseconds or longer. In some types of neurons, slow calcium spikes provide the driving force for a long burst of rapidly emitted sodium spikes. In cardiac muscle cells, on the other hand, an initial fast sodium spike provides a primer to provoke the rapid onset of a calcium spike, which then produces muscle contraction.

Accounting for the channel gating variables along with the membrane potential results in the 4×4 systems of nonlinear differential equations originally derived by Hodgkin and Huxley [**42, 43**]:

$$C_M\frac{dV}{dt} = -\bar{g}_K N^4(V-V_{\rm K})-\bar{g}_{Na}M^3H(V-V_{\rm Na})-\bar{g}_L(V-V_{\rm L})+I_0 \tag{20a}$$

$$\frac{dM}{dt} = \alpha_M(1-M)-\beta_M M \tag{20b}$$

$$\frac{dN}{dt} = \alpha_N(1-N)-\beta_N N \tag{20c}$$

$$\frac{dH}{dt} = \alpha_H(1-H)-\beta_H H \tag{20d}$$

where $V(t)$ is the voltage of the action potential that is driven by the voltage-gated variables for the potassium ($N(t)$) and the sodium ($M(t)$ and $H(t)$). The specific functions in the equations

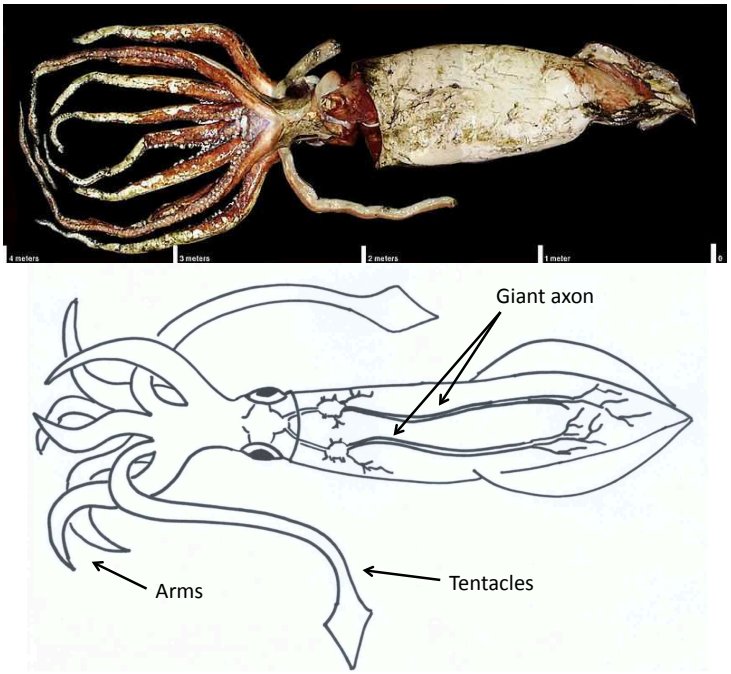

FIGURE 17. Picture and diagram of a giant squid with the giant axon denoted along the interior of the body. The giant axon was ideal for experiments given its length and girth. The top photograph is courtesy of NASA.

were derived to fit experimental data for the squid axon and are given by

$$\alpha_M = 0.1\frac{25-V}{\exp((25-V)/10)-1} \tag{21a}$$

$$\beta_M = 4\exp\left(\frac{-V}{18}\right) \tag{21b}$$

$$\alpha_H = 0.07\exp\left(\frac{-V}{20}\right) \tag{21c}$$

$$\beta_H = \frac{1}{\exp((30-V)/10)+1} \tag{21d}$$

$$\alpha_N = 0.01\frac{10-V}{\exp((10-V)/10)-1} \tag{21e}$$

$$\beta_N = 0.125\exp\left(\frac{-V}{80}\right) \tag{21f}$$

with the parameters in the first equation given by $\bar{g}_{Na} = 120$, $\bar{g}_K = 36$, $\bar{g}_L = 0.3$, $V_{Na} = 115$, $V_K = -12$ and $V_L = 10.6$. Here I_0 is some externally applied current to the cell. One of the characteristic behaviors observed in simulating this equation is the production of spiking behavior in the action potential which agrees with experimental observations in the giant squid axon.

FitzHugh–Nagumo reduction: The Hodgkin–Huxley model, it can be argued, marks the start of the field of computational neuroscience. Once it was established that neuron dynamics had sound theoretical underpinnings, the field quickly expanding and continues to be a dynamic and exciting

field of study today, especially as it pertains to characterizing the interactions of large groups of neurons responsible for decision making, functionality and data/signal-processing in sensory systems.

The basic Hodgkin–Huxley model already hinted at a number of potential simplifications that could be taken advantage of. In particular, there was clear separation of some of the time-scales involved in the voltage-gated ion flow. In particular, sodium-gated channels typically are very fast in comparison to calcium-gated channels. Thus on the *fast* scale, calcium-gating looks almost constant. Such arguments can be used to derive simpler versions of the Hodgkin–Huxley model. Perhaps the most famous of these is the FitzHugh–Nagumo model which is a two-component system that can be written in the form [**42, 43**]:

$$\frac{dV}{dt} = V(a - V)(V - 1) - W + I_0 \tag{22a}$$

$$\frac{dW}{dt} = bV - cW \tag{22b}$$

where $V(t)$ is the voltage dynamics and $W(t)$ models the refractory period of the neuron. The parameter a satisfies $0 < a < 1$ and determines much of the resulting dynamics. One of the most attractive features of this model is that standard phase-plane analysis techniques can be applied and a geometrical interpretation can be applied to the underlying dynamics. Although only qualitative in nature, the FitzHugh–Nagumo model has allowed for significant advancements in our qualitative understanding of neuron dynamics.

FitzHugh–Nagumo and propagation: Finally, we consider propagation effects in the axon itself. Up to this point, no propagation effects have been considered. One typical way to consider the effects of diffusion is to think of the voltage $V(t)$ as diffusing to neighboring regions of space and potentially causing these neighboring regions to become excitable and spike. The simplest nonlinear propagation model is given by the FitzHugh–Nagumo model with diffusion. This is a modification of (22) to include diffusion [**42, 43**]:

$$\frac{dV}{dt} = D\frac{d^2V}{dx^2} + V(a - V)(V - 1) - W + I_0 \tag{23a}$$

$$\frac{dW}{dt} = bV - cW \tag{23b}$$

where the parameter D measures the strength of the diffusion. For this case:

(a) Calculate the time dynamics of the Hodgkin–Huxley model for the default parameters given by the model.

(b) Apply an outside current $I_0(t)$, specifically a step function of finite duration, to see how this drives spiking behavior. Apply the step function periodically to see periodic spiking of the axon.

(c) Repeat (a) and (b) for the FitzHugh–Nagumo model.

(d) Although you can compute the fixed points and their stability for the FitzHugh–Nagumo model by hand, verify the analytic calculations for the location of the fixed points and determine their stability computationally.

(e) For fixed values of parameters in the FitzHugh–Nagumo model, perform a convergence study of the time-stepping method by controlling the error tolerance in the ODE solver. Show that indeed the schemes are fourth order and second order, respectively, by running the computation across a given time domain and adjusting the tolerance. In particular, plot on a log-log scale the average step-size (x-axis) using the *diff* and *mean* command versus the tolerance (y-axis) for a large number of tolerance values. What are the slopes of these lines? Note that the local error should be $O(\Delta t^5)$ and $O(\Delta t^3)$, respectively. What are the local errors for ODE113 and ODE15s?

FIGURE 18. Giants of celestial mechanics: (left) Nicolaus Copernicus (1473-1543, artist unknown) proposed a heliocentric model to replace the dominant Ptolemic viewpoint. (second from left) Galileo Galilei (1564-1642, portrait by Giusto Sustermans) invented the first telescope and was able to strongly argue for the Copernium model. Additionally, he studied and documented the first observations concerning uniformly accelerated bodies. (second from right) Johannes Kepler (1571-1630, artist unknown) made observations leading to three key findings about planetary orbits, including their elliptical trajectories. (right) Isaac Newton (1642-1727, portrait by Sir Godfrey Kneller), perhaps the most celebrated physicist in history, placed the work of Copernicus, Galileo and Kepler on a mathematical footing with the invention of calculus and the notion of a force called gravity.

(f) Assume a Gaussian initial pulse for the spatial version of the FitzHugh–Nagumo equation (23) and with $D = 0.1$ and investigate the dynamics for differing values of a. Assume periodic boundary conditions and solve using second-order finite differencing for the derivative. Explore the parameter space so as to produce spiking behavior and relaxation oscillations.

(g) Repeat (f) with implementation of the fast Fourier transform for computing the spatial derivatives. Compare methods (f) and (g) in terms of both computational time and convergence.

(h) Try to construct initial conditions for the FitzHugh–Nagumo equation (23) with $D = 0.1$ so that only a right traveling wave is created. (Hint: Make use of the refractory field which can be used to suppress propagation.)

(i) Make a movie of the spike wave dynamics and their propagation down the axon.

(j) Use these numerical models as a control algorithm for the construction of a giant squid made from spare parts in your garage. Attempt to pick up some of the smaller dogs in your neighborhood (see Fig. 17).

10.2. Celestial Mechanics and the Three-Body Problem. The movement of the heavenly bodies has occupied a central and enduring place in the history of science. From the earliest theoretical framework of the *Doctrine of the Perfect Circle* in the second century A.D. by Claudius Ptolemy to our modern understanding of celestial mechanics, gravitation has taken a pivotal role in nearly two thousand years of scientific and mathematical development. Indeed, some of the most prominent scientist in history have been involved in the development of the theoretical underpinnings of celestial mechanics and the notion of gravity, including Copernicus, Galileo, Kepler and Newton (Fig. 18). The two contemporaries, Galileo and Kepler, produced conjectures and observations that would lay the foundation for Newton's law of universal gravitation. Galileo built on the ideas of Copernicus and forcefully asserted that the Earth revolved around the Sun and further challenged the notion of the perfection of the heavenly spheres. Of course, his recalcitrant attitude towards the church did not help his career very much.

Kepler, a contemporary of Galileo's, further advanced the notion of planetary motion by asserting that (i) planets have elliptical orbits with the Sun as a focus, (ii) a radius vector from the Sun to a planet sweeps out equal areas in equal times, and (iii) the period of an orbit squared is proportional to the semi-major axis of the ellipse cubed. Such conjectures where able to improve

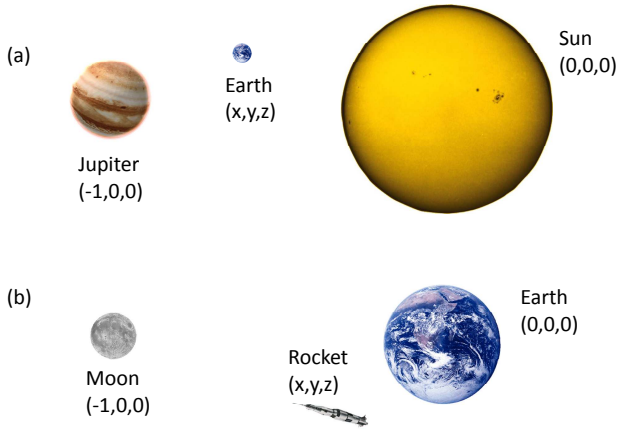

FIGURE 19. Two classic problems of the three-body problem: (a) the Sun-Jupiter-Earth system and (b) the Earth-Moon-Rocket/Satellite problem.

the accuracy of predicting planetary motion by two orders of magnitude over both the Ptolemic and Copernican systems.

The law of universal gravitation was proposed on the heels of the seminal work of Galileo and Kepler. In order to propose his theory of forces, Sir Isaac Newton formulated a new mathematical framework: calculus. This led to the much celebrated law of motion $\mathbf{F} = m\mathbf{a}$ where $\mathbf{F}$ is a given force vector and $\mathbf{a}$ is the acceleration vector, which is the second derivative of position. In the context of gravitation, Newton proposed the attractive gravitational law

$$F = \frac{GMm}{r^2} \tag{24}$$

where G is a universal constant, M and m are the masses of the two interacting bodies and r is the distance between them. With this law of attraction, the so-called Kepler's law could be written

$$\frac{d^2\mathbf{r}}{dt^2} = -\frac{\mu}{r^3}\mathbf{r} \tag{25}$$

where $\mu = G(m + M)$ and the dynamics of the smaller mass m is written in terms of a frame of reference with the larger mass M fixed at the origin. Thus $\mathbf{r}$ measures the distance from mass M to mass m. Equation (25) can be solved exactly to first, mathematically confirm the three assertions and/or observations made by Kepler, and second, to predict the dynamics of any two-mass system. This was known as the *two-body* problem and its solutions where conic sections in polar form, yielding straight lines, parabolas, ellipses and hyperbolas as solution orbits. Surprisingly enough, after more than 300 years of gravitation, this is the only gravitation problem for which we have a closed form solution. There are reasons for this as will be illustrated shortly.

The three-body problem: One can imagine the scientific excitement that came from Newton's theory of gravitation. No doubt, after the tremendous success of the two-body problem, there must have been some anticipation that solving a problem involving three masses would be more difficult, but perhaps a tenable proposition. As history has shown, the three-body problem has defied our ability to provide a closed form solution aside from some perturbative solutions in restricted asymptotic regimes.

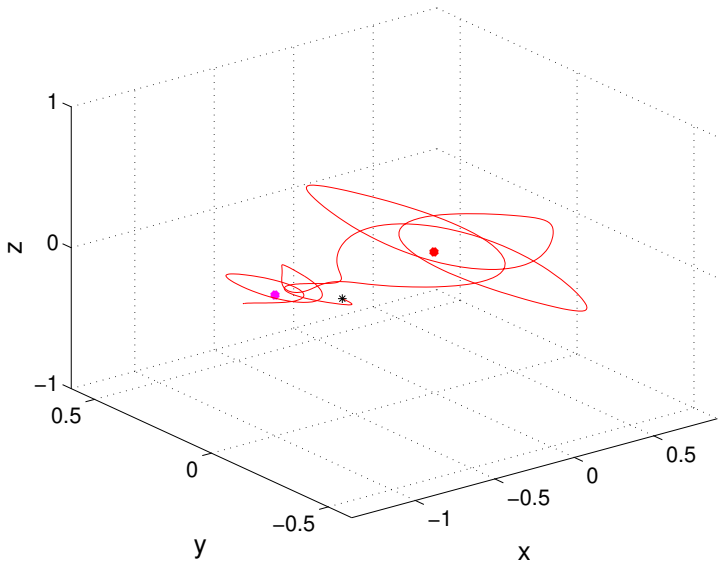

FIGURE 20. Trajectory of a satellite in a three-body system. In this case, the satellite performs an orbit transfer from m_1 to m_2 and back again. For this example, the following parameters where used $\mu = 0.1$ with initial conditions $(x, u, y, v, z, w) = (-1.2, 0, 0, 0, 0, 0)$. The evolution was for $t \in [0, 12]$. The red dot is mass m_1, the magenta dot if mass m_2, the red line is the trajectory following the mass m_3 denoted by a black star.

We consider the restricted three-body problem for which two of the masses m_1 and m_2 are much larger than m_3 so that $m_1, m_2 \gg m_3$ (see Fig. 19). Further, it is assumed that we are in a frame of reference such that the two masses m_1 and m_2 are located at the points $(0, 0)$ and $(-1, 0)$, respectively. Thus we consider the motion of m_3 under the gravitational pull of m_1 and m_2. This could, for instance, model an Earth–Moon–rocket system.

The governing equations are

$$
\begin{aligned}
x' &= u \\
u' &= 2v + x - \frac{\mu(-1+x+\mu)}{(y^2+z^2+(-1+x+\mu)^2)^{3/2}} - \frac{(1-\mu)(x+\mu)}{(y^2+z^2+(x+\mu)^2)^{3/2}} \\
y' &= v \\
v' &= -2u + y - \frac{\mu y}{(y^2+z^2+(-1+x+\mu)^2)^{3/2}} - \frac{(1-\mu)y}{(y^2+z^2+(x+\mu)^2)^{3/2}} \\
z' &= w \\
w' &= -\frac{\mu z}{(y^2+z^2+(-1+x+\mu)^2)^{3/2}} - \frac{(1-\mu)z}{(y^2+z^2+(x+\mu)^2)^{3/2}}
\end{aligned}
$$

where the position of mass m_3 is given by $\vec{r} = (x\ y\ z)$ and its velocity is $\vec{v} = (u\ v\ w)$. The parameter $0 < \mu < 1$ measures the relative masses of the smaller to large mass, i.e. m_2/m_1. An example trajectory for the three-body problem is shown in Fig. 20.

We can further restrict the problem so that m_3 is in the same plane (x–y plane) as masses m_1 and m_2. Thus we throw out the z and w dynamics and only consider x, y, u and v. For this case:

(a) Using a root solving routing (and analytic methods), find the five critical (Lagrange) points.

(b) Find the stability of the critical point by linearizing around each point. MATLAB can be used to help find the eigenvalues.
(c) Evolve the governing nonlinear ODEs and verify the stability calculations of (b) above.
(d) Show that the system is chaotic by demonstrating sensitivity to initial conditions, i.e. measure the separation in time of two nearly identical initial conditions.
(e) By exploring with MATLAB, show the following possible behaviors:
  (1) Stable evolution around m_1
  (2) Stable evolution around m_2
  (3) Transfer of orbit from m_1 to m_2
  (4) Periodic and stable evolution near the two Lagrange points where $y \neq 0$
  (5) Escape from the m_1, m_2 system.
(f) Explore the change in behavior as a function of the parameter μ, i.e. as the dominance, or lack thereof, of one of the masses changes.

10.3. Atmospheric Motion and the Lorenz Equations. Understanding climate patterns and providing accurate weather predictions are of central importance in the fields of atmospheric sciences and oceanography. And despite decades of study and large meteorological models, it would seem that weather forecasts are still quite inaccurate, and mostly untrustworthy for more than one week into the future. Ultimately, there is a fundamental mathematical reason for this. Much like the three-body problem of celestial mechanics, even a relatively simple idealized model can demonstrate the source of the underlying difficulties in predicting weather.

The first scientist to predict that weather and climate dynamics where indeed a *global* phenomenon was Sir Gilbert Walker. In 1904 at the age of 36, he was posted in India as the Director General of Observations. Of pressing interest to him was the ability to predict monsoon failures and the ensuing droughts that would accompany them as this had a profound impact on the people of India. In an attempt to understand these seemingly random failures, he looked for correlations in weather and climate across the globe and found that the random failure of monsoons in India often coincided with low pressure over Tahiti, high pressure over Darwin, Australia, and relaxed trade winds over the Pacific Ocean. Indeed, his observations showed that pressure highs in Tahiti were directly correlated to pressure lows in Darwin and vice versa. This is now called the *southern oscillation*. In years when the southern oscillation was weak, the following observations were made: there was heavy rainfall in the central Pacific, droughts in India, warm waters in south-west Canada and cold winters in south-east United States. This led to his conjecture that weather was indeed a global phenomenon. He was largely ignored for decades, and even ridiculed for his suggestion that climatic conditions over such widely separated regions of the globe could be linked.

Sixty-five years after Walker's claims, Jacob Bjerknes in 1969 advanced a conceptual mathematical model tying together the well documented El Niño phenomenon to the southern oscillation: the El Niño southern oscillation (ENSO). A schematic of the model is illustrated in Fig. 21 where the normal Walker circulation can be broken if the easterly trade winds slacken. In this case, warm water from the central Pacific is pushed back to the eastern Pacific leading to both the El Niño phenomenon and the see-sawing of high pressures between Darwin, Australia and Tahiti.

More than a schematic as shown in Fig. 21, the theory proposed by Bjerknes was a mathematical description of the circulation and its potential breakdown. The model makes a variety of simplifying assumptions. First on the list of assumptions is that the velocity of the surface wind u_a is driven by the temperature difference $T_e - T_w$, i.e. hot air moves to cold regions. In differential form, this can be stated mathematically as

$$\frac{du_a}{dt} = b(T_e - T_w) + r(u_0 - u_a) \tag{26}$$

where u_0 measures the velocity of the normal easterly winds, r is the rate at which the u_a relaxes to u_0, and b is the rate at which the ocean influences the atmosphere. But since the ocean temperature

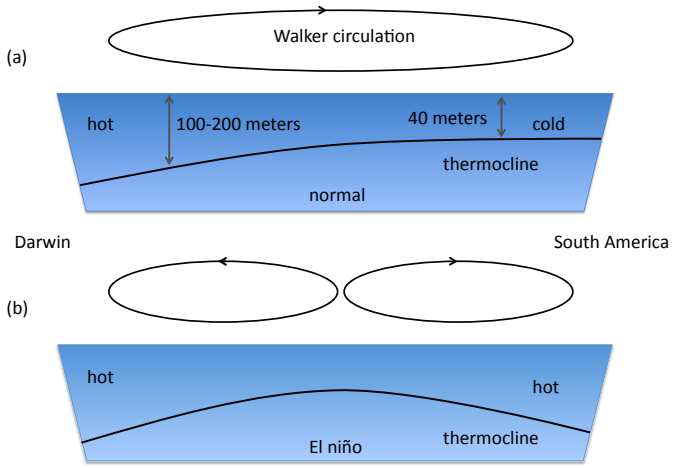

FIGURE 21. Schematic of the southern oscillation phenomenon and El niño. The normal Walker circulation pattern is illustrated in (a) where a large circulation occurs from the South American continent to Australia due to the easterly trade winds. If the trade winds slacken, warm water moves back east and warms the easter Pacific resulting in El niño. Note that the breakage of the normal Walker circulation pattern leads to a see-saw in pressure highs between Darwin and Tahiti

and velocity change slowly in relation to the atmospheric temperature and velocity, the following approximation holds: $du_a/dt \approx 0$. Thus

$$u_a = \frac{b}{r}(T_e - T_w) + u_0 \tag{27}$$

gives an approximation to the air velocity at the surface.

The air velocity interacts with the ocean through a stress coupling at the water–air interface. This stress coupling is responsible for generating the ocean circulation velocity near the surface of $u = U$. The simplest model for this coupling is given by

$$\frac{dU}{dt} = du_a - cU \tag{28}$$

where d measures the strength of the coupling and c is the damping (relaxation) back to the zero velocity state $U = 0$. Note that if the coupling is $d = 0$, then the solution is given by $U = U(0)\exp(-ct)$ so that $U \to 0$ as $t \to 0$. Thus u_a is responsible for driving the current. Substituting the value of u_a previously found into this expression yields the velocity dynamics

$$\frac{dU}{dt} = B(T_e - T_w) - C(U - U_0) \tag{29}$$

where $B = db/r$ and $U_0 = du_0/c$. This is the approximate governing equation for the air–sea interaction dynamics.

Ocean temperature advection: The equation governing the evolution of the temperature is a standard advection equation which is not derived here. The spatial advection of temperature is given by the partial differential equation

$$\frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} + w\frac{\partial T}{\partial z} \tag{30}$$

where $T = T(x, z, t)$ gives the temperature distribution as a function of time and the horizontal and vertical location. The temperature partial differential equation can be approximated using a

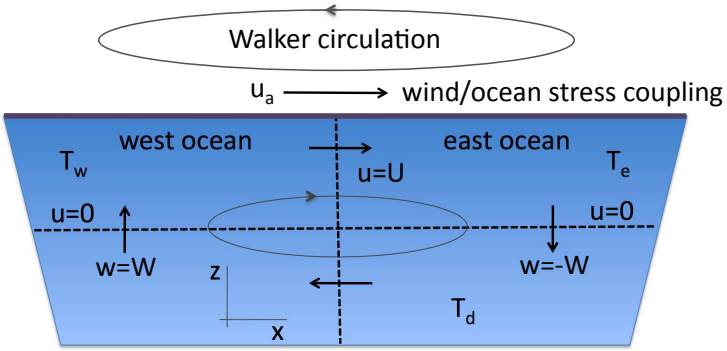

FIGURE 22. Mathematical construction of the southern oscillation scenario. The parameter u_a measures the wind speed from the Walker circulation that interacts via sheer stress with the ocean in the purple top year. The resulting ocean velocity in the top layer is assume to be $u = U$. The temperature in the west ocean, east ocean and deep ocean is given by T_w, T_e and T_d respectively. These temperature gradients drive the circulation dynamics.

finite difference scheme for the spatial derivatives. Thus we take

$$\frac{\partial T}{\partial x} \approx \frac{T(x+\Delta x) - T(x-\Delta x)}{2\Delta x}$$
$$\frac{\partial T}{\partial z} \approx \frac{T(z+\Delta z) - T(z-\Delta z)}{2\Delta z}$$

where Δz, $\Delta x \ll 1$ in order for the approximation to be somewhat accurate. The key now is to divide the ocean into four (east and west and top/bottom) finite difference boxes as illustrated in Fig. 22. In particular, there is assumed to be an east ocean box and a west ocean box. In the east ocean box, the following approximate relations hold:

$$u\frac{\partial T}{\partial x} \to u\frac{T(x+\Delta x) - T(x-\Delta x)}{2\Delta x} = \frac{U}{2\Delta x}(T_e - T_w)$$
$$w\frac{\partial T}{\partial z} \to w\frac{T(z+\Delta z) - T(z-\Delta z)}{2\Delta z} = -\frac{W}{2\Delta z}(T_e - T_d)\,.$$

Inserting these approximate expressions into the original temperature advection equations yields the temperature equation

$$\frac{\partial T_e}{\partial t} + \frac{U}{2\Delta x}(T_e - T_w) - \frac{W}{2\Delta z}(T_e - T_d) = 0\,. \tag{31}$$

Applying the same approximation in the west ocean box yields a similar formula aside from a change in sign for $w = W$. This yields the west ocean box equation

$$\frac{\partial T_w}{\partial t} + \frac{U}{2\Delta x}(T_e - T_w) + \frac{W}{2\Delta z}(T_w - T_d) = 0\,. \tag{32}$$

Thus a significant approximation occurred by assuming that the entire Pacific Ocean could be divided into two finite difference cells.

10.4. The Lorenz equations. Finally, in order to enforce mass conservation, we must require $W\Delta x = U\Delta z$ so that $W = U\Delta z/\Delta x$. Inserting this relation into the formulas above for the east and west ocean temperatures and considering in addition the air–sea interaction dynamics of (29) gives the governing

ENSO dynamics

$$x' = \sigma y - \rho(x - x_0) \tag{33a}$$
$$y' = x - xz - y \tag{33b}$$
$$z' = xy - z \tag{33c}$$

where the prime denotes time differentiation and the following rescalings are used: $x = U/(A2\Delta x)$, $y = (T_e - T_w)/(2T_0)$, $z = 1 - (T_e + T_w)/(2T_0)$, $t \to At$, $x_0 = U_0/(A2\Delta x)$, $\sigma = 2BT_0/(\Delta x A^2)$ and $\rho = c/A$. A final rescaling with $x_0 = 0$ can move the ENSO dynamics into the more commonly accepted form

$$x' = -\sigma x + \sigma y \tag{34a}$$
$$y' = rx - xz - y \tag{34b}$$
$$z' = xy - bz. \tag{34c}$$

These are the famed *Lorenz equations* that first arose in the study of the ENSO phenomenon at the end of the 1960s. Reasonable values of parameters for Earth's atmosphere give $\sigma = 8/3$ and $\sigma = 10$. The objective is then to see how the solutions change with the parameter r which relates the temperate difference in the layers of the atmosphere.

(a) Using a root solving routine and/or analytic methods, find the critical value of the parameter r where the number of critical points changes from one to three.
(b) Find the stability of the critical point by linearizing around each point. MATLAB can be used to help find the eigenvalues.
(c) Evolve the governing nonlinear ODEs and verify the stability calculations of (b) above.
(d) Determine the critical value $r = r_c$ numerically at which the system becomes chaotic by demonstrating sensitivity to initial conditions, i.e. measure the separation in time of two nearly identical initial conditions. Show that for $r < r_c$, the system is not chaotic while for $r > r_c$ the system is chaotic.
(e) By exploring with MATLAB, also consider the periodically forced Lorenz equations as a function of γ and ω:

$$x' = \sigma y - \rho(x - x_0) + \gamma\cos(\omega t) \tag{35a}$$
$$y' = x - xz - y \tag{35b}$$
$$z' = xy - z. \tag{35c}$$

Use the PLOT3 command to observe the dynamics parametrically in time as a function of $x(t), y(t)$ and $z(t)$.

10.5. Quantum Mechanics. One of the most celebrated physics developments of the twentieth century was the theoretical development of quantum mechanics. The genesis of quantum mechanics, and a quantum description of natural phenomena, began with Max Planck and his postulation that energy is radiated and absorbed in discrete quanta (or energy elements), thus providing a theoretical framework for understanding experimental observations of blackbody radiation. In his theory, each quantum of energy was proportional to its frequency (ν) so that

$$E = h\nu \tag{36}$$

where h is Planck's constant. This is known as Planck's law. Einstein used his hypothesis to explain the photoelectric effect, for which he won his Nobel Prize.

Shortly after the work of Planck and Einstein, the foundations of quantum mechanics were laid down in Europe by some of the most prominent physicists of the twentieth century. Figure 23 is a picture from the 1927 conference on quantum mechanics in Brussels, Belgium. It would be difficult to find a conference or workshop with a higher ratio of Nobel Prize winners than this conference.

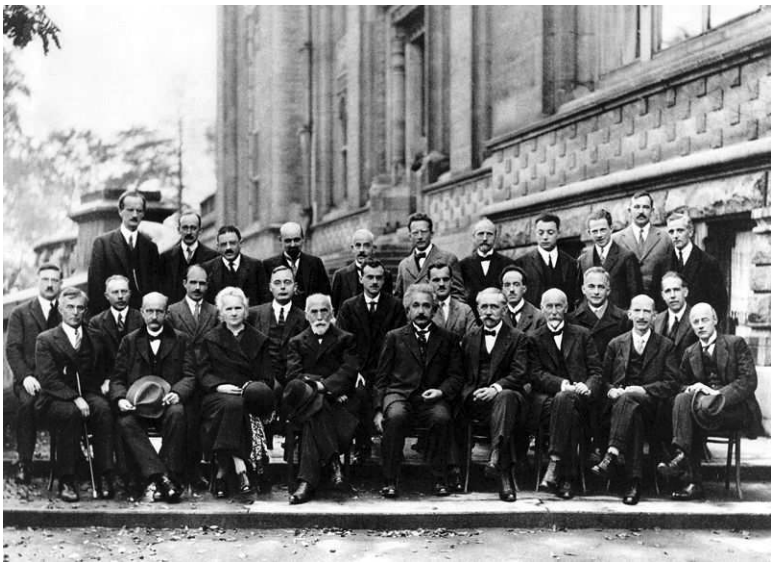

FIGURE 23. The Quantum Mechanics Dream Team at the 1927 Solvay Conference on Quantum Mechanics. The photograph is by Benjamin Couprie, Institut International de Physique Solvay, Brussels, Belgium. Front Row: Irving Langmuir, Max Planck, Marie Curie, Hendrik Lorentz, Albert Einstein, Paul Langevin, Charles Guye, Charles Thomson, Rees Wilson, Owen Richardson. Middle Row: Peter Debye, Martin Knudsen, William Bragg, Hendrik Kramers, Paul Dirac, Arthur Compton, Louis de Broglie, Max Born, Niels Bohr. Back Row: Auguste Piccard, Émile Henriot, Paul Ehrenfest, Édouard Herzen, Théophile de Donder, Erwin Schrödinger, Jules-Émile Verschaffelt, Wolfgang Pauli, Werner Heisenberg, Ralph Howard Fowler, Léon Brillouin. Thanks to this group, you have that sweet iPhone.

And only a few years later, the scientific community of Europe would begin to fracture under the rise of Hitler and the imminence of World War II.

Quantum mechanics has had enormous success in explaining, among other things, the individual behavior of the subatomic particles that make up all forms of matter (electrons, protons, neutrons, photons and others), how individual atoms combine covalently to form molecules, the basic operation of lasers, transistors and electron microscopes, etc. In semiconductors, the working mechanism for resonant tunneling in a diode device, based on the phenomenon of quantum tunneling through potential barriers, has led to the entire industry of quantum-electronically based devices, such as the iPhone and your laptop. Ultimately, quantum mechanics has gone from a theoretical construct with many interesting philosophical implications to the greatest engineering design and manufacturing tool of the modern, high-tech world.

In what we develop here, we will follow Schrödinger's formulation of the problem. De Broglie had already introduced the idea of wave-like behavior to atomic particles. However, the standard wave equation requires two initial conditions in order to correctly pose the initial value problem. But this violated *Heisenberg's uncertainty principle* which stated that the momentum and position of a particle could not be known simultaneously. Indeed, if the exact position was known, then no information was known about its momentum. On the other hand, if the exact momentum was known, then nothing could be known about its position. This led Schrödinger to formulate the following wave equation

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial t^2} \tag{37}$$

where $\hbar = h/2\pi$, m is the particle mass, and $\psi(x,t)$ is the wavefunction that represents the probability function for finding the particle at a location x at a time t. As a consequence of this probabilistic interpretation, $\int |\psi|^2 dx = 1$. This equation is known as Schrödinger's equation. It is a wave equation that requires only a single initial condition, thus it retains wave-like behavior without violating the Heinsenberg uncertainty principle.

In most cases, the particle is also subject to an external potential field. Thus the probability density evolution in a one-dimensional trapping potential, in this case assumed to be the harmonic potential, is governed by the partial differential equation:

$$i\hbar\psi_t + \frac{\hbar^2}{2m}\psi_{xx} + V(x)\psi = 0\,, \tag{38}$$

where ψ is the probability density and $V(x) = kx^2/2$ is the harmonic confining potential. A typical solution technique for this problem is to assume a solution of the form

$$\psi = \sum_1^N a_n \phi_n(x) \exp\left(i\frac{E_n}{2\hbar}t\right) \tag{39}$$

which is called an eigenfunction expansion solution ($\phi_n =$ eigenfunction, $E_n =$ eigenvalue). Plugging in this solution ansatz to Eq. (38) gives the boundary value problem:

$$\frac{d^2\phi_n}{dx^2} - \left[Kx^2 - \varepsilon_n\right]\phi_n = 0 \tag{40}$$

where we expect the solution $\phi_n(x) \to 0$ as $x \to \pm\infty$ and E_n is the quantum energy. Note here that $K = km/\hbar^2$ and $\varepsilon = Em/\hbar^2$. In what follows, take $K = 1$ and always normalize so that $\int_\infty^\infty |\phi_n|^2 dx = 1$.

(a) Calculate the first five *normalized* eigenfunctions (ϕ_n) and eigenvalues (ε_n) using a shooting scheme.

(b) Calculate the first five *normalized* eigenfunctions (ϕ_n) and eigenvalues (ε_n) using a direct method. Be sure to use forward- and backward-differencing for the boundary conditions. (Hint: $3 + \Delta x\sqrt{KL^2 - E} \approx 3$.)

(c) There has been suggestions that in some cases, nonlinearity plays a role such that

$$\frac{d^2\phi_n}{dx^2} - \left[\gamma|\phi|^2 Kx^2 - \varepsilon_n\right]\phi_n = 0\,. \tag{41}$$

Depending upon the sign of γ, the probability density is focused or defocused. Find the first three *normalized* modes for $\gamma = \pm 0.2$ using shooting.

(d) For a fixed value of the energy (take, for instance, ε_1), perform a convergence study of the shooting method by controlling the error tolerance in the ODE solver. Show that indeed the schemes are fourth order and second order, respectively, by running the computation accross the computational domain and adjusting the tolerance. In particular, plot on a log-log scale the average step-size (x-axis) using the *diff* and *mean* command versus the tolerance (y-axis) for a large number of tolerance values. What are the slopes of these lines? Note that the local error should be $O(\Delta t^5)$ and $O(\Delta t^3)$, respectively. What are the local errors for ODE113 and ODE15s?

(e) Compare your solutions with the exact Gauss–Hermite polynomial solutions for this problem.

(f) For a double-well potential

$$V(x) = \begin{cases} -1 & 1 < x < 2 \text{ and } -2 < x < -1 \\ 0 & \text{otherwise,} \end{cases} \tag{42}$$

calculate the symmetric ground state ϕ_1 and the first, antisymmetric state ϕ_2. Note that $|\varepsilon_1 - \varepsilon_2| \ll 1$, thus the difficulty in this problem. Make a 3D plot $(x, t, |\psi|)$ of the solution

(39) for the initial condition $\psi = \phi_1 + \phi_2$ which shows the tunneling between potential wells. Note that for plotting purposes, take $\hbar = 1$ and $m = 1$.

10.6. Electromagnetic Waveguides. The propagation of electromagnetic energy in a one-dimensional optical waveguide is governed by the partial differential equation:

$$2ikU_z + U_{xx} + k^2\Delta^2 n(x)U = 0\,, \tag{43}$$

where U is the envelope of the electromagnetic field and the equation has been nondimensionalized such that the unit length is the waveguide core radius $a = 10\mu$m. Here, the dimensionless wavenumber is $k = 2\pi n_0 a/\lambda_0$ where $n_0 = 1.46$ is the cladding index of refraction and $\lambda_0 = 1.55\mu$m is the free-space wavelength. The parameter $\Delta^2 = (n_{\text{core}}^2 - n_0^2)/n_0^2$ measures the difference between the peak value of the index in the core $n_{\text{core}} = 1.48$ and the cladding n_0. Note that z measures the distance traveled in the waveguide and x is the transverse dimension.

A typical solution technique for this problem is to assume a solution of the form

$$U = \sum_1^N A_n \psi_n(x) \exp\left(\frac{i}{2k}\beta_n z\right) \tag{44}$$

which is called an eigenfunction expansion solution ($\psi_n =$ eigenfunction, $\beta_n =$ eigenvalue). Plugging in this solution ansatz to Eq. (43) gives the boundary value problem

$$\frac{d^2\psi_n}{dx^2} + \left[k^2\Delta^2 n(x) - \beta_n\right]\psi_n = 0 \tag{45}$$

where we expect the solution $\psi_n(x) \to 0$ as $x \to \pm\infty$ and β_n is called the propagation constant. The function $n(x)$ gives the index of refraction profile of the fiber and is typically manufactured to enhance the performance of a given application. For ideal profiles,

$$n(x) = \begin{cases} 1 - |x|^\alpha & 0 \le |x| \le 1 \\ 0 & |x| > 1. \end{cases} \tag{46}$$

Two cases of particular interest are for $\alpha = 2$ and $\alpha = 10$.

(a) Calculate the first five *normalized* eigenfunctions (ψ_n) and eigenvalues (β_n) for these two cases using a shooting scheme. (Note: normalization $\int_{-\infty}^{\infty} |\psi_n|^2 dx = 1$.)

(b) Calculate the first five *normalized* eigenfunctions (ψ_n) and eigenvalues (β_n) for these two cases using a direct solve scheme. (Note: normalization $\int_{\infty}^{\infty} |\psi_n|^2 dx = 1$.)

(c) For high-intensity pulses, the index of refraction depends upon the intensity of the pulse itself. The propagating modes are thus found from

$$\frac{d^2\psi}{dx^2} + \left[\gamma|\psi|^2 + k^2\Delta^2 n(x) - \beta\right]\psi = 0\,. \tag{47}$$

Depending upon the sign of γ, the waveguide leads to focusing or defocusing of the electromagnetic field. Find the first three *normalized* modes for $\gamma = \pm 0.2$ using shooting.

(d) For the case $\alpha = 2$ and for a fixed value of the propagation constant (take, for instance, β_1), perform a convergence study of the shooting method. Show that indeed the schemes are fourth order and second order, respectively, by running the computation accross the computational domain and adjusting the tolerance. In particular, plot on a log-log scale the average step-size (x-axis) using the *diff* and *mean* command versus the tolerance (y-axis) for a large number of tolerance values. What are the slopes of these lines? Note that the local error should be $O(\Delta t^5)$ and $O(\Delta t^3)$, respectively. What are the local errors for ODE113 and ODE15s?

CHAPTER 9

Finite Difference Methods

Finite difference methods are based exclusively on Taylor expansions. They are one of the most powerful methods available since they are relatively easy to implement, can handle fairly complicated boundary conditions, and allow for explicit calculations of the computational error. The result of discretizing any given problem is the need to solve a large linear system of equations or perhaps manipulate large, sparse matrices. All this will be dealt with in the following sections.

1. Finite Difference Discretization

To discuss the solution of a given problem with the finite difference method, we consider a specific example from atmospheric sciences which is detailed in the exercises at the end of this chapter. The quasi-two-dimensional motion of the atmosphere can be modeled by the advection–diffusion behavior for the vorticity $\omega(x, y, t)$ which is coupled to the streamfunction $\psi(x, y, t)$:

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega \tag{1a}$$

$$\nabla^2 \psi = \omega \tag{1b}$$

where

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} \tag{2}$$

and $\nabla^2 = \partial_x^2 + \partial_y^2$ is the two-dimensional Laplacian. Note that this equation has both an advection component (hyperbolic) from $[\psi, \omega]$ and a diffusion component (parabolic) from $\nu \nabla^2 \omega$. We will assume that we are given the initial value of the vorticity

$$\omega(x, y, t = 0) = \omega_0(x, y)\,. \tag{3}$$

Additionally, we will proceed to solve this problem with periodic boundary conditions. This gives the following set of boundary conditions

$$\omega(-L, y, t) = \omega(L, y, t) \tag{4a}$$
$$\omega(x, -L, t) = \omega(x, L, t) \tag{4b}$$
$$\psi(-L, y, t) = \psi(L, y, t) \tag{4c}$$
$$\psi(x, -L, t) = \psi(x, L, t) \tag{4d}$$

where we are solving on the computational domain $x \in [-L, L]$ and $y \in [-L, L]$.

Basic algorithm structure: Before discretizing the governing partial differential equation, it is important to clarify what the basic solution procedure will be. Two physical quantities need to be solved as functions of time:

$$\psi(x, y, t) \quad \text{streamfunction} \tag{5a}$$
$$\omega(x, y, t) \quad \text{vorticity.} \tag{5b}$$

We are given the initial vorticity $\omega_0(x, y)$ and periodic boundary conditions. The solution procedure is as follows:

(1) **Elliptic solve:** Solve the elliptic problem $\nabla^2\psi = \omega_0$ to find the streamfunction at time zero $\psi(x, y, t = 0) = \psi_0$.
(2) **Time-stepping:** Given initial ω_0 and ψ_0, solve the advection–diffusion problem by time-stepping with a given method. The Euler method is illustrated below

$$\omega(x, y, t + \Delta t) = \omega(x, y, t) + \Delta t \left(\nu\nabla^2\omega(x, y, t) - [\psi(x, y, t), \omega(x, y, t)]\right).$$

This advances the solution Δt into the future.
(3) **Loop:** With the updated value of $\omega(x, y, \Delta t)$, we can repeat the process by again solving for $\psi(x, y, \Delta t)$ and updating the vorticity once again.

This gives the basic algorithmic structure which must be implemented in order to generate the solution for the vorticity and streamfunction as functions of time. It only remains to discretize the problem and solve.

Step 1: Elliptic solve: We begin by discretizing the elliptic solve problem for the streamfunction $\psi(x, y, t)$. The governing equation in this case is

$$\nabla^2\psi = \frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} = \omega. \tag{6}$$

Using the central difference formulas of Section 6 reduces the governing equation to a set of linearly coupled equations. In particular, we find for a second-order accurate central difference scheme that the elliptic equation reduces to

$$\begin{aligned}\frac{\psi(x + \Delta x, y, t) - 2\psi(x, y, t) + \psi(x - \Delta x, y, t)}{\Delta x^2} \\ + \frac{\psi(x, y + \Delta y, t) - 2\psi(x, y, t) + \psi(x, y - \Delta y, t)}{\Delta y^2} = \omega(x, y, t).\end{aligned} \tag{7}$$

Thus the solution at each point depends upon itself and four neighboring points. This creates a five-point stencil for solving this equation. Figure 1 illustrates the stencil which arises from discretization. For convenience we denote

$$\psi_{mn} = \psi(x_m, y_n, t). \tag{8}$$

By letting $\Delta x^2 = \Delta y^2 = \delta^2$, the discretized equations reduce to

$$-4\psi_{mn} + \psi_{(m-1)n} + \psi_{(m+1)n} + \psi_{m(n-1)} + \psi_{m(n+1)} = \delta^2\omega_{mn} \tag{9}$$

with periodic boundary conditions imposing the following constraints

$$\psi_{1n} = \psi_{(N+1)n} \tag{10a}$$

$$\psi_{m1} = \psi_{m(N+1)} \tag{10b}$$

where $N + 1$ is the total number of discretization points in the computational domain in both the x- and y-directions.

As a simple example, consider the four-point system for which $N = 4$. For this case, we have the following sets of equations

$$\begin{aligned}-4\psi_{11} + \psi_{41} + \psi_{21} + \psi_{14} + \psi_{12} &= \delta^2\omega_{11} \\ -4\psi_{12} + \psi_{42} + \psi_{22} + \psi_{11} + \psi_{13} &= \delta^2\omega_{12} \\ &\vdots \\ -4\psi_{21} + \psi_{11} + \psi_{31} + \psi_{24} + \psi_{22} &= \delta^2\omega_{21} \\ &\vdots\end{aligned} \tag{11}$$

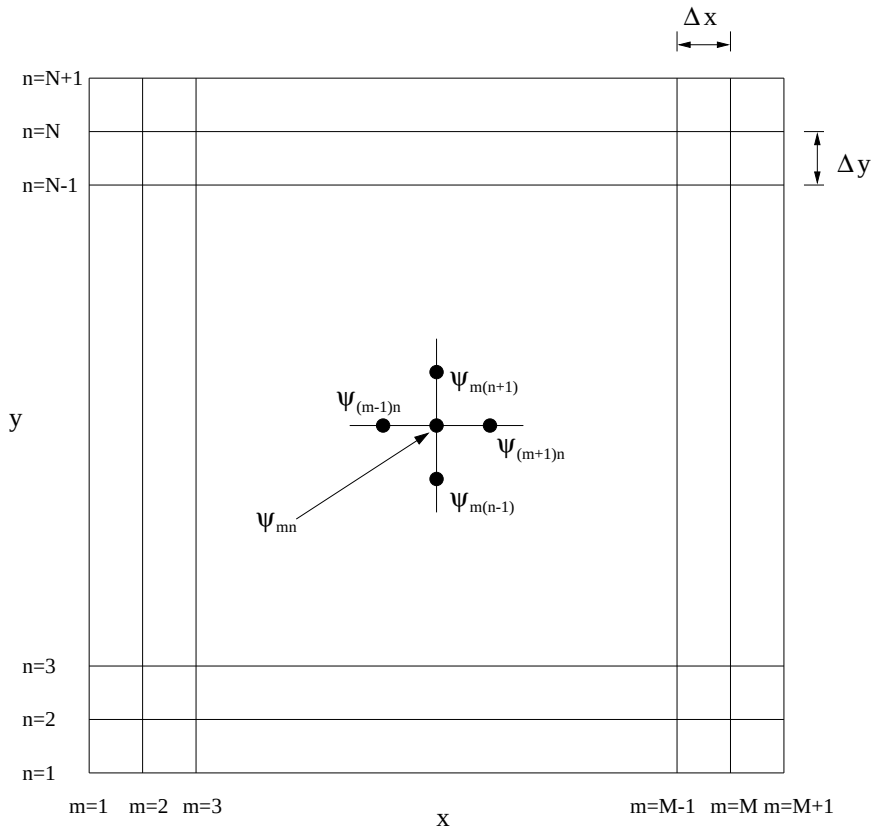

FIGURE 1. Discretization stencil for solving for the streamfunction with second-order accurate central difference schemes. Note that $\psi_{mn} = \psi(x_m, y_n)$.

which results in the sparse matrix (banded matrix) system

$$\mathbf{A}\psi = \delta^2\omega \tag{12}$$

where

$$\mathbf{A} = \begin{bmatrix}
-4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -4
\end{bmatrix} \tag{13}$$

and

$$\psi = (\psi_{11}\,\psi_{12}\,\psi_{13}\,\psi_{14}\,\psi_{21}\,\psi_{22}\,\psi_{23}\,\psi_{24}\,\psi_{31}\,\psi_{32}\,\psi_{33}\,\psi_{34}\,\psi_{41}\,\psi_{42}\,\psi_{43}\,\psi_{44})^{\mathrm{T}} \tag{14a}$$

$$\omega = (\omega_{11}\,\omega_{12}\,\omega_{13}\,\omega_{14}\,\omega_{21}\,\omega_{22}\,\omega_{23}\,\omega_{24}\,\omega_{31}\,\omega_{32}\,\omega_{33}\,\omega_{34}\,\omega_{41}\,\omega_{42}\,\omega_{43}\,\omega_{44})^{\mathrm{T}}\,. \tag{14b}$$

Any matrix solver can then be used to generate the values of the two-dimensional streamfunction which are contained completely in the vector ψ.

Step 2: Time-stepping: After generating the matrix **A** and the value of the streamfunction $\psi(x,y,t)$, we use this updated value along with the current value of the vorticity to take a time-step Δt into the future. The appropriate equation is the advection–diffusion evolution equation:

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega\,. \tag{15}$$

Using the definition of the bracketed term and the Laplacian, this equation is

$$\frac{\partial \omega}{\partial t} = \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} - \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} + \nu\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right)\,. \tag{16}$$

Second-order central-differencing discretization then yields

$$\begin{aligned}
\frac{\partial \omega}{\partial t} \;=\; & \left(\frac{\psi(x,y+\Delta y,t)-\psi(x,y-\Delta y,t)}{2\Delta y}\right)\left(\frac{\omega(x+\Delta x,y,t)-\omega(x-\Delta x,y,t)}{2\Delta x}\right) \\
& - \left(\frac{\psi(x+\Delta x,y,t)-\psi(x-\Delta x,y,t)}{2\Delta x}\right)\left(\frac{\omega(x,y+\Delta y,t)-\omega(x,y-\Delta y,t)}{2\Delta y}\right) \\
& + \nu\left\{\frac{\omega(x+\Delta x,y,t)-2\omega(x,y,t)+\omega(x-\Delta x,y,t)}{\Delta x^2}\right. \\
& \qquad \left. + \frac{\omega(x,y+\Delta y,t)-2\omega(x,y,t)+\omega(x,y-\Delta y,t)}{\Delta y^2}\right\}.
\end{aligned} \tag{17}$$

This is simply a large system of differential equations which can be stepped forward in time with any convenient time-stepping algorithm such as fourth-order Runge–Kutta. In particular, given that there are $N+1$ points and periodic boundary conditions, this reduces the system of differential equations to an $N \times N$ coupled system. Once we have updated the value of the vorticity, we must again update the value of the streamfunction to once again update the vorticity. This loop continues until the solution at the desired future time is achieved. Figure 2 illustrates how the five-point, two-dimensional stencil advances the solution.

The behavior of the vorticity is illustrated in Fig. 3 where the solution is advanced for eight time units. The initial condition used in this simulation is

$$\omega_0 = \omega(x,y,t=0) = \exp\left(-2x^2 - \frac{y^2}{20}\right)\,. \tag{18}$$

This stretched Gaussian is seen to rotate while advecting and diffusing vorticity. Multiple vortex solutions can also be considered along with oppositely signed vortices.

2. Advanced Iterative Solution Methods for Ax = b

In addition to the standard techniques of Gaussian elimination or LU decomposition for solving $\mathbf{Ax} = \mathbf{b}$, a wide range of iterative techniques are available.

Application to advection–diffusion: When discretizing many systems of interest, such as the advection–diffusion problem, we are left with a system of equations that is naturally geared toward iterative methods. Discretization of the stream/function previously yielded the system

$$-4\psi_{mn} + \psi_{(m+1)n} + \psi_{(m-1)n} + \psi_{m(n+1)} + \psi_{m(n-1)} = \delta^2 \omega_{mn}\,. \tag{1}$$

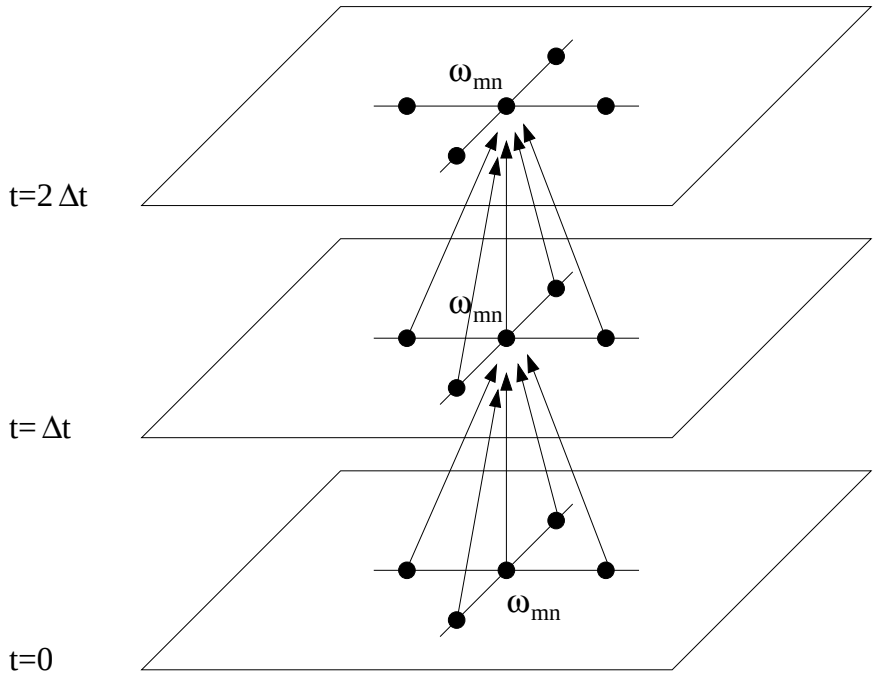

FIGURE 2. Discretization stencil resulting from center-differencing of the advection-diffusion equations. Note that for explicit stepping schemes the future solution only depends upon the present. Thus we are not required to solve a large linear system of equations.

The matrix **A** in this case is represented by the left-hand side of the equation. Letting ψ_{mn} be the diagonal term, the iteration procedure yields

$$\psi_{mn}^{k+1} = \frac{\psi_{(m+1)n}^{k} + \psi_{(m-1)n}^{k} + \psi_{m(n+1)}^{k} + \psi_{m(n-1)}^{k} - \delta^2 \omega_{mn}}{4}. \tag{2}$$

Note that the diagonal term has a coefficient of $|-4| = 4$ and the sum of the off-diagonal elements is $|1| + |1| + |1| + |1| = 4$. Thus the system is at the borderline of being diagonally dominant. So although convergence is not guaranteed, it is highly likely that we could get the Jacobi scheme to converge.

Finally, we consider the operation count associated with the iteration methods. This will allow us to compare this solution technique with Gaussian elimination and LU decomposition. The following basic algorithmic steps are involved:

(1) Update each ψ_{mn} which costs N operations times the number of nonzero diagonals D.
(2) For each ψ_{mn}, perform the appropriate additions and subtractions. In this case there are five operations.
(3) Iterate until the desired convergence which costs K operations.

Thus the total number of operations is $O(N \cdot D \cdot 5 \cdot K)$. If the number of iterations K can be kept small, then iteration provides a viable alternative to the direct solution techniques.

3. Fast Poisson Solvers: The Fourier Transform

Other techniques exist for solving many computational problems which are not based upon the standard Taylor series discretization. For instance, we have considered solving the streamfunction equation

$$\nabla^2 \psi = \omega \tag{1}$$

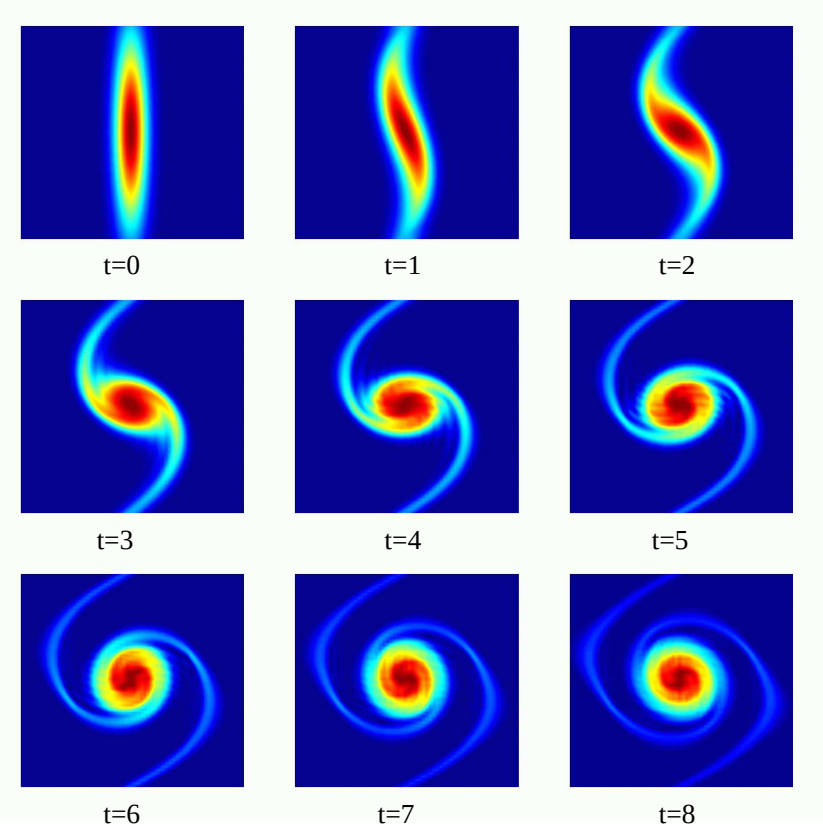

FIGURE 3. Time evolution of the vorticity $\omega(x, y, t)$ over eight time units with $\nu = 0.001$ and a spatial domain $x \in [-10, 10]$ and $y \in [-10, 10]$. The initial condition was a stretched Gaussian of the form $\omega(x, y, 0) = \exp(-2x^2 - y^2/20)$.

by discretizing in both the x- and y-directions and solving the associated linear problem $\mathbf{Ax} = \mathbf{b}$. At best, we can use a factorization scheme to solve this problem in $O(N^2)$ operations. Although iteration schemes have the possibility of outperforming this, it is not guaranteed.

Another alternative is to use the fast Fourier transform (FFT). The FFT is an integral transform defined over the entire line $x \in [-\infty, \infty]$. Given computational practicalities, however, we transform over a finite domain $x \in [-L, L]$ and assume periodic boundary conditions due to the oscillatory behavior of the kernel of the Fourier transform. The Fourier transform and its inverse can be defined as

$$F(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx \tag{2a}$$

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} F(k) dk \,. \tag{2b}$$

There are other equivalent definitions. However, this definition will serve to illustrate the power and functionality of the Fourier transform method. We again note that formally, the transform is over the entire real line $x \in [-\infty, \infty]$ whereas our computational domain is only over a finite domain $x \in [-L, L]$. Further, the kernel of the transform, $\exp(\pm ikx)$, describes oscillatory behavior. Thus the Fourier transform is essentially an eigenfunction expansion over all continuous wavenumbers k. And once we are on a finite domain $x \in [-L, L]$, the continuous eigenfunction expansion becomes a discrete sum of eigenfunctions and associated wavenumbers (eigenvalues).

Derivative relations: The critical property in the usage of Fourier transforms concerns derivative relations. To see how these properties are generated, we begin by considering the Fourier transform of $f'(x)$. We denote the Fourier transform of $f(x)$ as $\widehat{f(x)}$. Thus we find

$$\widehat{f'(x)} = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f'(x) dx = f(x) e^{-ikx}|_{-\infty}^{\infty} + \frac{ik}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx \,. \tag{3}$$

Assuming that $f(x) \to 0$ as $x \to \pm\infty$ results in

$$\widehat{f'(x)} = \frac{ik}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx = ik\widehat{f(x)} \,. \tag{4}$$

Thus the basic relation $\widehat{f'} = ik\widehat{f}$ is established. It is easy to generalize this argument to an arbitrary number of derivatives. The final result is the following relation between Fourier transforms of the derivative and the Fourier transform itself

$$\widehat{f^{(n)}} = (ik)^n \widehat{f} \,. \tag{5}$$

This property is what makes Fourier transforms so useful and practical.

As an example of the Fourier transform, consider the following differential equation

$$y'' - \omega^2 y = -f(x) \quad x \in [-\infty, \infty] \,. \tag{6}$$

We can solve this by applying the Fourier transform to both sides. This gives the following reduction

$$\begin{aligned} \widehat{y''} - \omega^2 \widehat{y} &= -\widehat{f} \\ -k^2 \widehat{y} - \omega^2 \widehat{y} &= -\widehat{f} \\ (k^2 + \omega^2)\widehat{y} &= \widehat{f} \\ \widehat{y} &= \frac{\widehat{f}}{k^2 + \omega^2}. \end{aligned} \tag{7}$$

To find the solution $y(x)$, we invert the last expression above to yield

$$y(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} \frac{\widehat{f}}{k^2 + \omega^2} dk. \tag{8}$$

This gives the solution in terms of an integral which can be evaluated analytically or numerically.

3.1. The fast Fourier transform. The fast Fourier transform routine was developed specifically to perform the forward and backward Fourier transforms. In the mid-1960s, Cooley and Tukey developed what is now commonly known as the FFT algorithm [**44**]. Their algorithm was named one of the top ten algorithms of the twentieth century for one reason: the operation count for solving a system dropped to $O(N \log N)$. For N large, this operation count grows almost linearly like N. Thus it represents a great leap forward from Gaussian elimination and LU decomposition. The key features of the FFT routine are as follows:

(1) It has a low operation count: $O(N \log N)$.
(2) It finds the transform on an interval $x \in [-L, L]$. Since the integration kernel $\exp(\pm ikx)$ is oscillatory, it implies that the solutions on this finite interval have periodic boundary conditions.
(3) The key to lowering the operation count to $O(N \log N)$ is in discretizing the range $x \in [-L, L]$ into 2^n points, i.e. the number of points should be $2, 4, 8, 16, 32, 64, 128, 256, \cdots$.
(4) The FFT has excellent accuracy properties, typically well beyond that of standard discretization schemes.

We will consider the underlying FFT algorithm in detail at a later time. For more information at the present, see [**44**] for a broader overview.

3.2. The streamfunction. The FFT algorithm provides a fast and efficient method for solving the streamfunction equation

$$\nabla^2\psi = \omega \tag{9}$$

given the vorticity ω. In two dimensions, this equation is equivalent to

$$\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} = \omega\,. \tag{10}$$

Denoting the Fourier transform in x as $\widehat{f(x)}$ and the Fourier transform in y as $\widetilde{g(y)}$, we transform the equation. We begin by transforming in x:

$$\widehat{\frac{\partial^2\psi}{\partial x^2}} + \widehat{\frac{\partial^2\psi}{\partial y^2}} = \widehat{\omega} \quad \rightarrow \quad -k_x^2\widehat{\psi} + \frac{\partial^2\widehat{\psi}}{\partial y^2} = \widehat{\omega}\,, \tag{11}$$

where k_x are the wavenumbers in the x-direction. Transforming now in the y-direction gives

$$-k_x^2\widetilde{\widehat{\psi}} + \widetilde{\frac{\partial^2\widehat{\psi}}{\partial y^2}} = \widetilde{\widehat{\omega}} \quad \rightarrow \quad -k_x^2\widetilde{\widehat{\psi}} + -k_y^2\widetilde{\widehat{\psi}} = \widetilde{\widehat{\omega}}\,. \tag{12}$$

This can be rearranged to obtain the final result

$$\widetilde{\widehat{\psi}} = -\frac{\widetilde{\widehat{\omega}}}{k_x^2 + k_y^2}\,. \tag{13}$$

The remaining step is to inverse-transform in x and y to get back to the solution $\psi(x, y)$.

There is one mathematical difficulty which must be addressed. The streamfunction equation with periodic boundary conditions does not have a unique solution. Thus if $\psi_0(x, y, t)$ is a solution, so is $\psi_0(x, y, t) + c$ where c is an arbitrary constant. When solving this problem with FFTs, the FFT will arbitrarily add a constant to the solution. Fundamentally, we are only interested in derivatives of the streamfunction. Therefore, this constant is inconsequential. When solving with direct methods for $\mathbf{Ax} = \mathbf{b}$, the nonuniqueness gives a singular matrix $\mathbf{A}$. Thus solving with Gaussian elimination, LU decomposition or iterative methods is problematic. But since the arbitray constant does not matter, we can simply pin the streamfunction to some prescribed value on our computational domain. This will fix the constant c and give a unique solution to $\mathbf{Ax} = \mathbf{b}$. For instance, we could impose the following constraint condition $\psi(-L, -L, t) = 0$. Such a condition pins the value of $\psi(x, y, t)$ at the left-hand corner of the computational domain and fixes c.

3.3. python commands. The commands for executing the fast Fourier transform and its inverse are as follows

- $fft(x)$: Forward Fourier transform a vector $\mathbf{x}$.
- $ifft(x)$: Inverse Fourier transform a vector $\mathbf{x}$.

4. Comparison of Solution Techniques for Ax = b: Rules of Thumb

The practical implementation of the mathematical tools available in python is crucial. This section will focus on the use of some of the more sophisticated routines in python which are cornerstones to scientific computing. Included in this section will be a discussion of the fast Fourier transform routines (*fft, ifft, fftshift, ifftshift, fft2, ifft2*), sparse matrix construction (*spdiag, spy*), and high-end iterative techniques for solving $\mathbf{Ax} = \mathbf{b}$ (*bicgstab, gmres*). These routines should be studied carefully since they are the building blocks of any serious scientific computing code.

4.1. Fast Fourier transform: FFT, IFFT, FFTSHIFT, IFFTSHIFT. The fast Fourier transform will be the first subject discussed. Its implementation is straightforward. Given a function which has been discretized with 2^n points and represented by a vector $\mathbf{x}$, the FFT is found with the command *fft(x)*. Aside from transforming the function, the algorithm associated with the FFT does three major things: it shifts the data so that $x \in [0, L] \to [-L, 0]$ and $x \in [-L, 0] \to [0, L]$, additionally it multiplies every other mode by -1, and it assumes you are working on a 2π periodic domain. These properties are a consequence of the FFT algorithm discussed in detail at a later time.

To see the practical implications of the FFT, we consider the transform of a Gaussian function. The transform can be calculated analytically so that we have the exact relations:

$$f(x) = \exp(-\alpha x^2) \quad \to \quad \widehat{f}(k) = \frac{1}{\sqrt{2\alpha}} \exp\left(-\frac{k^2}{4\alpha}\right). \tag{1}$$

A simple python code to verify this with $\alpha = 1$ is as follows

```
L = 20  # define the computational domain [-L/2,L/2]
n = 128  # define the number of Fourier modes 2^n
x2 = np.linspace(-L/2, L/2, n+1) # Define the domain
x = x2[:n] # Consider only the first n points

u = np.exp(-x * x)
ut = np.fft.fft(u)
utshift = np.fft.fftshift(ut)
```

The second figure generated by this script shows how the pulse is shifted. By using the command *fftshift*, we can shift the transformed function back to its mathematically correct positions as shown in the third figure generated. However, before inverting the transformation, it is crucial that the transform is shifted back to the form of the second figure. The command *ifftshift* does this. In general, unless you need to plot the spectrum, it is better not to deal with the *fftshift* and *ifftshift* commands. A graphical representation of the *fft* procedure and its shifting properties is illustrated in Fig. 4 where a Gaussian is transformed and shifted by the *fft* routine.

To take a derivative, we need to calculate the k values associated with the transformation. The following example does this. Recall that the FFT assumes a 2π periodic domain which gets shifted. Thus the calculation of the k values needs to shift and rescale to the 2π domain. The following example differentiates the function $f(x) = \text{sech}(x)$ three times. The first derivative is compared with the analytic value of $f'(x) = -\text{sech}(x)\tanh(x)$.

```
def sech(x):
    return 1 / np.cosh(x)

def tanh(x):
    return np.sinh(x) / np.cosh(x)

L = 20
n = 128
x2 = np.linspace(-L/2, L/2, n+1)
x = x2[:n]

u = sech(x)
ut = np.fft.fft(u)
```

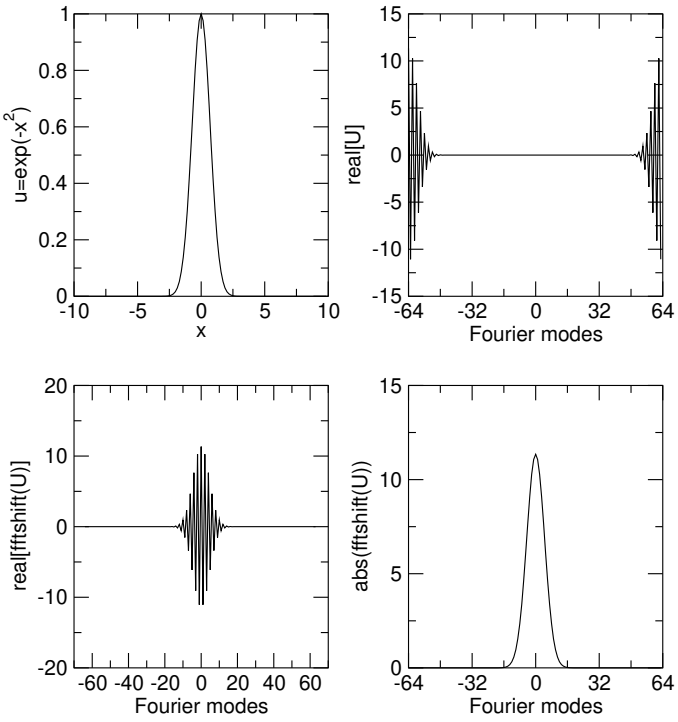

FIGURE 4. Fast Fourier Transform of Gaussian data illustrating the shifting properties of the FFT routine. Note that the *fftshift* command restores the transform to its mathematically correct, unshifted state.

```
k = (2 * np.pi / L) * np.concatenate((np.arange(0, n//2), np.arange(-n//2, 0)))
ut1 = 1j * k * ut # first derivative
ut2 = -k**2 * ut # second derivative
ut3 = -1j * k**3 * ut  # third derivative

u1 = np.fft.ifft(ut1) # Inverse transform
u2 = np.fft.ifft(ut2)
u3 = np.fft.ifft(ut3)
```

The routine accounts for the periodic boundaries, the correct k values, and differentiation. Note that no shifting was necessary since we constructed the k values in the shifted space.

For transforming in higher dimensions, a couple of choices in python are possible. For 2D transformations, it is recommended to use the commands *fft2* and *ifft2*. These will transform a matrix $\mathbf{A}$, which represents data in the x- and y-directions, respectively, along the rows and then columns. For higher dimensions, the *fft* command can be modified to *fft(x,[],N)* where N is the number of dimensions.

4.2. Sparse matrices: SPDIAGS, SPY. Under discretization, most physical problems yield sparse matrices, i.e. matrices which are largely composed of zeros. For instance, the matrices (11) and (13) are sparse matrices generated under the discretization of a boundary value problem and Poisson equation, respectively. The *spdiag* command allows for the construction of sparse matrices in a relatively simple fashion. The sparse matrix is then saved using a minimal amount of memory and all matrix operations are conducted as usual. The *spy* command allows you to look

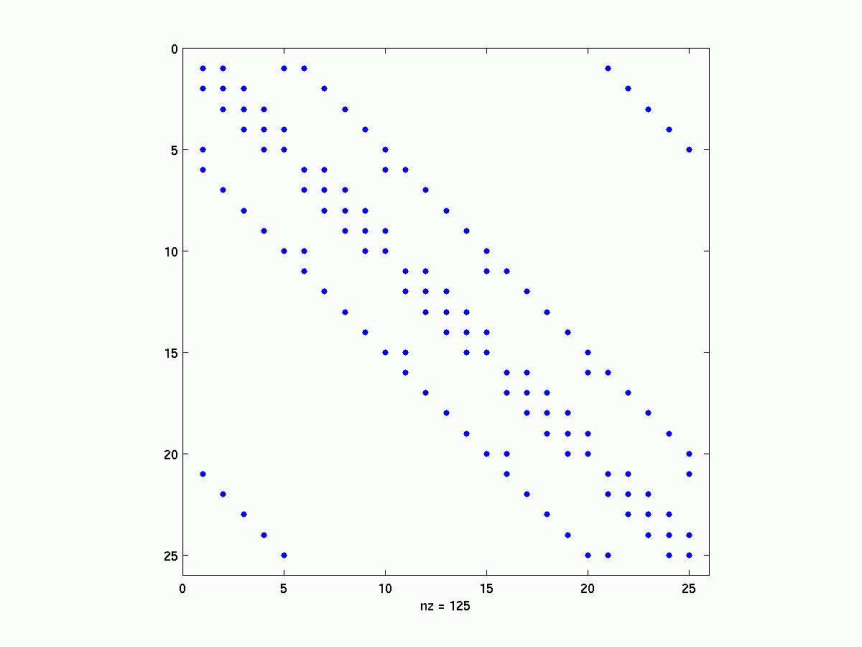

FIGURE 5. Sparse matrix structure for the Laplacian operator using second order discretization. The output is generated via the *spy* command in MATLAB.

at the nonzero components of the matrix structure (see Fig. 5). As an example, we construct the matrix given by (13) for the case of $N = 5$ in both the x- and y-directions.

```
from scipy.sparse import spdiags

m = 5    # N value in x and y directions
n = m * m  # total size of matrix

e0 = np.zeros((n, 1))  # vector of zeros
e1 = np.ones((n, 1))   # vector of ones
e2 = np.copy(e1)    # copy the one vector
e4 = np.copy(e0)    # copy the zero vector

for j in range(1, m+1):
    e2[m*j-1] = 0  # overwrite every m^th value with zero
    e4[m*j-1] = 1  # overwirte every m^th value with one

# Shift to correct positions
e3 = np.zeros_like(e2)
e3[1:n] = e2[0:n-1]
e3[0] = e2[n-1]

e5 = np.zeros_like(e4)
e5[1:n] = e4[0:n-1]
e5[0] = e4[n-1]

# Place diagonal elements
```

```
diagonals = [e1.flatten(), e1.flatten(), e5.flatten(),
             e2.flatten(), -4 * e1.flatten(), e3.flatten(),
             e4.flatten(), e1.flatten(), e1.flatten()]
offsets = [-(n-m), -m, -m+1, -1, 0, 1, m-1, m, (n-m)]

matA = spdiags(diagonals, offsets, n, n).toarray()
plt.spy(matA)
```

The appropriate construction of the sparse matrix is critical to solving any problem. It is essential to carefully check the matrix for accuracy before using it in an application. As can be seen from this example, it takes a bit of work to get the matrix correct. However, once constructed properly, it can be used with confidence in all other matrix applications and operations.

4.3. Iterative methods: BICGSTAB, GMRES. Iterative techniques need also to be considered. There are a wide variety of built-in iterative techniques in python. Two of the more promising methods are discussed here: the bi-conjugate stabilized gradient method (*bicgstab*) and the generalized minimum residual method (*gmres*). Both are easily implemented in python.

Recall that these iteration methods are for solving the linear system $\mathbf{Ax} = \mathbf{b}$. The basic call to the generalized minimum residual method is

```
from scipy.sparse.linalg import gmres
x = gmres(A, b)[0]
```

Likewise, the bi-conjugate stabilized gradient method is called by

```
from scipy.sparse.linalg import bicgstab
x, exit_code = bicgstab(A, b)
```

It is rare that you would use these commands without options. The iteration scheme stops upon convergence, failure to converge, or when the maximum number of iterations has been achieved. By default, the initial guess for the iteration procedure is the zero vector. The default number of maximum iterations is 10, which is rarely enough iterations for the purposes of scientific computing.

Thus it is important to understand the different options available with these iteration techniques. The most general user-specified command line for either *gmres* or *bicgstab* is as follows

```
x, flag, relres, iter = bicgstab(A, b, tol=tol,
     maxiter=maxit, M1=M1, M2=M2, x0=x0)
x, flag, relres, iter = gmres(A, b, tol=tol, restart=restart,
     maxiter=maxit, M=M1, x0=x0)
```

We already know that the matrix **A** and vector **b** come from the original problem. The remaining parameters are as follows:

tol = specified tolerance for convergence
maxit = maximum number of iterations
M1, M2 = preconditioning matrices
x0 = initial guess vector
restart = restart of iterations (gmres only).

In addition to these input parameters, *gmres* or *bicgstab* will give the relative residual, *relres*, and the number of iterations performed, *iter*. The *flag* variable gives information on whether the scheme converged in the maximum allowable iterations or not.

5. Overcoming Computational Difficulties

In developing algorithms for any given problem, you should always try to maximize the use of information available to you about the specific problem. Specifically, a well-developed numerical code will make extensive use of analytic information concerning the problem. Judicious use of properties of the specific problem can lead to significant reduction in the amount of computational time and effort needed to perform a given calculation.

Here, various practical issues which may arise in the computational implementation of a given method are considered. Analysis provides a strong foundation for understanding the issues which can be problematic on a computational level. Thus the focus here will be on details which need to be addressed before straightforward computing is performed.

5.1. Streamfunction equations: Nonuniqueness. We return to the consideration of the streamfunction equation

$$\nabla^2\psi = \omega \tag{1}$$

with the periodic boundary conditions

$$\psi(-L, y, t) = \psi(L, y, t) \tag{2a}$$

$$\psi(x, -L, t) = \psi(x, L, t)\,. \tag{2b}$$

The mathematical problem which arises in solving this Poisson equation has been considered previously. Namely, the solution can only be determined to an arbitrary constant. Thus if ψ_0 is a solution to the streamfunction equation, so is

$$\psi = \psi_0 + c, \tag{3}$$

where c is an arbitrary constant. This gives an infinite number of solutions to the problem. Thus upon discretizing the equation in x and y and formulating the matrix formulation of the problem $\mathbf{Ax} = \mathbf{b}$, we find that the matrix $\mathbf{A}$, which is given by (13), is singular, i.e. $\det \mathbf{A} = 0$.

Obviously, the fact that the matrix $\mathbf{A}$ is singular will create computational problems. However, does this nonuniqueness jeopardize the validity of the physical model? Recall that the streamfunction is a fictitious quantity which captures the fluid velocity through the quantities $\partial\psi/\partial x$ and $\partial\psi/\partial y$. In particular, we have the x and y velocity components

$$u = -\tfrac{\partial\psi}{\partial y} \quad (x\text{-component of velocity}) \tag{4a}$$

$$v = \tfrac{\partial\psi}{\partial x} \quad (y\text{-component of velocity}). \tag{4b}$$

Thus the arbitrary constant c drops out when calculating physically meaningful quantities. Further, when considering the advection–diffusion equation

$$\frac{\partial\omega}{\partial t} + [\psi, \omega] = \nu\nabla^2\omega \tag{5}$$

where

$$[\psi, \omega] = \frac{\partial\psi}{\partial x}\frac{\partial\omega}{\partial y} - \frac{\partial\psi}{\partial y}\frac{\partial\omega}{\partial x}\,, \tag{6}$$

only the derivative of the streamfunction is important, which again removes the constant c from the problem formulation.

We can thus conclude that the nonuniqueness from the constant c does not generate any problems for the physical model considered. However, we still have the mathematical problem of dealing with a singular matrix. It should be noted that this problem also occurs when all the boundary conditions are of the Neuman type, i.e. $\partial\psi/\partial n = 0$ where n denotes the outward normal direction.

To overcome this problem numerically, we simply observe that we can arbitrarily add a constant to the solution. Or alternatively, we can pin down the value of the streamfunction at a single location in the computational domain. This constraint fixes the arbitrary constant problem and removes the singularity from the matrix $\mathbf{A}$. Thus to fix the problem, we can simply pick an arbitrary point in our computational domain ψ_{mn} and fix its value. Essentially, this will alter a single component of the sparse matrix (13). And in fact, this is the simplest thing to do. For instance, given the construction of the matrix (13), we could simply add the following line of python code:

```
A[0, 0]=0
```

Then $\det \mathbf{A} \neq 0$ and the matrix can be used in any of the linear solution methods. Note that the choice of the matrix component and its value are completely arbitrary. However, in this example, if you choose to alter this matrix component, to overcome the matrix singularity you must have $A(1,1) \neq -4$.

5.2. Fast Fourier transforms: Divide by zero. In addition to solving the streamfunction equation by standard discretization and $\mathbf{Ax} = \mathbf{b}$, we could use the Fourier transform method. In particular, Fourier transforming in both x and y reduces the streamfunction equation

$$\nabla^2 \psi = \omega \tag{7}$$

to Eq. (13)

$$\widetilde{\widehat{\psi}} = -\frac{\widetilde{\widehat{\omega}}}{k_x^2 + k_y^2}\,. \tag{8}$$

Here we have denoted the Fourier transform in x as $\widehat{f(x)}$ and the Fourier transform in y as $\widetilde{g(y)}$. The final step is to inverse-transform in x and y to get back to the solution $\psi(x, y)$. It is recommended that the routines *fft2* and *ifft2* be used to perform the transformations. However, *fft* and *ifft* may also be used in loops or with the dimension option set to 2.

An observation concerning (8) is that there will be a divide by zero when $k_x = k_y = 0$ at the zero mode. Two options are commonly used to overcome this problem. The first is to modify (8) so that

$$\widetilde{\widehat{\psi}} = -\frac{\widetilde{\widehat{\omega}}}{k_x^2 + k_y^2 + eps} \tag{9}$$

where *eps* is the command for generating a machine precision number which is on the order of $O(10^{-15})$. It essentially adds a round-off to the denominator which removes the divide by zero problem. A second option, which is more highly recommended, is to redefine the $\mathbf{k}_x$ and $\mathbf{k}_y$ vectors associated with the wavenumbers in the x- and y-directions. Specifically, after defining the $\mathbf{k}_x$ and $\mathbf{k}_y$, we could simply add the command line

```
kx[0] = 1e-6
ky[0] = 1e-6
```

The values of $kx[0] = ky[0] = 0$ by default. This would make the values small but finite so that the divide by zero problem is effectively removed with only a small amount of error added to the problem.

5.3. Sparse derivative matrices: Advection terms. The sparse matrix (13) represents the discretization of the Laplacian which is accurate to second order. However, when calculating the advection terms given by

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x}\,, \tag{10}$$

only the first derivative is required in both x and y. Associated with each of these derivative terms is a sparse matrix which must be calculated in order to evaluate the advection term.

The calculation of the x derivative will be considered here. The y derivative can be calculated in a similar fashion. Obviously, it is crucial that the sparse matrices representing these operations be correct. Consider then the second-order discretization of

$$\frac{\partial \omega}{\partial x} = \frac{\omega(x+\Delta x, y) - \omega(x-\Delta x, y)}{2\Delta x}. \tag{11}$$

Using the notation developed previously, we define $\omega(x_m, y_n) = \omega_{mn}$. Thus the discretization yields

$$\frac{\partial \omega_{mn}}{\partial x} = \frac{\omega_{(m+1)n} - \omega_{(m-1)n}}{2\Delta x}, \tag{12}$$

with periodic boundary conditions. The first few terms of this linear system are as follows:

$$\begin{aligned} \frac{\partial \omega_{11}}{\partial x} &= \frac{\omega_{21} - \omega_{n1}}{2\Delta x} \\ \frac{\partial \omega_{12}}{\partial x} &= \frac{\omega_{22} - \omega_{n2}}{2\Delta x} \\ &\vdots \\ \frac{\partial \omega_{21}}{\partial x} &= \frac{\omega_{31} - \omega_{11}}{2\Delta x} \\ &\vdots \end{aligned} \tag{13}$$

From these individual terms, the linear system $\partial \boldsymbol{\omega}/\partial x = \mathbf{B}\boldsymbol{\omega}$ can be constructed. The critical component is the sparse matrix $\mathbf{B}$. Note that we have used the usual definition of the vector $\boldsymbol{\omega}$

$$\boldsymbol{\omega} = \begin{pmatrix} \omega_{11} \\ \omega_{12} \\ \vdots \\ \omega_{1n} \\ \omega_{21} \\ \omega_{22} \\ \vdots \\ \omega_{n(n-1)} \\ \omega_{nn} \end{pmatrix}. \tag{14}$$

It can be verified then that the sparse matrix $\mathbf{B}$ is given by the matrix

$$\mathbf{B} = \frac{1}{2\Delta x} \begin{bmatrix} \mathbf{0} & \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} & -\mathbf{I} \\ -\mathbf{I} & \mathbf{0} & \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & \mathbf{0} & -\mathbf{I} & \mathbf{0} & \mathbf{I} \\ \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} & -\mathbf{I} & \mathbf{0} \end{bmatrix}, \tag{15}$$

where $\mathbf{0}$ is an $n \times n$ zero matrix and $\mathbf{I}$ is an $n \times n$ identity matrix. Recall that the matrix $\mathbf{B}$ is an $n^2 \times n^2$ matrix. The sparse matrix associated with this operation can be constructed with the *spdiag* command. Following the same analysis, the y derivative matrix can also be constructed. If

we call this matrix $\mathbf{C}$, then the advection operator can simply be written as

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} = (\mathbf{B}\boldsymbol{\psi})(\mathbf{C}\boldsymbol{\omega}) - (\mathbf{C}\boldsymbol{\psi})(\mathbf{B}\boldsymbol{\omega})\,. \tag{16}$$

This forms part of the right-hand side of the system of differential equations which is then solved using a standard time-stepping algorithm.

6. Problems and Exercises

6.1. Advection–Diffusion and Atmospheric Dynamics. The shallow-water wave equations are of fundamental interest in several contexts. In one sense, the ocean can be thought of as a shallow-water description over the surface of the Earth. Thus the circulation and movement of currents can be studied. Second, the atmosphere can be thought of as a relatively thin layer of fluid (gas) above the surface of the Earth. Again the circulation and atmospheric dynamics are of general interest. The shallow-water approximation relies on a separation of scale: the height of the fluid (or gas) must be much less than the characteristic horizontal scales. The physical setting for shallow-water modeling is illustrated in Fig. 6. In this figure, the characteristic height is given by the parameter D while the characteristic horizontal fluctuations are given by the parameter L. For the shallow-water approximation to hold, we must have

$$\delta = \frac{D}{L} \ll 1. \tag{17}$$

This gives the necessary separation of scales for reducing the Navier–Stokes equations to the shallow-water description.

The motion of the layer of fluid is described by its velocity field

$$\mathbf{v} = \begin{pmatrix} u \\ v \\ w \end{pmatrix} \tag{18}$$

where u, v, and w are the velocities in the x-, y-, and z-directions, respectively. An alternative way to describe the motion of the fluid is through the quantity known as the vorticity. Roughly speaking the vorticity is a vector which measures the twisting of the fluid, i.e. $\boldsymbol{\Omega} = (\boldsymbol{\omega_x} \quad \boldsymbol{\omega_y} \quad \boldsymbol{\omega_z})^{\mathbf{T}}$. Since with shallow water we are primarily interested in the vorticity or fluid rotation in the $x-y$ plane, we define the vorticity of interest to be

$$\omega_z = \omega = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}. \tag{19}$$

This quantity will characterize the evolution of the fluid in the shallow-water approximation.

Conservation of mass: The equations of motion are a consequence of a few basic physical principles, one of those being conservation of mass. The implications of conservation of mass are that the time rate of change of mass in a volume must equal the net inflow/outflow through the boundaries of the volume. Consider a volume from Fig. 6 which is bounded between $x \in [x_1, x_2]$ and $y \in [y_1, y_2]$. The mass in this volume is given by

$$\text{mass} = \int_{x_1}^{x_2} \int_{y_1}^{y_2} \rho(x, y) h(x, y, t) dx dy \tag{20}$$

where $\rho(x, y)$ is the fluid density and $h(x, y, t)$ is the surface height. We will assume the density ρ is constant in what follows. The conservation of mass in integral form is then expressed as

$$\begin{aligned} &\frac{\partial}{\partial t} \int_{x_1}^{x_2} \int_{y_1}^{y_2} h(x, y, t) dx dy \\ &\quad + \int_{y_1}^{y_2} \left[u(x_2, y, t) h(x_2, y, t) - u(x_1, y, t) h(x_1, y, t) \right] dy \\ &\quad + \int_{x_1}^{x_2} \left[v(x, y_2, t) h(x, y_2, t) - v(x, y_1, t) h(x, y_1, t) \right] dx = 0, \end{aligned} \tag{21}$$

where the density has been divided out. Here the first term measures the rate of change of mass while the second and third terms measure the flux of mass across the x boundaries and y boundaries,

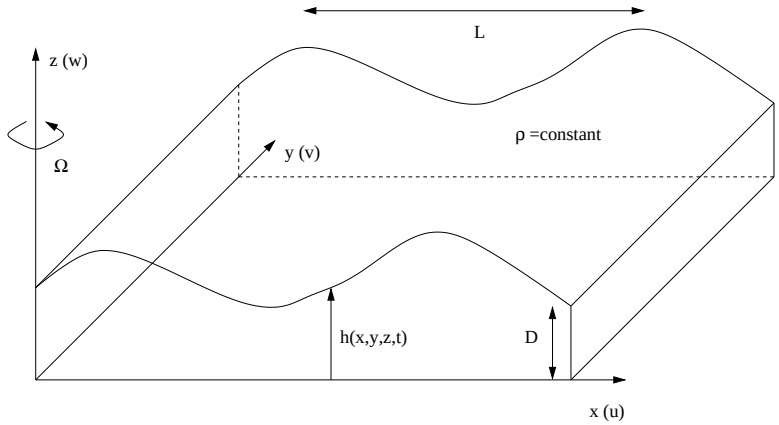

FIGURE 6. Physical setting of a shallow atmosphere or shallow water equations with constant density ρ and scale separation $D/L \ll 1$. The vorticity is measured by the parameter Ω.

respectively. Note that from the fundamental theorem of calculus, we can rewrite the second and third terms:

$$\int_{y_1}^{y_2} \left[u(x_2,y,t)h(x_2,y,t) - u(x_1,y,t)h(x_1,y,t)\right] dy = \int_{x_1}^{x_2}\int_{y_1}^{y_2} \frac{\partial}{\partial x}(uh)dxdy \tag{22a}$$

$$\int_{x_1}^{x_2} \left[v(x,y_2,t)h(x,y_2,t) - v(x,y_1,t)h(x,y_1,t)\right] dx = \int_{x_1}^{x_2}\int_{y_1}^{y_2} \frac{\partial}{\partial y}(vh)dxdy. \tag{22b}$$

Replacing these new expressions in (21) shows all terms to have a double integral over the volume. The integrand must then be identically zero for conservation of mass. This results in the expression

$$\frac{\partial h}{\partial t} + \frac{\partial}{\partial x}(hu) + \frac{\partial}{\partial y}(hv) = 0. \tag{23}$$

Thus a fundamental relationship is established from a simple first-principles argument. The second and third equations of motion for the shallow-water approximation result from the conservation of momentum in the x- and y-directions. These equations may also be derived directly from the Navier–Stokes equations or conservation laws.

Shallow-water equations: The conservation of mass and momentum generates the following three governing equations for the shallow-water description

$$\frac{\partial h}{\partial t} + \frac{\partial}{\partial x}(hu) + \frac{\partial}{\partial y}(hv) = 0 \tag{24a}$$

$$\frac{\partial}{\partial t}(hu) + \frac{\partial}{\partial x}\left(hu^2 + \frac{1}{2}gh^2\right) + \frac{\partial}{\partial y}\left(huv\right) = fhv \tag{24b}$$

$$\frac{\partial}{\partial t}(hv) + \frac{\partial}{\partial y}\left(hv^2 + \frac{1}{2}gh^2\right) + \frac{\partial}{\partial x}(huv) = -fhu. \tag{24c}$$

Various approximations are now used to reduce the governing equations to a more manageable form. The first is to assume that at leading order the fluid height $h(x,y,t)$ is constant. The conservation of mass equation (24a) then reduces to

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \tag{25}$$

which is referred to as the *incompressible flow* condition. Thus under this assumption, the fluid cannot be compressed.

Under the assumption of a constant height $h(x, y, t)$, the remaining two equations reduce to

$$\frac{\partial u}{\partial t} + 2u\frac{\partial u}{\partial x} + \frac{\partial}{\partial y}(uv) = fv \tag{26a}$$

$$\frac{\partial v}{\partial t} + 2v\frac{\partial v}{\partial y} + \frac{\partial}{\partial x}(uv) = -fu. \tag{26b}$$

To simplify further, take the y derivative of the first equation and the x derivative of the second equation. The new equations are

$$\frac{\partial^2 u}{\partial t \partial y} + 2\frac{\partial u}{\partial y}\frac{\partial u}{\partial x} + 2u\frac{\partial^2 u}{\partial x \partial y} + \frac{\partial^2}{\partial y^2}(uv) = f\frac{\partial v}{\partial y} \tag{27a}$$

$$\frac{\partial^2 v}{\partial t \partial x} + 2\frac{\partial v}{\partial x}\frac{\partial v}{\partial y} + 2v\frac{\partial^2 v}{\partial x \partial y} + \frac{\partial^2}{\partial x^2}(uv) = -f\frac{\partial u}{\partial x}. \tag{27b}$$

Subtracting the first equation from the second gives the following reductions

$$\begin{aligned}
&\frac{\partial}{\partial t}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) - 2\frac{\partial u}{\partial y}\frac{\partial u}{\partial x} - 2u\frac{\partial^2 u}{\partial x \partial y} - \frac{\partial^2}{\partial y^2}(uv) \\
&\quad + 2\frac{\partial v}{\partial x}\frac{\partial v}{\partial y} + 2v\frac{\partial^2 v}{\partial x \partial y} + \frac{\partial^2}{\partial x^2}(uv) = -f\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) \\
&\frac{\partial}{\partial t}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) + 2\frac{\partial u}{\partial x}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) + 2\frac{\partial v}{\partial y}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) \\
&\quad + u\left(\frac{\partial^2 v}{\partial x^2} - \frac{\partial^2 v}{\partial y^2} - 2\frac{\partial^2 u}{\partial x \partial y}\right) + v\left(\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} + 2\frac{\partial^2 v}{\partial x \partial y}\right) = -f\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) \\
&\frac{\partial}{\partial t}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) + u\frac{\partial}{\partial x}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) + v\frac{\partial}{\partial y}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) - u\frac{\partial}{\partial y}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) \\
&\quad + v\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) + 2\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right)\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) = -f\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right).
\end{aligned} \tag{28}$$

The final equation is reduced greatly by recalling the definition of the vorticity (19) and using the imcompressibility condition (25). The governing equations then reduce to

$$\frac{\partial \omega}{\partial t} + u\frac{\partial \omega}{\partial x} + v\frac{\partial \omega}{\partial y} = 0. \tag{29}$$

This gives the governing evolution of a shallow-water fluid in the absence of diffusion.

The streamfunction: It is typical in many fluid dynamics problems to work with the quantity known as the streamfunction. The streamfunction $\psi(x, y, t)$ is defined as follows:

$$u = -\frac{\partial \psi}{\partial y} \quad v = \frac{\partial \psi}{\partial x}. \tag{30}$$

Thus the streamfunction is specified up to an arbitrary constant. Note that the streamfunction automatically satisfies the imcompressibility condition since

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = -\frac{\partial^2 \psi}{\partial x \partial y} + \frac{\partial^2 \psi}{\partial x \partial y} = 0. \tag{31}$$

In terms of the vorticity, the streamfunction is related as follows:

$$\omega = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = \nabla^2 \psi. \tag{32}$$

This gives a second equation of motion which must be considered in solving the shallow-water equations.

Advection–diffusion: The advection of a fluid is governed by the evolution (29). In the presence of frictional forces, modification of this governing equation occurs. Specifically, the motion in the shallow-water limit is given by

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega \tag{33a}$$

$$\nabla^2 \psi = \omega \tag{33b}$$

where

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} \tag{34}$$

and $\nabla^2 = \partial_x^2 + \partial_y^2$ is the two-dimensional Laplacian. The diffusion component which is proportional to ν measures the frictional forces present in the fluid motion.

The advection–diffusion equations have the characteristic behavior of the three partial differential equation classifications: parabolic, elliptic and hyperbolic:

$$\text{parabolic:} \qquad \frac{\partial \omega}{\partial t} = \nu \nabla^2 \omega \tag{35a}$$

$$\text{elliptic:} \qquad \nabla^2 \psi = \omega \tag{35b}$$

$$\text{hyperbolic:} \qquad \frac{\partial \omega}{\partial t} + [\psi, \omega] = 0. \tag{35c}$$

Two things need to be solved for as a function of time:

$$\psi(x, y, t) \quad \text{streamfunction} \tag{36a}$$

$$\omega(x, y, t) \quad \text{vorticity.} \tag{36b}$$

We are given the initial vorticity $\omega_0(x, y)$ and periodic boundary conditions. The solution procedure is as follows:

- **Elliptic solve** Solve the elliptic problem $\nabla^2 \psi = \omega_0$ to find the streamfunction at time zero $\psi(x, y, t = 0) = \psi_0$.
- **Time-stepping** Given now ω_0 and ψ_0, solve the advection–diffusion problem by time-stepping with a given method. The Euler method is illustrated below

  $$\omega(x, y, t + \Delta t) = \omega(x, y, t) + \Delta t \left(\nu \nabla^2 \omega(x, y, t) - [\psi(x, y, t), \omega(x, y, t)] \right).$$

  This advances the solution Δt into the future.
- **Loop** With the updated value of $\omega(x, y, \Delta t)$, we can repeat the process by again solving for $\psi(x, y, \Delta t)$ and updating the vorticity once again.

This gives the basic algorithmic structure which must be followed in order to generate the solution for the vorticity and streamfunction as a function of time. It only remains to discretize the problem and solve.

Project and application: The time evolution of the vorticity $\omega(x, y, t)$ and streamfunction $\psi(x, y, t)$ are given by the governing equation:

$$\omega_t + [\psi, \omega] = \nu \nabla^2 \omega \tag{37}$$

where $[\psi, \omega] = \psi_x \omega_y - \psi_y \omega_x$, $\nabla^2 = \partial_x^2 + \partial_y^2$, and the streamfunction satisfies

$$\nabla^2 \psi = \omega. \tag{38}$$

Initial conditions. Assume a Gaussian shaped mound of initial vorticity for $\omega(x, y, 0)$. In particular, assume that the vorticity is elliptical with a ratio of 4:1 or more between the width of the Gaussian in the x- and y-directions. I'll let you pick the initial amplitude (one is always a good start).

Diffusion. In most applications, the diffusion is a small parameter. This fact helps the numerical stability considerably. Here, take $\nu = 0.001$.

Boundary conditions. Assume periodic boundary conditions for both the vorticity and streamfunction. Also, I'll let you experiment with the size of your domain. One of the restrictions is that the initial Gaussian lump of vorticity should be well-contained within your spatial domains.

Numerical integration procedure. Discretize (second-order) the vorticity equation and use ODE23 to step forward in time.

(a) Solve these equations where for the streamline ($\nabla^2 \psi = \omega$) use a fast Fourier transform.
(b) Solve these equations where for the streamline ($\nabla^2 \psi = \omega$) use the following methods:
    (1) A/b

(2) LU decomposition
(3) BICGSTAB
(4) GMRES.

Compare all of these methods with your FFT routine developed in part (a) (check out the CPUTIME command for MATLAB). In particular, keep track of the computationl speed of each method. Also, for BICGSTAB and GMRES, for the first few times solving the streamfunction equations, keep track of the residual and number of iterations needed to converge to the solution. Note that you should adjust the tolerance settings in BICGSTAB and GMRES to be consistent with your accuracy in the time-stepping. Experiment with the tolerance to see how much more quickly these iteration schemes converge.

(c) Try out these initial conditions with your favorite/fastest solver on the streamfunction equations.
   (1) Two oppositely "charged" Gaussian vorticies next to each other, i.e. one with positive amplitude, the other with negative amplitude.
   (2) Two same "charged" Gaussian vorticies next to each other.
   (3) Two pairs of oppositely "charged" vorticies which can be made to collide with each other.
   (4) A random assortment (in position, strength, charge, ellipticity, etc.) of vorticies on the periodic domain. Try 10–15 vorticies and watch what happens.

(d) Make a 2D movie of the dynamics. Color and coolness are key here. (MATLAB command: movie, getframe). I would very much like to see everyone's movies.

CHAPTER 10

Time and Space Stepping Schemes: Method of Lines

With the Fourier transform and discretization in hand, we can turn towards the solution of partial differential equations whose solutions need to be advanced forward in time. Thus, in addition to spatial discretization, we will need to discretize in time in a self-consistent way so as to advance the solution to a desired future time. Issues of stability and accuracy are, of course, at the heart of a discussion on time- and space-stepping schemes.

1. Basic Time-Stepping Schemes

Basic time-stepping schemes involve discretization in both space and time. Issues of numerical stability as the solution is propagated in time and accuracy from the space–time discretization are of greatest concern for any implementation. To begin this study, we will consider very simple and well-known examples from partial differential equations. The first equation we consider is the heat (diffusion) equation

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} \tag{1}$$

with the periodic boundary conditions $u(-L,t) = u(L,t)$. We discretize the spatial derivative with a second-order scheme (see Table 1) so that

$$\frac{\partial u}{\partial t} = \frac{\kappa}{\Delta x^2}\left[u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)\right] . \tag{2}$$

This approximation reduces the partial differential equation to a system of ordinary differential equations. We have already considered a variety of time-stepping schemes for differential equations and we can apply them directly to this resulting system.

1.1. The ODE system. To define the system of ODEs, we discretize and define the values of the vector $\mathbf{u}$ in the following way.

$$\begin{aligned}
&u(-L,t) = u_1\\
&u(-L+\Delta x,t) = u_2\\
&\vdots\\
&u(L-2\Delta x,t) = u_{n-1}\\
&u(L-\Delta x,t) = u_n\\
&u(L,t) = u_{n+1}\,.
\end{aligned}$$

Recall from periodicity that $u_1 = u_{n+1}$. Thus our system of differential equations solves for the vector

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} . \tag{3}$$

The governing equation (2) is then reformulated as the differential equation system

$$\frac{d\mathbf{u}}{dt} = \frac{\kappa}{\Delta x^2}\mathbf{A}\mathbf{u}\,, \tag{4}$$

where $\mathbf{A}$ is given by the sparse matrix

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 & 1 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & 0 & 1 & -2 & 1 \\ 1 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix}, \tag{5}$$

and the values of 1 on the upper right and lower left of the matrix result from the periodic boundary conditions.

1.2. python implementation. The system of differential equations can now be easily solved with a standard time-stepping algorithm such as *ode23* or *ode45*. The basic algorithm would be as follows

(1) Build the sparse matrix $\mathbf{A}$.

```
e1 = np.ones(n) # Build a vector of ones
diagonals = [e1, -2*e1, e1]
offsets = [-1, 0, 1]
A = diags(diagonals, offsets, shape=(n, n), format='csr')
A[0, n-1] = 1 # Periodic boundaries
A[n-1, 0] = 1
```

2 Generate the desired initial condition vector $\mathbf{u} = \mathbf{u}_0$.

3 Call an ODE solver from the python suite. The matrix $\mathbf{A}$, the diffusion constant κ and spatial step Δx need to be passed into this routine.

```
def rhs(u, t, k, dx, A):   # Define the right-hand side
    return (k / dx**2) * A.dot(u)

u0 = np.exp(-x**2) # Initial condition
y = odeint(rhs, u0, tspan, args=(k, dx, A)) # Solve ODE
```

4 Plot the results as a function of time and space.

The algorithm is thus fairly routine and requires very little effort in programming since we can make use of the standard time-stepping algorithms already available in python.

1.3. 2D python implementation. In the case of two dimensions, the calculation becomes slightly more difficult since the 2D data are represented by a matrix and the ODE solvers require a vector input for the initial data. For this case, the governing equation is

$$\frac{\partial u}{\partial t} = \kappa\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right). \tag{6}$$

Discretizing in x and y gives a right-hand side which takes on the form of (7). Provided $\Delta x = \Delta y = \delta$ are the same, the system can be reduced to the linear system

$$\frac{d\mathbf{u}}{dt} = \frac{\kappa}{\delta^2}\mathbf{A}\mathbf{u}\,, \tag{7}$$

where we have arranged the vector $\mathbf{u}$ in a similar fashion to (14) so that

$$\mathbf{u} = \begin{pmatrix} u_{11} \\ u_{12} \\ \vdots \\ u_{1n} \\ u_{21} \\ u_{22} \\ \vdots \\ u_{n(n-1)} \\ u_{nn} \end{pmatrix}, \tag{8}$$

where we have defined $u_{jk} = u(x_j, y_k)$. The matrix $\mathbf{A}$ is a nine diagonal matrix given by (13). The sparse implementation of this matrix is also given previously.

Again, the system of differential equations can now be easily solved with a standard time-stepping algorithm such as *ode23* or *ode45*. The basic algorithm follows the same course as the 1D case, but extra care is taken in arranging the 2D data into a vector.

(1) Build the sparse matrix $\mathbf{A}$ (13).
(2) Generate the desired initial condition matrix $\mathbf{U} = \mathbf{U}_0$ and reshape it to a vector $\mathbf{u} = \mathbf{u}_0$. This example considers the case of a simple Gaussian as the initial condition. The *reshape* and *meshgrid* commands are important for computational implementation.

```
Lx = 20    # spatial domain of x
Ly = 20    # spatial domain of y
nx = 100   # number of discretization points in x
ny = 100   # number of discretization points in y
N = nx * ny   # elements in reshaped initial condition

x2 = np.linspace(-Lx/2, Lx/2, nx+1) # x domain
x = x2[:nx]
y2 = np.linspace(-Ly/2, Ly/2, ny+1) # y domain
y = y2[:ny]
X, Y = np.meshgrid(x, y)  # make 2D

U = np.exp(-X**2 - Y**2) # Generate a Gaussian matrix
u = U.flatten()[:N].reshape(N, 1) # Reshape into a vector
```

3 Call an ODE solver from the python suite. The matrix $\mathbf{B}$, the diffusion constant κ and spatial step $\Delta x = \Delta y = dx$ need to be passed into this routine.

```
y = odeint(rhs, u0, tspan, args=(k, dx, B)) # Solve ODE
```

The function *rhs.m* should be of the following form

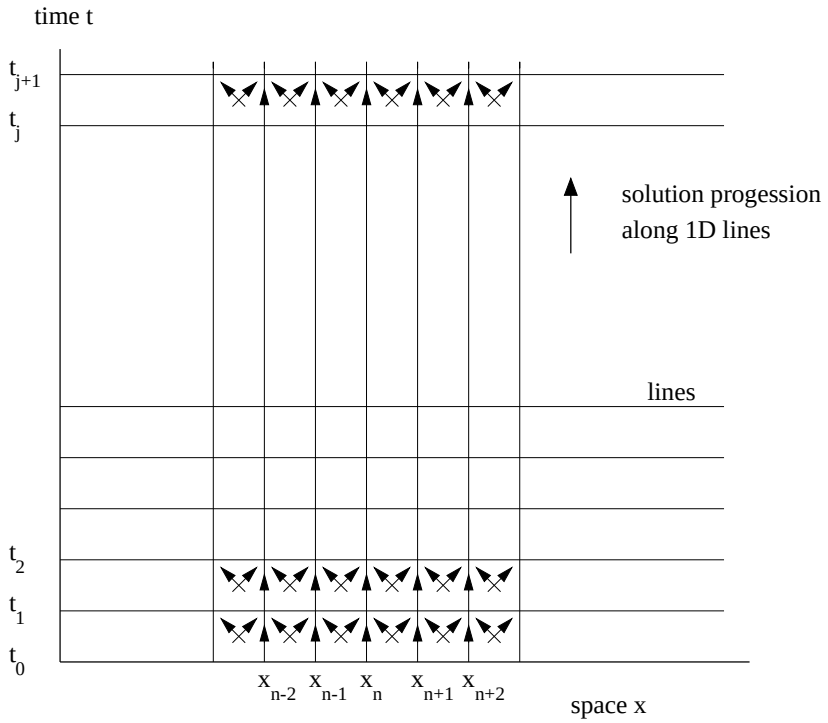

FIGURE 1. Graphical representation of the progression of the numerical solution using the method of lines. Here a second order discretization scheme is considered which couples each spatial point with its nearest neighbor.

```
def rhs(u, t, k, dx, B):   # Define the right-hand side
    return (k / dx**2) * B.dot(u)
```

4 Reshape and plot the results as a function of time and space.

The algorithm is again fairly routine and requires very little effort in programming since we can make use of the standard time-stepping algorithms.

1.4. Method of lines. Fundamentally, these methods use the data at a single slice of time to generate a solution Δt in the future. This is then used to generate a solution $2\Delta t$ into the future. The process continues until the desired future time is achieved. This process of solving a partial differential equation is known as the *method of lines*. Each *line* is the value of the solution at a given time slice. The lines are used to update the solution to a new timeline and progressively generate future solutions. Figure 1 depicts the process involved in the method of lines for the 1D case. The 2D case was illustrated previously in Fig. 2.

2. Time-Stepping Schemes: Explicit and Implicit Methods

Now that several technical and computational details have been addressed, we continue to develop methods for time-stepping the solution into the future. Some of the more common schemes will be considered along with a graphical representation of the scheme. Every scheme eventually leads to an iteration procedure which the computer can use to advance the solution in time.

We will begin by considering the simplest partial differential equations. Often, it is difficult to do much analysis with complicated equations. Therefore, considering simple equations is not merely an exercise, but rather they are typically the only equations we can make analytical progress with.

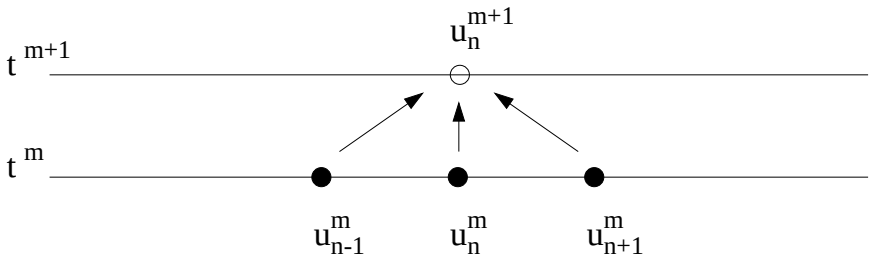

FIGURE 2. Four-point stencil for second-order spatial discretization and Euler time-stepping of the one-wave wave equation.

As an example, we consider the one-way wave equation

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x}\,. \tag{1}$$

The simplest discretization of this equation is to first central-difference in the x-direction. This yields

$$\frac{du_n}{dt} = \frac{c}{2\Delta x}\left(u_{n+1} - u_{n-1}\right)\,, \tag{2}$$

where $u_n = u(x_n, t)$. We can then step forward with an Euler time-stepping method. Denoting $u_n^{(m)} = u(x_n, t_m)$, and applying the method of lines iteration scheme gives

$$u_n^{(m+1)} = u_n^{(m)} + \frac{c\Delta t}{2\Delta x}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right)\,. \tag{3}$$

This simple scheme has the four-point stencil shown in Fig. 2. To illustrate more clearly the iteration procedure, we rewrite the discretized equation in the form

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{4}$$

where

$$\lambda = \frac{c\Delta t}{\Delta x} \tag{5}$$

is known as the CFL (Courant, Friedrichs and Lewy) number [**45**]. The iteration procedure assumes that the solution does not change significantly from one time-step to the next, i.e. $u_n^{(m)} \approx u_n^{(m+1)}$. The accuracy and stability of this scheme is controlled almost exclusively by the CFL number λ. This parameter relates the spatial and time discretization schemes in (5). Note that decreasing Δx without decreasing Δt leads to an increase in λ which can result in instabilities. Smaller values of Δt suggest smaller values of λ and improved numerical stability properties. In practice, you want to take Δx and Δt as large as possible for computational speed and efficiency without generating instabilities.

There are practical considerations to keep in mind relating to the CFL number. First, given a spatial discretization step-size Δx, you should choose the time discretization so that the CFL number is kept in check. Often a given scheme will only work for CFL conditions below a certain value, thus the importance of choosing a small enough time-step. Second, if indeed you choose to work with very small Δt, then although stability properties are improved with a lower CFL number, the code will also slow down accordingly. Thus achieving good stability results is often counter-productive to fast numerical solutions.

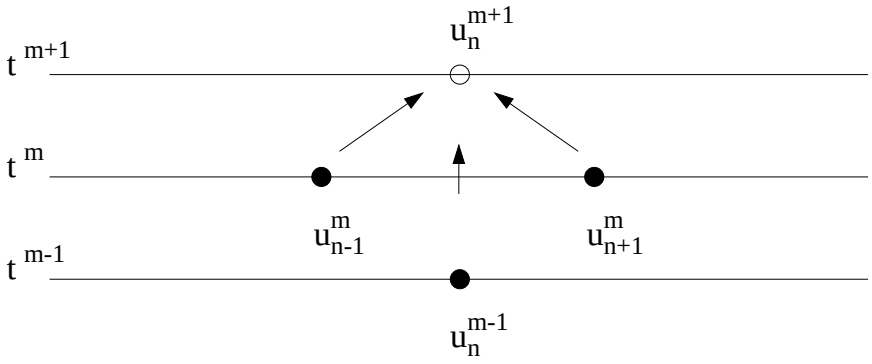

FIGURE 3. Four-point stencil for second-order spatial discretization and central-difference time-stepping of the one-wave wave equation.

2.1. Central differencing in time. We can discretize the time-step in a similar fashion to the spatial discretization. Instead of the Euler time-stepping scheme used above, we could central-difference in time using Table 1. Thus after spatial discretization we have

$$\frac{du_n}{dt} = \frac{c}{2\Delta x}\left(u_{n+1} - u_{n-1}\right), \tag{6}$$

as before. And using a central difference scheme in time now yields

$$\frac{u_n^{(m+1)} - u_n^{(m-1)}}{2\Delta t} = \frac{c}{2\Delta x}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right). \tag{7}$$

This last expression can be rearranged to give

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right). \tag{8}$$

This iterative scheme is called *leap-frog (2,2)* since it is $O(\Delta t^2)$ accurate in time and $O(\Delta x^2)$ accurate in space. It uses a four-point stencil as shown in Fig. 3. Note that the solution utilizes two time slices to leap-frog to the next time slice. Thus the scheme is not self-starting since only one time slice (initial condition) is given.

2.2. Improved accuracy. We can improve the accuracy of any of the above schemes by using higher order central differencing methods. The fourth-order accurate scheme from Table 2 gives

$$\frac{\partial u}{\partial x} = \frac{-u(x+2\Delta x) + 8u(x+\Delta x) - 8u(x-\Delta x) + u(x-2\Delta x)}{12\Delta x}. \tag{9}$$

Combining this fourth-order spatial scheme with second-order central differencing in time gives the iterative scheme

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left[\frac{4}{3}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) - \frac{1}{6}\left(u_{n+2}^{(m)} - u_{n-2}^{(m)}\right)\right]. \tag{10}$$

This scheme, which is based upon a six-point stencil, is called *leap-frog (2,4)*. It is typical that for *(2,4)* schemes, the maximum CFL number for stable computations is reduced from the basic *(2,2)* scheme.

2.3. Lax–Wendroff. Another alternative to discretizing in time and space involves a clever use of the Taylor expansion

$$u(x, t+\Delta t) = u(x,t) + \Delta t\frac{\partial u(x,t)}{\partial t} + \frac{\Delta t^2}{2!}\frac{\partial^2 u(x,t)}{\partial t^2} + O(\Delta t^3). \tag{11}$$

Scheme	Stability
Forward Euler	unstable for all λ
Backward Euler	stable for all λ
Leap Frog (2,2)	stable for $\lambda \leq 1$
Leap Frog (2,4)	stable for $\lambda \leq 0.707$

TABLE 1. Stability of time-stepping schemes as a function of the CFL number.

But we note from the governing one-way wave equation

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x} \approx \frac{c}{2\Delta x}\left(u_{n+1} - u_{n-1}\right) . \tag{12a}$$

Taking the derivative of the equation results in the relation

$$\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2} \approx \frac{c^2}{\Delta x^2}\left(u_{n+1} - 2u_n + u_{n-1}\right) . \tag{13a}$$

These two expressions for $\partial u/\partial t$ and $\partial^2 u/\partial t^2$ can be substituted into the Taylor series expression to yield the iterative scheme

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) + \frac{\lambda^2}{2}\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right) . \tag{14}$$

This iterative scheme is similar to the Euler method. However, it introduces an important *stabilizing* diffusion term which is proportional to λ^2. This is known as the *Lax–Wendroff* scheme. Although useful for this example, it is difficult to implement in practice for variable coefficient problems. It illustrates, however, the variety and creativity in developing iteration schemes for advancing the solution in time and space.

2.4. Backward Euler: Implicit scheme. The backward Euler method uses the future time for discretizing the spatial domain. Thus upon discretizing in space and time we arrive at the iteration scheme

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m+1)} - u_{n-1}^{(m+1)}\right) . \tag{15}$$

This gives the tridiagonal system

$$u_n^{(m)} = -\frac{\lambda}{2}u_{n+1}^{(m+1)} + u_n^{(m+1)} + \frac{\lambda}{2}u_{n-1}^{(m+1)} , \tag{16}$$

which can be written in matrix form as

$$\mathbf{A}\mathbf{u}^{(m+1)} = \mathbf{u}^{(m)} \tag{17}$$

where

$$\mathbf{A} = \frac{1}{2}\begin{pmatrix} 2 & -\lambda & \cdots & 0 \\ \lambda & 2 & -\lambda & \cdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & \lambda & 2 \end{pmatrix} . \tag{18}$$

Thus before stepping forward in time, we must solve a matrix problem. This can severely affect the computational time of a given scheme. The only thing which may make this method viable is if the CFL condition is such that much larger time-steps are allowed, thus overcoming the limitations imposed by the matrix solve.

2.5. MacCormack scheme. In the MacCormack scheme, the variable coefficient problem of the Lax–Wendroff scheme and the matrix solve associated with the backward Euler are circumvented by using a predictor–corrector method. The computation thus occurs in two pieces:

$$u_n^{(P)} = u_n^{(m)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{19a}$$

$$u_n^{(m+1)} = \frac{1}{2}\left[u_n^{(m)} + u_n^{(P)} + \lambda\left(u_{n+1}^{(P)} - u_{n-1}^{(P)}\right)\right]. \tag{19b}$$

This method essentially combines forward and backward Euler schemes so that we avoid the matrix solve and the variable coefficient problem.

The CFL condition will be discussed in detail in the next section. For now, the basic stability properties of many of the schemes considered here are given in Table 1. The stability of the various schemes holds only for the one-way wave equation considered here as a prototypical example. Each partial differential equation needs to be considered and classified individually with regards to stability.

3. Stability Analysis

In the preceeding section, we considered a variety of schemes which can solve the time–space problem posed by partial differential equations. However, it remains undetermined which scheme is best for implementation purposes. Two criteria provide a basis for judgement: accuracy and stability. For each scheme, we already know the accuracy due to the discretization error. However, a determination of stability is still required.

To start to understand stability, we once again consider the one-way wave equation

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x}. \tag{1}$$

Using Euler time-stepping and central differencing in space gives the iteration procedure

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{2}$$

where λ is the CFL number.

3.1. von Neumann analysis. To determine the stability of a given scheme, we perform a *von Neumann analysis* [**46**]. This assumes the solution is of the form

$$u_n^{(m)} = g^m \exp(i\xi_n h) \quad \xi \in [-\pi/h, \pi/h] \tag{3}$$

where $h = \Delta x$ is the spatial discretization parameter. Essentially this assumes the solution can be constructed of Fourier modes. The key then is to determine what happens to g^m as $m \to \infty$. Two possibilities exist

$$\lim_{m\to\infty} |g|^m \to \infty \qquad \text{unstable scheme} \tag{4a}$$

$$\lim_{m\to\infty} |g|^m \leq 1 \ (< \infty) \qquad \text{stable scheme.} \tag{4b}$$

Thus this stability check is very much like that performed for the time-stepping schemes developed for ordinary differential equations.

3.2. Forward Euler for one-way wave equation. The Euler discretization of the one-way wave equation produces the iterative scheme (2). Plugging in the ansatz (3) gives the following relations:

$$\begin{aligned} g^{m+1}\exp(i\xi_n h) &= g^m \exp(i\xi_n h) + \frac{\lambda}{2}\left(g^m \exp(i\xi_{n+1}h) - g^m \exp(i\xi_{n-1}h)\right) \\ g^{m+1}\exp(i\xi_n h) &= g^m\left(\exp(i\xi_n h) + \frac{\lambda}{2}\left(\exp(i\xi_{n+1}h) - \exp(i\xi_{n-1}h)\right)\right) \\ g &= 1 + \frac{\lambda}{2}\left[\exp(ih(\xi_{n+1} - \xi_n)) - \exp(ih(\xi_{n-1} - \xi_n))\right]. \end{aligned} \tag{5}$$

Letting $\xi_n = n\zeta$ reduces the equations further to

$$\begin{aligned} g(\zeta) &= 1 + \frac{\lambda}{2}[\exp(i\zeta h) - \exp(-i\zeta h)] \\ g(\zeta) &= 1 + i\lambda \sin(\zeta h)\,. \end{aligned} \tag{6}$$

From this we can deduce that

$$|g(\zeta)| = \sqrt{g^*(\zeta) g(\zeta)} = \sqrt{1 + \lambda^2 \sin^2 \zeta h}\,. \tag{7}$$

Thus

$$|g(\zeta)| \geq 1 \quad \rightarrow \quad \lim_{m\to\infty} |g|^m \to \infty \quad \text{unstable scheme.} \tag{8}$$

Thus the forward Euler time-stepping scheme is unstable for any value of λ. The implementation of this scheme will force the solution to infinity due to numerical instability.

3.3. Backward Euler for one-way wave equation. The Euler discretization of the one-way wave equation produces the iterative scheme

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m+1)} - u_{n-1}^{(m+1)}\right)\,. \tag{9}$$

Plugging in the ansatz (3) gives the following relations:

$$\begin{aligned} g^{m+1}\exp(i\xi_n h) &= g^m \exp(i\xi_n h) + \frac{\lambda}{2}\left(g^{m+1}\exp(i\xi_{n+1}h) - g^{m+1}\exp(i\xi_{n-1}h)\right) \\ g &= 1 + \frac{\lambda}{2} g\left[\exp(ih(\xi_{n+1} - \xi_n)) - \exp(ih(\xi_{n-1} - \xi_n))\right]\,. \end{aligned} \tag{10}$$

Letting $\xi_n = n\zeta$ reduces the equations further to

$$\begin{aligned} g(\zeta) &= 1 + \frac{\lambda}{2} g[\exp(i\zeta h) - \exp(-i\zeta h)] \\ g(\zeta) &= 1 + i\lambda g \sin(\zeta h) \\ g[1 - i\lambda \sin(\zeta h)] &= 1 \\ g(\zeta) &= \frac{1}{1 - i\lambda \sin \zeta h}\,. \end{aligned} \tag{11}$$

From this we can deduce that

$$|g(\zeta)| = \sqrt{g^*(\zeta) g(\zeta)} = \sqrt{\frac{1}{1 + \lambda^2 \sin^2 \zeta h}}\,. \tag{12}$$

Thus

$$|g(\zeta)| \leq 1 \quad \rightarrow \quad \lim_{m\to\infty} |g|^m \leq 1 \quad \text{unconditionally stable.} \tag{13}$$

Thus the backward Euler time-stepping scheme is stable for any value of λ. The implementation of this scheme will not force the solution to infinity.

3.4. Lax–Wendroff for one-way wave equation. The Lax–Wendroff scheme is not as transparent as the forward and backward Euler schemes.

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) + \frac{\lambda^2}{2}\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right)\,. \tag{14}$$

Plugging in the standard ansatz (3) gives

$$\begin{aligned} g^{m+1}\exp(i\xi_n h) &= g^m\exp(i\xi_n h) + \frac{\lambda}{2}g^m\left(\exp(i\xi_{n+1}h) - \exp(i\xi_{n-1}h)\right) \\ &\quad + \frac{\lambda^2}{2}g^m\left(\exp(i\xi_{n+1}h) - 2\exp(i\xi_n h) + \exp(i\xi_{n-1}h)\right) \\ g &= 1 + \frac{\lambda}{2}\left[\exp(ih(\xi_{n+1}-\xi_n)) - \exp(ih(\xi_{n-1}-\xi_n))\right] \\ &\quad + \frac{\lambda^2}{2}\left[\exp(ih(\xi_{n+1}-\xi_n)) + \exp(ih(\xi_{n-1}-\xi_n)) - 2\right]. \end{aligned} \tag{15}$$

Letting $\xi_n = n\zeta$ reduces the equations further to

$$\begin{aligned} g(\zeta) &= 1 + \frac{\lambda}{2}[\exp(i\zeta h) - \exp(-i\zeta h)] + \frac{\lambda^2}{2}[\exp(i\zeta h) + \exp(-i\zeta h) - 2] \\ g(\zeta) &= 1 + i\lambda\sin\zeta h + \lambda^2(\cos\zeta h - 1) \\ g(\zeta) &= 1 + i\lambda\sin\zeta h - 2\lambda^2\sin^2(\zeta h/2). \end{aligned} \tag{16}$$

This results in the relation

$$|g(\zeta)|^2 = \lambda^4\left[4\sin^4(\zeta h/2)\right] + \lambda^2\left[\sin^2\zeta h - 4\sin^2(\zeta h/2)\right] + 1. \tag{17}$$

This expression determines the range of values for the CFL number λ for which $|g| \le 1$. Ultimately, the stability of the Lax–Wendroff scheme for the one-way wave equation is determined.

3.5. Leap-frog (2,2) for one-way wave equation. The leap-frog discretization for the one-way wave equation yields the iteration scheme

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right). \tag{18}$$

Plugging in the ansatz (3) gives the following relations:

$$\begin{aligned} g^{m+1}\exp(i\xi_n h) &= g^{m-1}\exp(i\xi_n h) + \lambda g^m\left(\exp(i\xi_{n+1}h) - \exp(i\xi_{n-1}h)\right) \\ g^2 &= 1 + \lambda g\left[\exp(ih(\xi_{n+1}-\xi_n)) - \exp(ih(\xi_{n-1}-\xi_n))\right]. \end{aligned} \tag{19}$$

Letting $\xi_n = n\zeta$ reduces the equations further to

$$\begin{aligned} g^2 &= 1 + \lambda g[\exp(i\zeta h) - \exp(-i\zeta h)] \\ g - \frac{1}{g} &= 2i\lambda\sin(\zeta h). \end{aligned} \tag{20}$$

To assure stability, it can be shown that we require

$$\lambda \le 1. \tag{21}$$

Thus the leap-frog (2,2) time-stepping scheme is stable for values of $\lambda \le 1$. The implementation of this scheme with this restriction will not force the solution to infinity.

A couple of general remarks should be made concerning the von Neumann analysis.

- It is a general result that a scheme which is forward in time and centered in space is unstable for the one-way wave equation. This assumes a standard forward discretization, not something like Runge–Kutta.
- von Neumann analysis is rarely enough to guarantee stability, i.e. it is necessary but not sufficient.
- Many other mechanisms for unstable growth are not captured by von Neumann analysis.
- Nonlinearity usually kills the von Neumann analysis immediately. Thus a large variety of nonlinear partial differential equations are beyond the scope of a von Neumann analysis.

- Accuracy versus stability: it is always better to worry about accuracy. An unstable scheme will quickly become apparent by causing the solution to blow-up to infinity, whereas an inaccurate scheme will simply give you a wrong result without indicating a problem.

4. Comparison of Time-Stepping Schemes

A few open questions remain concerning the stability and accuracy issues of the time- and space-stepping schemes developed. In particular, it is not clear how the stability results derived in the previous section apply to other partial differential equations.

Consider, for instance, the leap-frog (2,2) method applied to the one-way wave equation

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x}\,. \tag{1}$$

The leap-frog discretization for the one-way wave equation yielded the iteration scheme

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right)\,, \tag{2}$$

where $\lambda = c\Delta t/\Delta x$ is the CFL number. The von Neumann stability analysis based upon the ansatz $u_n^{(m)} = g^m \exp(i\xi_n h)$ results in the expression

$$g - \frac{1}{g} = 2i\lambda\sin(\zeta h)\,, \tag{3}$$

where $\xi_n = n\zeta$. Thus the scheme was stable provided $\lambda \leq 1$. Note that to double the accuracy, both the time and space discretizations Δt and Δx need to be simultaneously halved.

4.1. Diffusion equation. We will again consider the leap-frog (2,2) discretization applied to the diffusion equation. The stability properties will be found to be very different from those of the one-way wave equation. The difference in the one-way wave equation and diffusion equation is an extra x derivative so that

$$\frac{\partial u}{\partial t} = c\frac{\partial^2 u}{\partial x^2}\,. \tag{4}$$

Discretizing the spatial second derivative yields

$$\frac{\partial u_n^{(m)}}{\partial t} = \frac{c}{\Delta x^2}\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right)\,. \tag{5}$$

Second-order center-differencing in time then yields

$$u_n^{(m+1)} = u_n^{(m-1)} + 2\lambda\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right)\,, \tag{6}$$

where now the CFL number is given by

$$\lambda = \frac{c\Delta t}{\Delta x^2}\,. \tag{7}$$

In this case, to double the accuracy and hold the CFL number constant requires cutting the time-step by a factor of 4. This is generally true of any partial differential equation with the highest derivative term having two derivatives. The von Neumann stability analysis based upon the ansatz $u_n^{(m)} = g^m \exp(i\xi_n h)$ results in the expression

$$\begin{aligned} g^{m+1}\exp(i\xi_n h) &= g^{m-1}\exp(i\xi_n h) + 2\lambda g^m\left(\exp(i\xi_{n+1}h) - 2\exp(i\xi_n h) + \exp(i\xi_{n-1}h)\right) \\ g - \frac{1}{g} &= 2\lambda(\exp(i\zeta h) + \exp(-i\zeta h) - 2) \\ g - \frac{1}{g} &= 2\lambda(i\sin(\zeta h) - 1)\,, \end{aligned} \tag{8}$$

where we let $\xi_n = n\zeta$. Unlike the one-way wave equation, the addition of the term $-2u_n^{(m)}$ in the discretization makes $|g(\zeta)| \geq 1$ for all values of λ. Thus the leap-frog (2,2) is unstable for all CFL numbers for the diffusion equation.

Interestingly enough, the forward Euler method which was unstable when applied to the one-way wave equation can be stable for the diffusion equation. Using an Euler discretization instead of a center-difference scheme in time yields

$$u_n^{(m+1)} = u_n^{(m)} + \lambda \left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)} \right) . \tag{9}$$

The von Neumann stability analysis based upon the ansatz $u_n^{(m)} = g^m \exp(i\xi_n h)$ results in the expression

$$g = 1 + \lambda(i \sin(\zeta h) - 1) \tag{10}$$

This gives

$$|g| = 1 - 2\lambda + \lambda^2(1 + \sin^2 \zeta h) , \tag{11}$$

so that

$$|g| \leq 1 \quad \text{for} \quad \lambda \leq \frac{1}{2} . \tag{12}$$

Thus the maximum step-size in time is given by $\Delta t = \Delta x^2/2c$. Again, it is clear that to double accuracy, the time-step must be reduced by a factor of 4.

4.2. Hyper-diffusion. Higher derivatives in x require central differencing schemes which successively add powers of Δx to the denominator. This makes for severe time-stepping restrictions in the CFL number. As a simple example, consider the fourth-order diffusion equation

$$\frac{\partial u}{\partial t} = -c \frac{\partial^4 u}{\partial x^4} . \tag{13}$$

Using a forward Euler method in time and a central difference scheme in space gives the iteration scheme

$$u_n^{(m+1)} = u_n^{(m)} - \lambda \left(u_{n+2}^{(m)} - 4u_{n+1}^{(m)} + 6u_n^{(m)} - 4u_{n-1}^{(m)} + u_{n-2}^{(m)} \right) , \tag{14}$$

where the CFL number λ is now given by

$$\lambda = \frac{c\Delta t}{\Delta x^4} . \tag{15}$$

Thus doubling the accuracy requires a drop in the step-size to $\Delta t/16$, i.e. the run time takes 32 times as long since there are twice as many spatial points and 16 times as many time-steps. This kind of behavior is often referred to as numerical stiffness.

Numerical stiffness of this sort is not a result of the central differencing scheme. Rather, it is an inherent problem with hyper-diffusion. For instance, we could consider solving the fourth-order diffusion equation (13) with fast Fourier transforms. Transforming the equation in the x-direction gives

$$\frac{\partial \widehat{u}}{\partial t} = -c(ik)^4 \widehat{u} = -ck^4 \widehat{u} . \tag{16}$$

This can then be solved with a differential equation time-stepping routine. The time-step of any of the standard time-stepping routines is based upon the size of the right-hand side of the equation. If there are $n = 128$ Fourier modes, then $k_{\max} = 64$. But note that

$$(k_{\max})^4 = (64)^4 = 16.8 \times 10^6 , \tag{17}$$

which is a very large value. The time-stepping algorithm will have to adjust to these large values which are generated strictly from the physical effect of the higher order diffusion.

There are a few key issues in dealing with numerical stiffness.

- Use a variable time-stepping routine which uses smaller steps when necessary but large steps if possible. Every built-in python differential equation solver uses an adaptive stepping routine to advance the solution.
- If numerical stiffness is a problem, then it is often useful to use an implicit scheme. This will generally allow for larger time-steps. The time-stepping algorithm *ode113* uses a predictor–corrector method which partially utilizes an implicit scheme.
- If stiffness comes from the behavior of the solution itself, i.e. you are considering a singular problem, then it is advantageous to use a solver specifically built for this stiffness. The time-stepping algorithm *ode15s* relies on Gear methods and is well suited to this type of problem.
- From a practical viewpoint, beware of the accuracy of *ode23* or *ode45* when strong nonlinearity or singular behavior is expected.

5. Operator Splitting Techniques

With either the finite difference or spectral methods, the governing partial differential equations are transformed into a system of differential equations which are advanced in time with any of the standard time-stepping schemes. A von Neumann analysis can often suggest the appropriateness of a scheme. For instance, we have the following:

A.: wave behavior: $\partial u/\partial t = \partial u/\partial x$
- forward Euler: unstable for all λ
- leap-frog (2,2): stable $\lambda \leq 1$

B.: diffusion behavior: $\partial u/\partial t = \partial^2 u/\partial x^2$
- forward Euler: stable $\lambda \leq 1/2$
- leap-frog (2,2): unstable for all λ

Thus the physical behavior of the governing equation dictates the kind of scheme which must be implemented. But what if we wanted to consider the equation

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2}? \tag{1}$$

This has elements of both diffusion and wave propagation. What time-stepping scheme is appropriate for such an equation? Certainly the Euler method seems to work well for diffusion, but destabilizes wave propagation. In contrast, leap-frog (2,2) works well for wave behavior and destabilizes diffusion.

5.1. Operator splitting. The key idea behind operator splitting is to decouple the various physical effects of the problem from each other [**47**]. Over very small time-steps, for instance, one can imagine that diffusion would essentially act independently of the wave propagation and vice versa in (1). Just as in (1), the advection–diffusion can also be thought of as decoupling. So we can consider

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega \tag{2}$$

where the bracketed terms represent the advection (wave propagation) and the right-hand side represents the diffusion. Again there is a combination of wave dynamics and diffusive spreading.

Over a very small time interval Δt, it is reasonable to conjecture that the diffusion process is independent of the advection. Thus we could split the calculation into the following two pieces:

$$\Delta t: \quad \frac{\partial \omega}{\partial t} + [\psi, \omega] = 0 \quad \text{advection only} \tag{3a}$$

$$\Delta t: \quad \frac{\partial \omega}{\partial t} = \nu \nabla^2 \omega \quad \text{diffusion only.} \tag{3b}$$

This then allows us to time-step each physical effect independently over Δt. Advantage can then be taken of an appropriate time-stepping scheme which is stable for those particular terms. For

instance, in this decoupling we could solve the advection (wave propagation) terms with a leap-frog (2,2) scheme and the diffusion terms with a forward Euler method.

5.2. Additional advantages of splitting. There can be additional advantages to the splitting scheme. To see this, we consider the nonlinear Schrödinger equation

$$i\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial^2 u}{\partial x^2} + |u|^2 u = 0 \tag{4}$$

where we split the operations so that

$$\textbf{I.}\ \ \Delta t: \quad i\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial^2 u}{\partial x^2} = 0 \tag{5a}$$

$$\textbf{II.}\ \ \Delta t: \quad i\frac{\partial u}{\partial t} + |u|^2 u = 0. \tag{5b}$$

Thus the solution will be decomposed into a linear and nonlinear part. We begin by solving the linear part **I.** Fourier transforming yields

$$i\frac{d\widehat{u}}{dt} - \frac{k^2}{2}\widehat{u} = 0 \tag{6}$$

which has the solution

$$\widehat{u} = \widehat{u}_0 \exp\left(-i\frac{k^2}{2}t\right). \tag{7}$$

So at each step, we simply need to calculate the transform of the initial condition $\widehat{u}_0$, multiply by $\exp(-ik^2t/2)$, and then invert.

The next step is to solve the nonlinear evolution **II** over a time-step Δt. This is easily done since the nonlinear evolution admits the exact solution

$$u = u_0 \exp\big(i|u_0|^2 t\big) \tag{8}$$

where u_0 is the initial condition. This part of the calculation then requires no computation, i.e. we have an exact, analytic solution. The fact that we can take advantage of this property is why the splitting algorithm is so powerful.

To summarize then, the split-step method for the nonlinear Schrödinger equation yields the following algorithm.

(1) Dispersion: $u_1 = \mathrm{FFT}^{-1}\left[\widehat{u}_0 \exp(-ik^2\Delta t/2)\right]$
(2) Nonlinearity: $u_2 = u_1 \exp(i|u_1|^2\Delta t)$
(3) Solution: $u(t+\Delta t) = u_2$.

Note the advantage that is taken of the analytic properties of the solution of each aspect of the split operators.

5.3. Symmetrized splitting. Although we will not perform an error analysis on this scheme, it is not hard to see that the error will depend heavily on the time-step Δt. Thus our scheme for solving

$$\frac{\partial u}{\partial t} = Lu + N(u) \tag{9}$$

involves the splitting

$$\textbf{I.}\ \ \Delta t: \quad \frac{\partial u}{\partial t} + Lu = 0 \tag{10a}$$

$$\textbf{II.}\ \ \Delta t: \quad \frac{\partial u}{\partial t} + N(u) = 0\,. \tag{10b}$$

To drop the error down by another $O(\Delta t)$, we use what is called a Strang splitting technique [**48**] which is based upon the Trotter product formula. This essentially involves symmetrizing the splitting algorithm so that

$$\textbf{I.}\ \frac{\Delta t}{2}: \quad \frac{\partial u}{\partial t} + Lu = 0 \tag{11a}$$

$$\textbf{II.}\ \Delta t: \quad \frac{\partial u}{\partial t} + N(u) = 0 \tag{11b}$$

$$\textbf{III.}\ \frac{\Delta t}{2}: \quad \frac{\partial u}{\partial t} + Lu = 0. \tag{11c}$$

This essentially cuts the time-step in half and allows the error to drop down an order of magnitude. It should always be used so as to keep the error in check.

6. Optimizing Computational Performance: Rules of Thumb

Computational performance is always crucial in choosing a numerical scheme. Speed, accuracy and stability all play key roles in determining the appropriate choice of method for solving. We will consider three prototypical equations:

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x} \qquad \text{one-way wave equation} \tag{1a}$$

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \qquad \text{diffusion equation} \tag{1b}$$

$$i\frac{\partial u}{\partial t} = \frac{1}{2}\frac{\partial^2 u}{\partial x^2} + |u|^2 u \qquad \text{nonlinear Schrödinger equation.} \tag{1c}$$

Periodic boundary conditions will be assumed in each case. The purpose of this section is to build a numerical routine which will solve these problems using the iteration procedures outlined in the previous two sections. Of specific interest will be the setting of the CFL number and the consequences of violating the stability criteria associated with it.

The equations are considered in one dimension such that the first and second derivative are given by

$$\frac{\partial u}{\partial x} \to \frac{1}{2\Delta x}\begin{bmatrix} 0 & 1 & 0 & \cdots & 0 & -1 \\ -1 & 0 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & 0 & -1 & 0 & 1 \\ 1 & 0 & \cdots & 0 & -1 & 0 \end{bmatrix}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} \tag{2}$$

and

$$\frac{\partial^2 u}{\partial x^2} \to \frac{1}{\Delta x^2}\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 & 1 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & 0 & 1 & -2 & 1 \\ 1 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}. \tag{3}$$

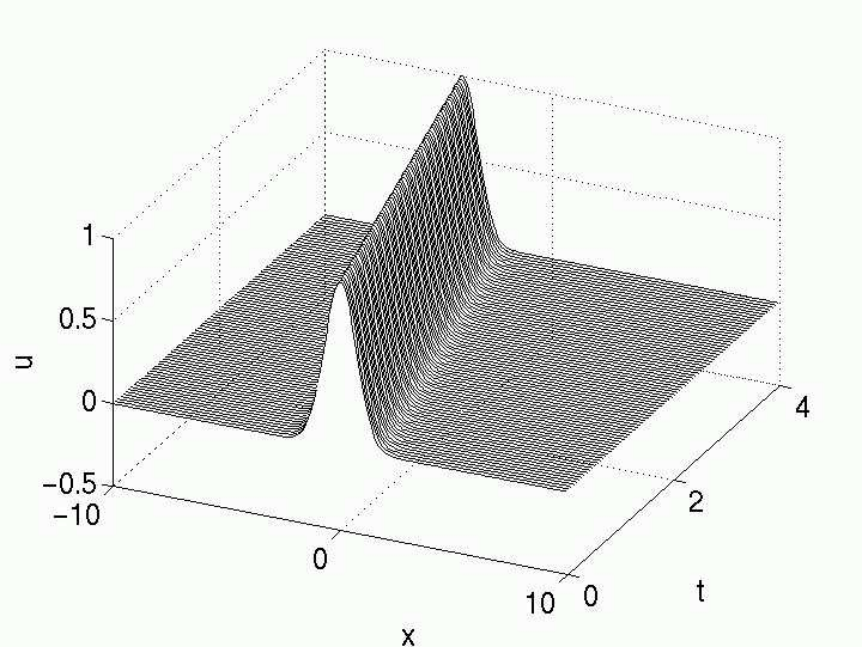

FIGURE 4. Evolution of the one-way wave equation with the leap-frog (2,2) scheme and with CFL=0.5. The stable traveling wave solution is propagated in this case to the left.

From the previous sections, we have the following discretization schemes for the one-way wave equation (1):

$$\text{Euler (unstable):} \qquad u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{4a}$$

$$\text{leap-frog (2,2) (stable for } \lambda \le 1\text{):} \qquad u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{4b}$$

where the CFL number is given by $\lambda = \Delta t/\Delta x$. Similarly for the diffusion equation (1)

$$\text{Euler (stable for } \lambda \le 1/2\text{):} \qquad u_n^{(m+1)} = u_n^{(m)} + \lambda\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right) \tag{5a}$$

$$\text{leap-frog (2,2) (unstable):} \qquad u_n^{(m+1)} = u_n^{(m-1)} + 2\lambda\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right) \tag{5b}$$

where now the CFL number is given by $\lambda = \Delta t/\Delta x^2$. The nonlinear Schrödinger equation discretizes to the following form:

$$\frac{\partial u_n^{(m)}}{\partial t} = -\frac{i}{2\Delta x^2}\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right) - i|u_n^{(m)}|^2 u_n^{(m)}\,. \tag{6}$$

We will explore Euler and leap-frog (2,2) time-stepping with this equation.

6.1. One-way wave equation. We first consider the leap-frog (2,2) scheme applied to the one-way wave equation. Figure 4 depicts the evolution of an initial Gaussian pulse. For this case, the CFL is 0.5 so that stable evolution is analytically predicted. The solution propagates to the left as expected from the exact solution. The leap-frog (2,2) scheme becomes unstable for $\lambda \ge 1$ and the system is always unstable for the Euler time-stepping scheme. Figure 5 depicts the unstable evolution of the leap-frog (2,2) scheme with CFL = 2 and the Euler time-stepping scheme. The initial conditions used are identical to that in Fig. 4. Since we have predicted that the leap-frog numerical scheme is only stable provided $\lambda < 1$, it is not surprising that the figure on the left goes

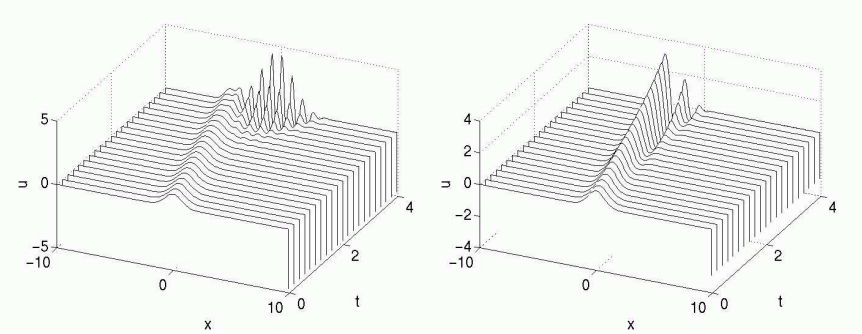

FIGURE 5. Evolution of the one-way wave equation using the leap-frog (2,2) scheme with CFL=2 (left) along with the Euler time-stepping scheme (right). The analysis predicts stable evolution of leap-frog provided the CFL$\leq$ 1. Thus the onset of numerical instability near $t \approx 3$ for the CFL=2 case is not surprising. Likewise, the Euler scheme is expected to be unstable for all CFL.

unstable. Likewise, the figure on the right shows the numerical instability generated in the Euler scheme. Note that both of these unstable evolutions develop high-frequency oscillations which eventually blow up. The python code used to generate the leap-frog and Euler iterative solutions is given by

```
Time = 4   # time domain of solution
L = 20   # domain size
n = 200   # spatial discretization points
x2 = np.linspace(-L/2, L/2, n+1)
x = x2[:n]
dx = x[1] - x[0]
dt = 0.2
CFL = dt / dx
time_steps = int(Time / dt)
t = np.arange(0, Time + dt, dt)

u0 = np.exp(-x**2) # Initial conditions
u1 = np.exp(-(x + dt)**2)
usol = np.zeros((n, time_steps + 1))
usol[:, 0] = u0
usol[:, 1] = u1

e1 = np.ones(n)  # Sparse matrix for derivative
A = diags([-e1, e1], [-1, 1], shape=(n, n)).toarray()
A[0, -1] = -1
A[-1, 0] = 1

for j in range(time_steps-1):  # Leap frog (2,2)
    u2 = u0 + CFL * A.dot(u1)  # Leap frog (2,2)
    u0 = u1
    u1 = u2
    usol[:, j + 2] = u2
```

6.2. Heat equation. In a similar fashion, we investigate the evolution of the diffusion equation when the space–time discretization is given by the leap-frog (2,2) scheme or Euler stepping. Figure 6

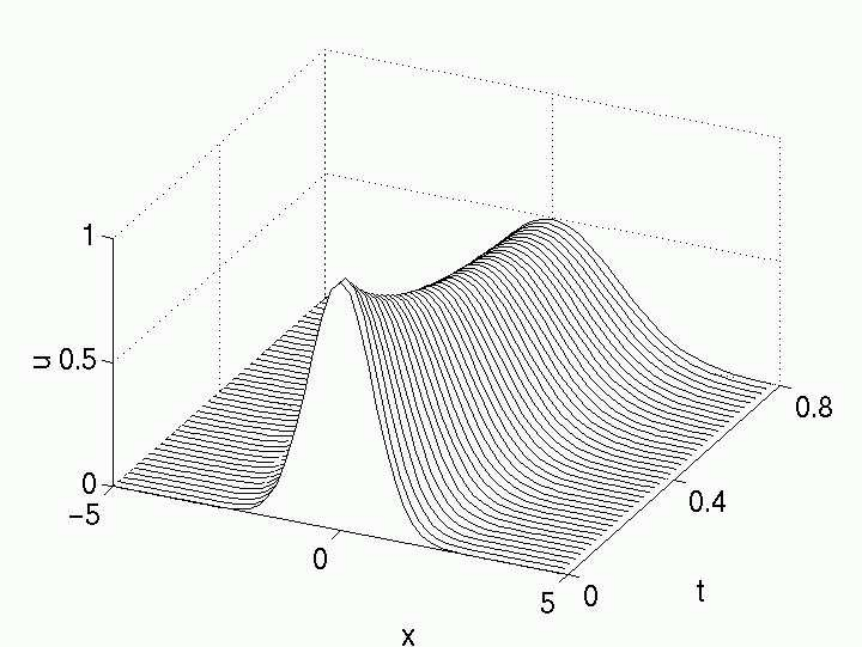

FIGURE 6. Stable evolution of the heat equation with the Euler scheme with CFL=0.5. The initial Gaussian is diffused in this case.

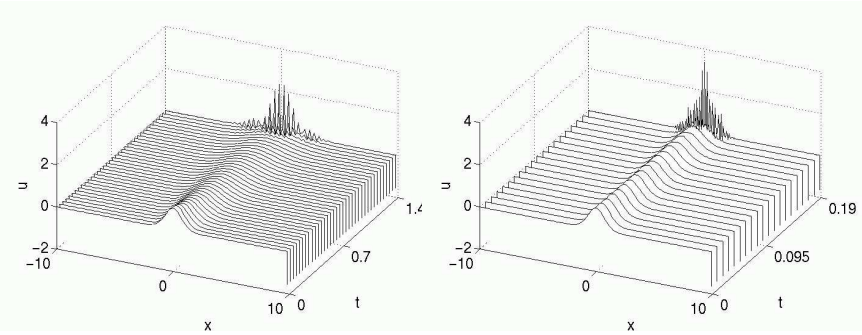

FIGURE 7. Evolution of the heat equation with the Euler time-stepping scheme (left) and leap-frog (2,2) scheme (right) with CFL=1. The analysis predicts that both these schemes are unstable. Thus the onset of numerical instability is observed.

shows the expected diffusion behavior for the stable Euler scheme ($\lambda \leq 0.5$). In contrast, Fig. 7 shows the numerical instabilities which are generated from violating the CFL constraint for the Euler scheme or using the always unstable leap-frog (2,2) scheme for the diffusion equation. The numerical code used to generate these solutions follows that given previously for the one-way wave equation. However, the sparse matrix is now given by

```
e1 = np.ones(n)
A = diags([e1, -2*e1, e1], [-1, 0, 1], shape=(n, n)).toarray()
A[0, -1] = 1
A[-1, 0] = 1
```

Further, the iterative process is now

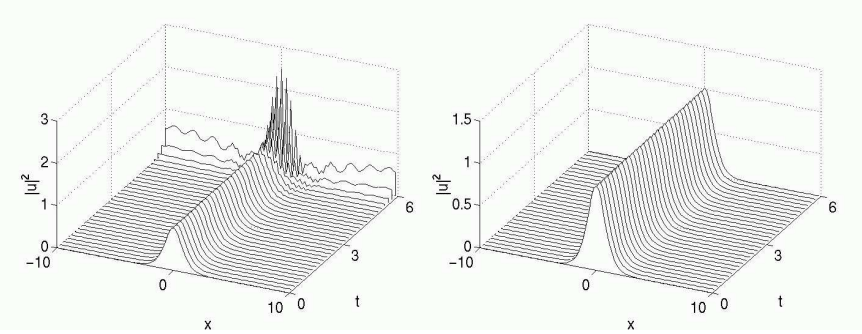

FIGURE 8. Evolution of the nonlinear Schrödinger equation with the Euler time-stepping scheme (left) and leap-frog (2,2) scheme (right) with CFL=0.05.

```
# Leap frog (2,2) iteration scheme
for j in range(time_steps - 1):
    u2 = u0 + 2 * CFL * A.dot(u1)  # Leap frog (2,2)
    u0 = u1
    u1 = u2
    usol[:, j + 2] = u2
```

where we recall that the CFL condition is now given by $\lambda = \Delta t/\Delta x^2$, i.e.

```
CFL=dt/dx/dx
```

This solves the one-dimensional heat equation with periodic boundary conditions.

6.3. Nonlinear Schrödinger equation. The nonlinear Schrödinger equation can easily be discretized by the above techniques. However, as with most nonlinear equations, it is a bit more difficult to perform a von Neumann analysis. Therefore, we explore the behavior for this system for two different discretization schemes: Euler and leap-frog (2,2). The CFL number will be the same with both schemes ($\lambda = 0.05$) and the stability will be investigated through numerical computations. Figure 8 shows the evolution of the exact one-soliton solution of the nonlinear Schrödinger equation ($u(x, 0) = \text{sech}(x)$) over six units of time. The Euler scheme is observed to lead to numerical instability whereas the leap-frog (2,2) scheme is stable. In general, the leap-frog schemes work well for wave propagation problems while Euler methods are better for problems of a diffusive nature.

The python code modifications necessary to solve the nonlinear Schrödinger equation are trivial. Specifically, the iteration scheme requires change. For the stable leap-frog scheme, the following command structure is required

```
u2 = u0 + -1j * CFL * A.dot(u1) - 1j * 2 * dt * (np.conj(u1) * u1) * u1
u0 = u1; u1 = u2
```

Note that i is automatically defined in python as $i = \sqrt{-1}$. Thus it is imperative that you do not use the variable i as a counter in your FOR loops. You will solve a very different equation if you are not careful with this definition.

7. Problems and Exercises

7.1. The Wave Equation. One of the classic equations of mathematical physics and differential equations is the *wave equation*. It has long been the prototypical equation for modeling and understanding the underlying behavior of linear wave propagation and phenomena in a myriad of fields including electrodynamics, water waves, and acoustics to name a few application areas. And although we have a number of analytic techniques available to solve the wave equation in closed form, it is a great test bed for the application of the numerical techniques developed here for solving differential and partial differential equations.

To begin, the classic derivation of the wave equation will be given. It is based entirely on Newton's Law $\mathbf{F} = m\mathbf{a}$. The derivation will be given for the displacement of an elastic string in space and time $U(x,t)$ that is pinned at two end points: $U(0,t) = 0$ and $U(L,t) = 0$. Moreover, the string will be assumed to be of a constant density (mass per unit length) and that the effects of gravity can be ignored.

Figure 9 shows the basic physical configuration associated with the vibrating string. If we consider any given location x on the string, then the string is assumed to vibrate vertically at that point, thus no translations of the string are allowed. A simple consequence of this argument, which holds for sufficiently small displacements, is that the sum of the horizontal forces are zero while the sum of the vertical are not. Mathematically, this is given as

$$\text{vertical direction:} \quad \sum \mathbf{F}_y = m\mathbf{a}_y \tag{7a}$$

$$\text{horizontal direction:} \quad \sum \mathbf{F}_x = 0 \tag{7b}$$

where the subscripts $\mathbf{F}_y$ and $\mathbf{F}_x$ are the forces in the vertical and horizontal directions respectively.

Consider then the specific spatial interval from x to $x + \Delta x$ as depicted in Fig. 9. For Δx infinitesimally small, the ideas of calculus dictate the derivation of the governing equations. In particular, the tension (forces) on the string at x and $x + \Delta x$ are given by the vectors T_- and T_+ respectively. The vertical and horizontal components of these forces can be easily computed using trigonometry rules so that

$$\text{vertical direction:} \quad T_- \sin\alpha_- - T_+ \sin\alpha_+ = -\rho\Delta x \frac{d^2U}{dt^2} \tag{8a}$$

$$\text{horizontal direction:} \quad T_+ \cos\alpha_+ - T_- \cos\alpha_- = 0 \tag{8b}$$

where ρ is the constant density of the string so that $\rho\Delta x$ is the mass of the infinitesimally small piece of string considered. Note that the second condition ensures that there is no acceleration of the string in the horizontal direction. Further, the sign is negative in front of the second derivative term in the vertical direction since the acceleration is in the negative direction as indicated in the figure.

From (8b), we have that $T_+ \cos\alpha_+ = T_- \cos\alpha_- = T$. Dividing (8a)) by the constant T yields

$$\begin{aligned}
\frac{T_- \sin\alpha_-}{T} - \frac{T_+ \sin\alpha_+}{T} &= -\frac{\rho\Delta x}{T}\frac{d^2U}{dt^2} \\
\frac{T_- \sin\alpha_-}{T_- \cos\alpha_-} - \frac{T_+ \sin\alpha_+}{T_+ \cos\alpha_+} &= -\frac{\rho\Delta x}{T}\frac{d^2U}{dt^2} \\
\tan\alpha_- - \tan\alpha_+ &= -\frac{\rho\Delta x}{T}\frac{d^2U}{dt^2}\,.
\end{aligned} \tag{9}$$

The important thing to note at this point in the derivation is that $\tan\alpha_\pm$ is simply the slope at x and $x + \Delta x$ respectively. Dividing through by a factor of $-\Delta x$ and replacing the tangent with its

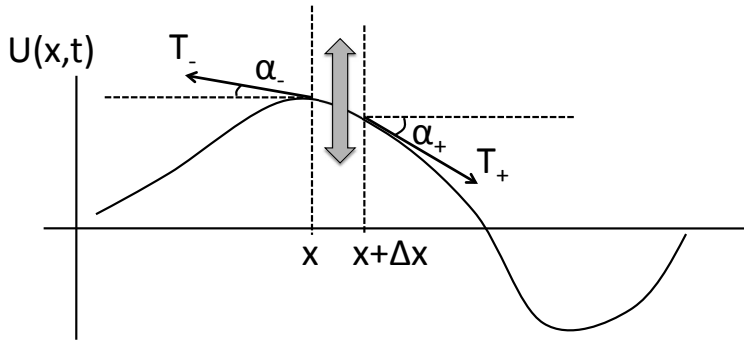

FIGURE 9. Forces associated with the deflection of a vibrating elastic string. At the points x and $x + \Delta x$, the string is subject to the tension (force) vectors $T_{\pm}$. The horizontal forces must balance to keep the string from translating whereas the vertical forces are not balanced and produce acceleration and motion of the string.

slope yields the equation

$$\frac{1}{\Delta x}\left[\frac{\partial U(x+\Delta x,t)}{\partial x} - \frac{\partial U(x,t)}{\partial x}\right] = \frac{\rho}{T}\frac{\partial^2 U(x,t)}{\partial t^2} \tag{10}$$

As $\Delta x \to 0$, the definition of derivative applies and the left hand side is the derivative of the derivative, i.e. the second derivative. This gives the wave equation

$$\frac{\partial^2 U}{\partial t^2} = c^2 \frac{\partial^2 U}{\partial x^2} \tag{11}$$

where $c^2 = T/\rho$ is a positive quantity.

Interestingly enough, the derivation can also be applied to a vibrating string where the density various in x. This gives rise to the non-constant coefficient version of the wave equation:

$$\frac{\partial^2 U}{\partial t^2} = \frac{\partial}{\partial x}\left(c^2(x)\frac{\partial U}{\partial x}\right) \tag{12}$$

This makes it more difficult to solve, but only moderately so as will be seen in the projects associated with the wave equation.

Initial Conditions: In addition to the governing equations (11) or (11) both boundary and initial conditions must be specified. Much like differential equations, the number of initial conditions to be specified depends upon the highest derivative in time. The wave equations has two derivatives in time and thus requires two initial conditions:

$$U(x,0) = f(x) \tag{13a}$$

$$\frac{\partial U(0,x)}{\partial t} = g(x) \tag{13b}$$

Thus once $f(x)$ and $g(x)$ are specified, the system can be evolved forward in time.

Boundary Conditions: A variety of boundary conditions can be applied including pinned where $U = 0$ at the boundary, no-flux where $\partial U/\partial x = 0$ at the boundary, periodic where $U(0,t) = U(L,t)$ at the boundary or some combination of pinned and no-flux. In the algorithms developed here,

consider then the following possibilities:

$$\text{pinned:} \quad U(0,t) = U(L,t) = 0 \tag{14a}$$

$$\text{no-flux:} \quad \frac{\partial U(0,t)}{\partial x} = \frac{\partial U(L,t)}{\partial x} = 0 \tag{14b}$$

$$\text{mixed:} \quad U(0,t) = 0 \text{ and } \frac{\partial U(L,t)}{\partial x} = 0 \tag{14c}$$

$$\text{periodic:} \quad U(0,t) = U(L,t) \tag{14d}$$

Each of these gives rise to a different differentiation matrix $\mathbf{A}$ for computing two derivatives in a finite difference scheme.

Projects and application:

(a) Given $f(x) = \exp(-x.^2)$ and $g(x) = 0$ as initial conditions, compute the solution numerically on the domain $x \in [0, 40]$ using a second-order accurate finite difference scheme in space with time-stepping dictated by **ode45**. Be sure to first rewrite the system as a set of first order equations. Solve the system for all four boundary types given above and be sure to simulate the system long enough to see the waves interact with the boundaries several times.
(b) Repeat experiment (a), but now implement fourth-order accurate schemes for computing the derivatives. In this case, the no-flux conditions can use second-order accurate schemes for computing the effects of the boundary on the differentiation matrix.
(c) Using periodic boundary conditions, simulate the system using the Fast Fourier Transform method.
(d) Do a convergence study of the second-order, fourth-order and spectral methods using a high-resolution simulation of the spectral method as the *true solution*. See how the solution converges to this true solution as a function of Δx. Verify that the differentiation matrices are indeed 2nd-order and 4th-order accurate. Further, determine the approximate order of accuracy of the FFT method.
(e) What happens when the density changes as a function of the space variable x? Consider $c^2(x) = \cos(10\pi x/L)$ and solve (12) with pinned boundaries and fourth-order accuracy. Note how the waves slow-down and speed up depending upon the local value of c^2.
(f) Given the Gaussian initial condition for the field, determine an appropriate initial time derivative condition $g(x)$ such that the wave travels only right or only left.
(g) For electromagnetic phenomena, the intensity of the electromagnetic field can be brought to such levels that the material properties (index of refraction) changes with the intensity of the field itself. This gives a governing wave equation:

$$\frac{\partial^2 U}{\partial t^2} = c^2 \frac{\partial^2 U}{\partial x^2} + \beta |U|^2 U \tag{15}$$

where $|U|^2$ is the intensity of the field. Solve this equation for β either positive or negative. As β is increased, note the effect it has on the wave propagation.

CHAPTER 11

Spectral Methods

Spectral methods are one of the most powerful solution techniques for ordinary and partial differential equations. The best-known example of a spectral method is the Fourier transform. We have already made use of the Fourier transform using FFT routines. Other spectral techniques exist which render a variety of problems easily tractable and often at significant computational savings.

1. Fast Fourier Transforms and Cosine/Sine Transform

From our previous chapters, we are already familiar with the Fourier transform and some of its properties. At the time, the distinct advantage to using the FFT was its computational efficiency of solving a problem in $O(N \log N)$. This section explores the underlying mathematical reasons for such performance.

One of the abstract definitions of a Fourier transform pair is given by

$$F(k) = \int_{-\infty}^{\infty} e^{-ikx} f(x) dx \tag{1a}$$

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{ikx} F(k) dk \,. \tag{1b}$$

On a practical level, the value of the Fourier transform revolves squarely around the derivative relationship

$$\widehat{f^{(n)}} = (ik)^n \widehat{f} \tag{2}$$

which results from the definition of the Fourier transform and integration by parts. Recall that we denote the Fourier transform of $f(x)$ as $\widehat{f(x)}$.

When considering a computational domain, the solution can only be found on a finite length domain. Thus the definition of the Fourier transform needs to be modified in order to account for the finite sized computational domain. Instead of expanding in terms of a continuous integral for values of wavenumber k and cosines and sines ($\exp(ikx)$), we expand in a Fourier series

$$F(k) = \sum_{n=1}^{N} f(n) \exp\left[-i\frac{2\pi(k-1)}{N}(n-1)\right] \quad 1 \le k \le N \tag{3a}$$

$$f(n) = \frac{1}{N} \sum_{k=1}^{N} F(k) \exp\left[i\frac{2\pi(k-1)}{N}(n-1)\right] \quad 1 \le n \le N. \tag{3b}$$

Thus the Fourier transform is nothing but an expansion in a basis of cosine and sine functions. If we define the fundamental oscillatory piece as

$$w^{nk} = \exp\left(\frac{2i\pi(k-1)(n-1)}{N}\right) = \cos\left(\frac{2\pi(k-1)(n-1)}{N}\right) + i \sin\left(\frac{2\pi(k-1)(n-1)}{N}\right), \tag{4}$$

then the Fourier transform results in the expression

$$F_n = \sum_{k=0}^{N-1} w^{nk} f_k, \qquad 0 \le n \le N-1. \tag{5}$$

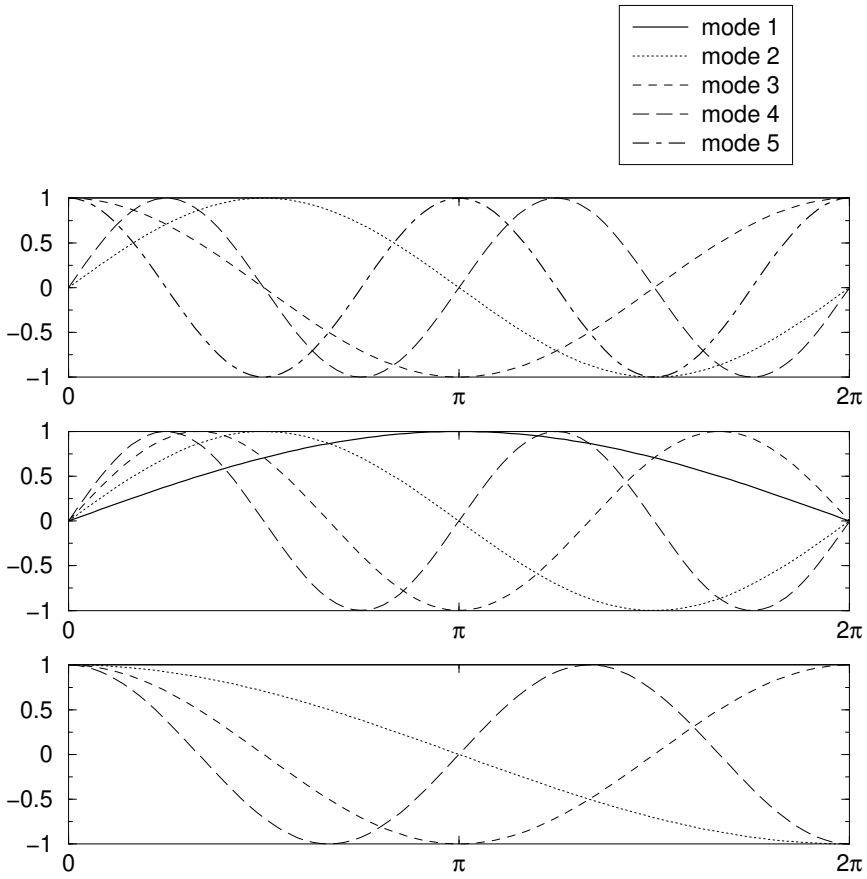

FIGURE 1. Basis functions used for a Fourier mode expansion (top), a sine expansion (middle), and a cosine expansion (bottom).

Thus the calculation of the Fourier transform involves a double sum and an $O(N^2)$ operation. Thus, at first, it would appear that the Fourier transform method is the same operation count as LU decomposition. The basis functions used for the Fourier transform, sine transform and cosine transform are depicted in Fig. 1. The process of solving a differential or partial differential equation involves evaluating the coefficient of each of the modes. Note that this expansion, unlike the finite difference method, is a *global* expansion in that every basis function is evaluated on the entire domain.

The Fourier, sine and cosine transforms behave very differently at the boundaries. Specifically, the Fourier transform assumes periodic boundary conditions whereas the sine and cosine transforms assume pinned and no-flux boundaries, respectively. The cosine and sine transform are often chosen for their boundary properties. Thus for a given problem, an evaluation must be made of the type of transform to be used based upon the boundary conditions needing to be satisfied. Table 1 illustrates the three different expansions and their associated boundary conditions. The appropriate python command is also given.

1.1. Fast Fourier transforms: Cooley–Tukey algorithm. To see how the FFT gets around the computational restriction of an $O(N^2)$ scheme, we consider the Cooley–Tukey algorithm which drops the computations to $O(N \log N)$. To consider the algorithm, we begin with the $N \times N$ matrix $\mathbf{F}_N$ whose components are given by

$$(F_N)_{jk} = w_n^{jk} = \exp(i2\pi jk/N). \tag{6}$$

command	expansion	boundary conditions
fft	$F_k = \sum_{j=0}^{2N-1} f_j \exp(i\pi jk/N)$	periodic: $f(0) = f(L)$
dst	$F_k = \sum_{j=1}^{N-1} f_j \sin(\pi jk/N)$	pinned: $f(0) = f(L) = 0$
dct	$F_k = \sum_{j=0}^{N-2} f_j \cos(\pi jk/2N)$	no-flux: $f'(0) = f'(L) = 0$

TABLE 1. MATLAB functions for Fourier, sine, and cosine transforms and their associated boundary conditions. To invert the expansions, the MATLAB commands are *ifft, idst,* and *idct* repectively.

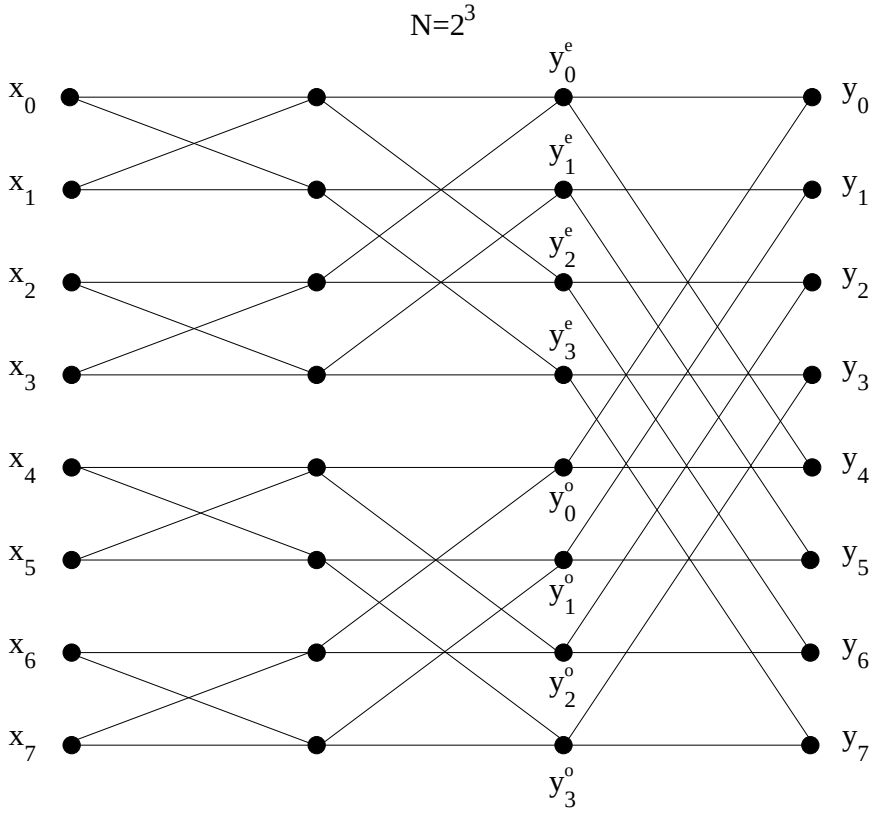

FIGURE 2. Graphical description of the Fast Fourier Transform process which systematically continues to factor the problem in two. This process allows the FFT routine to drop to $O(N \log N)$ operations.

The coefficients of the matrix are points on the unit circle since $|w_n^{jk}| = 1$. They are also the basis functions for the Fourier transform.

The FFT starts with the following trivial observation

$$w_{2n}^2 = w_n, \tag{7}$$

which is easy to show since $w_n = \exp(i2\pi/n)$ and

$$w_{2n}^2 = \exp(i2\pi/(2n)) \exp(i2\pi/(2n)) = \exp(i2\pi/n) = w_n. \tag{8}$$

The consequences of this simple relationship are enormous and are at the core of the success of the FFT algorithm. Essentially, the FFT is a matrix operation

$$\mathbf{y} = \mathbf{F}_N \mathbf{x} \tag{9}$$

which can now be split into two separate operations. Thus defining

$$\mathbf{x}^e = \begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ \vdots \\ x_{N-2} \end{pmatrix} \quad \text{and} \quad \mathbf{x}^o = \begin{pmatrix} x_1 \\ x_3 \\ x_5 \\ \vdots \\ x_{N-1} \end{pmatrix} \tag{10}$$

which are both vectors of length $M = N/2$, we can form the two $M \times M$ systems

$$\mathbf{y}^e = \mathbf{F}_M \mathbf{x}^e \quad \text{and} \quad \mathbf{y}^o = \mathbf{F}_M \mathbf{x}^o \tag{11}$$

for the even coefficient terms $\mathbf{x}^e, \mathbf{y}^e$ and odd coefficient terms $\mathbf{x}^o, \mathbf{y}^o$. Thus the computation size goes from $O(N^2)$ to $O(2M^2) = O(N^2/2)$. However, we must be able to reconstruct the original $\mathbf{y}$ from the smaller system of $\mathbf{y}^e$ and $\mathbf{y}^o$. And indeed we can reconstruct the original $\mathbf{y}$. In particular, it can be shown that component by component

$$y_n = y_n^e + w_N^n y_n^o \quad n = 0, 1, 2, \ldots, M-1 \tag{12a}$$

$$y_{n+M} = y_n^e - w_N^n y_n^o \quad n = 0, 1, 2, \ldots, M-1\,. \tag{12b}$$

This is where the shift occurs in the FFT routine which maps the domain $x \in [0, L]$ to $[-L, 0]$ and $x \in [-L, 0]$ to $[0, L]$. The command *fftshift* undoes this shift. Details of this construction can be found elsewhere [**44, 48**].

There is no reason to stop the splitting process at this point. In fact, provided we choose the size of our domain and matrix $\mathbf{F}_N$ so that N is a power of 2, then we can continue to split the system until we have a simple algebraic, i.e. a 1×1 system, solution to perform. The process is illustrated graphically in Fig. 2 where the switching and factorization are illustrated. Once the final level is reached, the algebraic expression is solved and the process is reversed. This factorization process renders the FFT scheme $O(N \log N)$.

2. Chebychev Polynomials and Transform

The fast Fourier transform is only one of many possible expansion bases, i.e. there is nothing special about expanding in cosines and sines. Of course, the FFT expansion does have the unusual property of factorization which drops it to an $O(N \log N)$ scheme. Regardless, there are a myriad of other expansion bases which can be considered. The primary motivation for considering other expansions is based upon the specifics of the given governing equations and its physical boundaries and constraints. Special functions are often prime candidates for use as expansion bases. The following are some important examples

- Bessel functions: radial, 2D problems
- Legendre polynomials: 3D Laplaces equation
- Hermite–Gauss polynomials: Schrödinger with harmonic potential
- Spherical harmonics: radial, 3D problems
- Chebychev polynomials: bounded 1D domains.

The two key properties required to use any expansion basis successfully are its orthogonality properties and the calculation of the norm. Regardless, all the above expansions appear to require $O(N^2)$ calculations.

2.1. Chebychev polynomials.

Although not often encountered in mathematics courses, the Chebychev polynomial is an important special function from a computational viewpoint. The reason for its prominence in the computational setting will become apparent momentarily. For the present, we note that the Chebychev polynomials are solutions to the differential equation

$$\sqrt{1-x^2}\frac{d}{dx}\left(\sqrt{1-x^2}\frac{dT_n}{dx}\right) + n^2 T_n = 0 \quad x \in [-1, 1]. \tag{1}$$

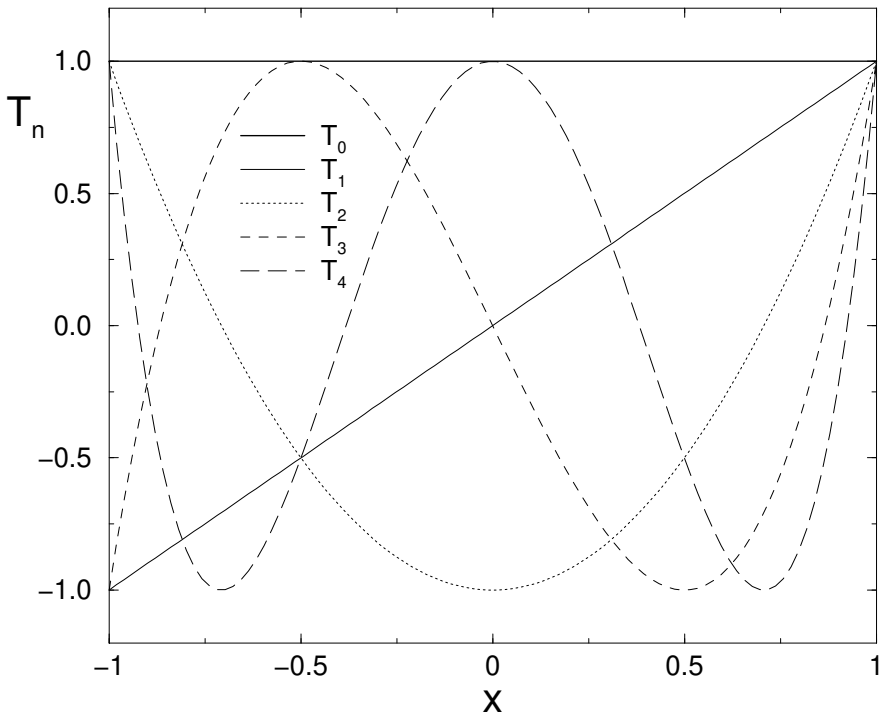

FIGURE 3. The first five Chebychev polynomials over the the interval of definition $x \in [-1, 1]$.

This is a self-adjoint Sturm–Lioville problem. Thus the following properties are known:

(1) Eigenvalues are real: $\lambda_n = n^2$
(2) Eigenfunctions are real: $T_n(x)$
(3) Eigenfunctions are orthogonal:

$$\int_{-1}^{1} (1-x^2)^{-1/2} T_n(x) T_m(x) dx = \frac{\pi}{2} c_n \delta_{nm} \tag{2}$$

where $c_0 = 2, c_n = 1 (n > 0)$ and δ_{nm} is the delta function
(4) Eigenfunctions form a complete basis.

Each Chebychev polynomial (of degree n) is defined by

$$T_n(\cos\theta) = \cos n\theta. \tag{3}$$

Thus we find

$$T_0(x) = 1 \tag{4a}$$
$$T_1(x) = x \tag{4b}$$
$$T_2(x) = 2x^2 - 1 \tag{4c}$$
$$T_3(x) = 4x^3 - 3x \tag{4d}$$
$$T_4(x) = 8x^4 - 8x^2 + 1. \tag{4e}$$

The behavior of the first five Chebychev polynomials is illustrated in Fig. 3.

It is appropriate to ask why the Chebychev polynomials, of all the special functions listed, are of such computational interest. Especially given that the equation which the $T_n(x)$ satisfy, and their functional form shown in Fig. 3, appear to be no better than Bessel, Hermite–Gauss, or any other special function. The distinction with the Chebychev polynomials is that you can transform

them so that use can be made of the $O(N \log N)$ discrete cosine transform. This effectively renders the Chebychev expansion scheme an $O(N \log N)$ transformation. Specifically, we transform from the interval $x \in [-1, 1]$ by letting

$$x = \cos\theta \qquad \theta \in [0, \pi]. \tag{5}$$

Thus when considering a function $f(x)$, we have $f(\cos\theta) = g(\theta)$. Under differentiation we find

$$\frac{dg}{d\theta} = -f' \cdot \sin\theta. \tag{6}$$

Thus $dg/d\theta = 0$ at $\theta = 0, \pi$, i.e. no-flux boundary conditions are satisfied. This allows us to use the *dct* (discrete cosine transform) to solve a given problem in the new transformed variables.

The Chebychev expansion is thus given by

$$f(x) = \sum_{k=0}^{\infty} a_k T_k(x) \tag{7}$$

where the coefficients a_k are determined from orthogonality and inner products to be

$$a_k = \int_{-1}^{1} \frac{1}{\sqrt{1-x^2}} f(x) T_k(x) dx. \tag{8}$$

It is these coefficients which are calculated in $O(N \log N)$ time. Some of the properties of the Chebychev polynomials are as follows:

- $T_{n+1} = 2xT_n(x) - T_{n-1}(x)$
- $|T_n(x)| \leq 1, \ |T_n'(x)| \leq n^2$
- $T_n(\pm 1) = (\pm 1)^n$
- $d^p/dx^p(T_n(\pm 1)) = (\pm 1)^{n+p} \prod_{k=0}^{p-1}(n^2 - k^2)/(2k+1)$
- If n is even (odd), $T_n(x)$ is even (odd).

There are a couple of critical practical issues which must be considered when using the Chebychev scheme. Specifically the grid generation and spatial resolution are a little more difficult to handle. In using the discrete cosine transform on the variable $\theta \in [0, \pi]$, we recall that our original variable is actually $x = \cos\theta$ where $x \in [-1, 1]$. Thus the discretization of the θ variable leads to

$$x_m = \cos\left(\frac{(2m-1)\pi}{2n}\right) \qquad m = 1, 2, \ldots, n. \tag{9}$$

Thus although the grid points are uniformly spaced in θ, the grid points are clustered in the original x variable. Specifically, there is a clustering of grid points at the boundaries. The Chebychev scheme then automatically has higher resolution at the boundaries of the computational domain. The clustering of the grid points at the boundary is illustrated in Fig. 4. So, as the resolution is increased, it is important to be aware that the resolution increase is not uniform across the computational domain.

2.2. Solving differential equations. As with any other solution method, a solution scheme must have an efficient way of relating derivatives to the function itself. For the FFT method, there was a very convenient relationship between the transform of a function and the transform of its derivatives. Although not as transparent as the FFT method, we can also relate the Chebychev transform derivatives to the Chebychev transform itself.

Defining L to be a linear operator so that

$$Lf(x) = \sum_{n=0}^{\infty} b_n T_n(x), \tag{10}$$

then with $f(x) = \sum_{n=0}^{\infty} a_n T_n(x)$ we find

- $Lf = f'(x) : c_n b_n = 2 \sum_{p=n+1(p+nodd)}^{\infty} p a_p$

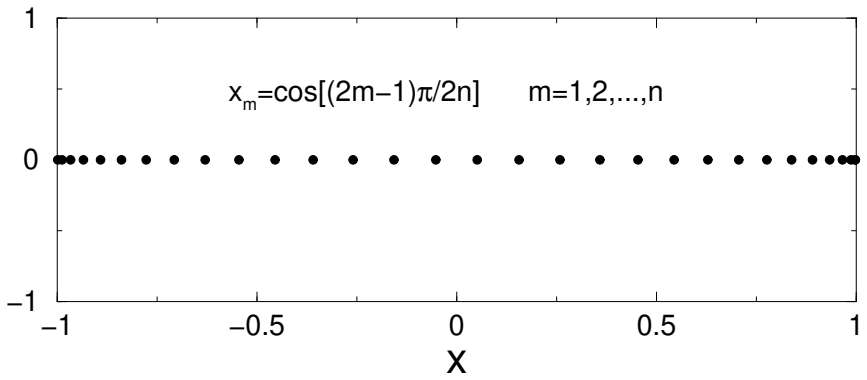

FIGURE 4. Clustered grid generation for $n = 30$ points using the Chebychev polynomials. Note that although the points are uniformly spaced in θ, they are clustered due to the fact that $x_m = \cos[(2m-1)\pi/2n]$ where $m = 1, 2, ..., n$.

- $Lf = xf(x) : b_n = (c_{n-1}a_{n-1} + a_{n+1})/2$
- $Lf = x^2 f(x) : b_n = (c_{n-2}a_{n-2} + (c_n + c_{n-1})a_n + a_{n+2})/4$

where $c_0 = 2, c_n = 0(n < 0), c_n = 1(n > 0), d_n = 1(n \geq 0)$, and $d_n = 0(n < 0)$.

3. Spectral Method Implementation

In this section, we develop an algorithm which implements a spectral method solution technique. We begin by considering the general partial differential equation

$$\frac{\partial u}{\partial t} = Lu + N(u) \tag{1}$$

where L is a linear, constant coefficient operator, i.e. it can take the form $L = ad^2/dx^2 + bd/dx + c$ where a, b, and c are constants. The second term $N(u)$ includes the nonlinear and nonconstant coefficient terms. An example of this would be $N(u) = u^3 + f(x)u + g(x)d^2u/dx^2$.

By applying a Fourier transform, the equations reduce to the system of differential equations

$$\frac{d\,\widehat{u}}{dt} = \alpha(k)\widehat{u} + \widehat{N(u)}\,. \tag{2}$$

This system can be stepped forward in time with any of the standard time-stepping techniques. Typically *ode45* or *ode23* is a good first attempt.

The parameter $\alpha(k)$ arises from the linear operator Lu and is easily determined from Fourier transforming. Specifically, if we consider the linear operator

$$Lu = a\frac{d^2u}{dx^2} + b\frac{du}{dx} + cu \tag{3}$$

then upon transforming this becomes

$$\begin{aligned}(ik)^2 a\widehat{u} + b(ik)\widehat{u} + c\widehat{u} \\ &= (-k^2 a + ibk + c)\widehat{u} \\ &= \alpha(k)\widehat{u}.\end{aligned} \tag{4}$$

The parameter $\alpha(k)$ therefore takes into account all constant coefficient, linear differentiation terms.

The nonlinear terms are a bit more difficult to handle, but they are still relatively easy. Consider the following examples

(1) $f(x)du/dx$
- determine $du/dx \to \widehat{du/dx} = ik\widehat{u}, du/dx = FFT^{-1}(ik\widehat{u})$
- multiply by $f(x) \to f(x)du/dx$

- Fourier transform $FFT(\, f(x)du/dx)$.

(2) u^3
- Fourier transform $FFT(u^3)$.

(3) $u^3 d^2u/dx^2$
- determine $d^2u/dx^2 \to \widehat{d^2u/dx^2} = (ik)^2\widehat{u}, d^2u/dx^2 = FFT^{-1}(-k^2\widehat{u})$
- multiply by $u^3 \to u^3 d^2u/dx^2$
- Fourier transform $FFT(u^3 d^2u/dx^2)$.

These examples give an outline of how the nonlinear, nonconstant coefficient schemes would work in practice.

To illustrate the implementation of these ideas, we solve the advection–diffusion equations spectrally. Thus far the equations have been considered largely with finite difference techniques. However, the python codes presented here solve the equations using the FFT for both the streamfunction and vorticity evolution. We begin by initializing the appropriate numerical parameters

```
tspan = np.arange(0, 12, 2)
nu = 0.001
Lx, Ly = 20, 20
nx, ny = 64, 64
N = nx * ny

# Define spatial domain and initial conditions
x2 = np.linspace(-Lx/2, Lx/2, nx + 1)
x = x2[:nx]
y2 = np.linspace(-Ly/2, Ly/2, ny + 1)
y = y2[:ny]
```

Thus the computational domain is $x \in [-10, 10]$ and $y \in [-10, 10]$. The diffusion parameter is chosen to be $\nu = 0.001$. The initial conditions are then defined as a stretched Gaussian

```
# initial conditions
X, Y = np.meshgrid(x, y)
w = 1 * np.exp(-0.25 * X**2 - Y**2) + 1j * np.zeros((nx, ny))
```

The next step is to define the spectral k values in both the x- and y-directions. This allows us to solve for the streamfunction (8) spectrally. Once the streamfunction is determined, the vorticity can be stepped forward in time.

```
# Define spectral k values
kx = (2 * np.pi / Lx) * np.concatenate((np.arange(0, nx/2), np.arange(-nx/2, 0)))
kx[0] = 1e-6
ky = (2 * np.pi / Ly) * np.concatenate((np.arange(0, ny/2), np.arange(-ny/2, 0)))
ky[0] = 1e-6
KX, KY = np.meshgrid(kx, ky)
K = KX**2 + KY**2

# Solve the ODE and plot the results
wt0= np.hstack([np.real(fft2(w).reshape(N)),np.imag(fft2(w).reshape(N))])
wtsol = odeint(spc_rhs, wt0, tspan, args=(nx, ny, N, KX, KY, K, nu))
```

The right-hand side of the system of differential equations which results from Fourier transforming is contained within the function *spc_rhs.m*. The outline of this routine is provided below. Note that the matrix components are reshaped into a vector and a large system of differential equations is solved.

```
# Define the ODE system
def spc_rhs(wt2, t, nx, ny, N, KX, KY, K, nu):
    wtc = wt2[0:N] + 1j*wt2[N:]
    wt = wtc.reshape((nx, ny))
    psit = -wt / K
    psix = np.real(ifft2(1j * KX * psit))
    psiy = np.real(ifft2(1j * KY * psit))
    wx = np.real(ifft2(1j * KX * wt))
    wy = np.real(ifft2(1j * KY * wt))
    rhs = (-nu * K * wt + fft2(wx * psiy - wy * psix)).reshape(N)
    return np.hstack([np.real(rhs),np.imag(rhs)])
```

The code will quickly and efficiently solve the advection–diffusion equations in two dimensions. Figure 3 demonstrates the evolution of the initial stretched Gaussian vortex over $t \in [0, 8]$.

4. Pseudo-Spectral Techniques with Filtering

The decomposition of the solution into Fourier mode components does not always lead to high-performance computing. Specifically when some form of numerical stiffness is present, computational performance can suffer dramatically. However, there are methods available which can effectively eliminate some of the numerical stiffness by making use of analytic insight into the problem.

We consider again the example of hyper-diffusion for which the governing equations are

$$\frac{\partial u}{\partial t} = -\frac{\partial^4 u}{\partial x^4}. \tag{1}$$

In the Fourier domain, this becomes

$$\frac{d\,\widehat{u}}{dt} = -(ik)^4\,\widehat{u} = -k^4\,\widehat{u} \tag{2}$$

which has the solution

$$\widehat{u} = \widehat{u}_0 \exp(-k^4 t). \tag{3}$$

However, a time-stepping scheme would obviously solve (2) directly without making use of the analytic solution.

For $n = 128$ Fourier modes, the wavenumber k ranges in values from $k \in [-64, 64]$. Thus the largest value of k for the hyper-diffusion equation is

$$k^4_{\max} = (64)^4 = 16,777,216, \tag{4}$$

or roughly $k^4_{\max} = 1.7 \times 10^7$. For $n = 1024$ Fourier modes, this is

$$k^4_{\max} = 6.8 \times 10^{10}. \tag{5}$$

Thus even if our solution is small at the high wavenumbers, this can create problems. For instance, if the solution at high wavenumbers is $O(10^{-6})$, then

$$\frac{d\,\widehat{u}}{dt} = -(10^{10})(10^{-6}) = -10^4. \tag{6}$$

Such large numbers on the right-hand side of the equations force time-stepping schemes like *ode23* and *ode45* to take much smaller time-steps in order to maintain tolerance constraints. This is a form of numerical stiffness which can be circumvented.

4.1. Filtered pseudo-spectral. There are a couple of ways to help get around the above mentioned numerical stiffness. We again consider the very general partial differential equation

$$\frac{\partial u}{\partial t} = Lu + N(u) \tag{7}$$

where as before L is a linear, constant coefficient operator, i.e. it can take the form $L = ad^2/dx^2 + bd/dx + c$ where a, b, and c are constants. The second term $N(u)$ includes the nonlinear and nonconstant coefficient terms. An example of this would be $N(u) = u^3 + f(x)u + g(x)d^2u/dx^2$.

Previously, we transformed the equation by Fourier transforming and constructing a system of differential equations. However, there is a better way to handle this general equation and remove some numerical stiffness at the same time. To introduce the technique, we consider the first-order differential equation

$$\frac{dy}{dt} + p(t)y = g(t). \tag{8}$$

We can multiply by the integrating factor $\mu(t)$ so that

$$\mu\frac{dy}{dt} + \mu p(t)y = \mu g(t). \tag{9}$$

We note from the chain rule that $(\mu y)' = \mu' y + \mu y'$ where the prime denotes differentiation with respect to t. Thus the differential equation becomes

$$\frac{d}{dt}(\mu y) = \mu g(t), \tag{10}$$

provided $d\mu/dt = \mu p(t)$. The solution then becomes

$$y = \frac{1}{\mu}\left[\int \mu(t)g(t)dt + c\right] \qquad \mu(t) = \exp\left(\int p(t)dt\right) \tag{11}$$

which is the standard integrating factor method of a first course on differential equations.

We use the key ideas of the integrating factor method to help solve the general partial differential equation and remove stiffness. Fourier transforming (7) results in the spectral system of differential equations

$$\frac{d\,\widehat{u}}{dt} = \alpha(k)\widehat{u} + \widehat{N(u)}\,. \tag{12}$$

This can be rewritten

$$\frac{d\,\widehat{u}}{dt} - \alpha(k)\widehat{u} = \widehat{N(u)}\,. \tag{13}$$

Multiplying by $\exp(-\alpha(k)t)$ gives

$$\begin{aligned}
&\frac{d\,\widehat{u}}{dt}\exp(-\alpha(k)t) - \alpha(k)\widehat{u}\exp(-\alpha(k)t) = \exp(-\alpha(k)t)\widehat{N(u)} \\
&\frac{d}{dt}[\widehat{u}\exp(-\alpha(k)t)] = \exp(-\alpha(k)t)\widehat{N(u)}.
\end{aligned}$$

By defining $\widehat{v} = \widehat{u}\exp(-\alpha(k)t)$, the system of equations reduces to

$$\frac{d\,\widehat{v}}{dt} = \exp(-\alpha(k)t)\widehat{N(u)} \tag{14a}$$

$$\widehat{u} = \widehat{v}\exp(\alpha(k)t). \tag{14b}$$

Thus the linear, constant coefficient terms are solved for explicitly, and the numerical stiffness associated with the Lu term is effectively eliminated.

4.2. Example: Fisher–Kolmogorov equation. As an example of the implementation of the filtered pseudo-spectral scheme, we consider the Fisher–Kolmogorov equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + u^3 + cu. \tag{15}$$

Fourier transforming the equation yields

$$\frac{d\,\widehat{u}}{dt} = (ik)^2\widehat{u} + \widehat{u^3} + c\widehat{u} \tag{16}$$

which can be rewritten

$$\frac{d\,\widehat{u}}{dt} + (k^2 - c)\widehat{u} = \widehat{u^3}. \tag{17}$$

Thus $\alpha(k) = c - k^2$ and

$$\frac{d\,\widehat{v}}{dt} = \exp[-(c-k^2)t]\widehat{u^3} \tag{18a}$$

$$\widehat{u} = \widehat{v}\exp[(c-k^2)t]. \tag{18b}$$

It should be noted that the solutions u must continually be updating the value of u^3 which is being transformed in the right-hand side of the equations. Thus after every time-step Δt, the new u should be used to evaluate $\widehat{u^3}$.

There are a few practical issues that should be considered when implementing this technique:

- When solving the general equation

$$\frac{\partial u}{\partial t} = Lu + N(u) \tag{19}$$

  it is important to only step forward Δt in time with

$$\frac{d\,\widehat{v}}{dt} = \exp(-\alpha(k)t)\widehat{N(u)}$$
$$\widehat{u} = \widehat{v}\exp(\alpha(k)t)$$

  before the nonlinear term $\widehat{N(u)}$ is updated.
- The computational saving for this method generally does not manifest itself unless there are more than two spatial derivatives in the highest derivative of Lu.
- Care must be taken in handling your time-step Δt in python since it uses adaptive time-stepping.

4.3. Comparison of Spectral and Finite Difference Methods. Before closing the discussion on the spectral method, we investigate the advantages and disadvantages associated with the spectral and finite difference schemes. Of particular interest are the issues of accuracy, implementation, computational efficiency and boundary conditions. The strengths and weaknesses of the schemes will be discussed.

A.: Accuracy

- **Finite Differences:** Accuracy is determined by the Δx and Δy chosen in the discretization. Accuracy is fairly easy to compute and generally much worse than spectral methods.
- **Spectral Method:** Spectral methods rely on a global expansion and are often called *spectrally accurate*. In particular, spectral methods have *infinite order accuracy*. Although the details of what this means will not be discussed here, it will suffice to say that they are generally of much higher accuracy than finite differences.

B.: Implementation

- **Finite Differences:** The greatest difficulty in implementing the finite difference schemes is generating the correct sparse matrices. Many of these matrices are very complicated with higher order schemes and in higher dimensions. Further, when solving the resulting system $\mathbf{Ax} = \mathbf{b}$, it should always be checked whether $\det \mathbf{A} = 0$. The MATLAB command *cond(A)* checks the condition number of the matrix. If *cond(A)*$> 10^{15}$, then $\det \mathbf{A} \approx 0$ and steps must be taken in order to solve the problem correctly.
- **Spectral Method:** The difficulty with using FFTs is the continual switching between the time or space domain and the spectral domain. Thus it is imperative to know exactly when and where in the algorithm this switching must take place.

C.: Computational Efficiency

- **Finite Differences:** The computational time for finite differences is determined by the size of the matrices and vectors in solving $\mathbf{Ax} = \mathbf{b}$. Generally speaking, you can guarantee $O(N^2)$ efficiency by using LU decomposition. At times, iterative schemes can lower the operation count, but there are no guarantees about this.
- **Spectral Method:** The FFT algorithm is an $O(N \log N)$ operation. Thus it is almost always guaranteed to be faster than the finite difference solution method which is $O(N^2)$. Recall that this efficiency improvement comes with an increased accuracy as well. Thus making the spectral highly advantageous when implemented.

D.: Boundary Conditions

- **Finite Differences:** Of the above categories, spectral methods are generally better in every regard. However, finite differences are clearly superior when considering boundary conditions. Implementing the generic boundary conditions

$$\alpha u(L) + \beta \frac{du(L)}{dx} = \gamma \tag{20}$$

  is easily done in the finite difference framework. Also, more complicated computational domains may be considered. Generally any computational domain which can be constructed of rectangles is easily handled by finite difference methods.
- **Spectral Method:** Boundary conditions are the critical limitation on using the FFT method. Specifically, only periodic boundary conditions can be considered. The use of the discrete sine or cosine transform allows for the consideration of pinned or no-flux boundary conditions, but only odd or even solutions are admitted respectively.

5. Boundary Conditions and the Chebychev Transform

Thus far, we have focused on the use of the FFT as the primary tool for spectral methods and their implementation. However, many problems do not in fact have periodic boundary conditions and the accuracy and speed of the FFT is rendered useless. The Chebychev polynomials are a set of mathematical functions which still allow for the construction of a spectral method which is both fast and accurate. The underlying concept is that Chebychev polynomials can be related to sines and cosines, and therefore they can be connected to the FFT routine.

Before constructing the details of the Chebychev method, we begin by considering three methods for handling nonperiodic boundary conditions.

5.1. Method 1: Periodic extension with FFTs. Since the FFT only handles periodic boundary conditions, we can periodically extend a general function $f(x)$ in order to make the function itself now periodic. The FFT routine can now be used. However, the periodic extension will in general generate discontinuities in the periodically extended function. The discontinuities give rise to Gibb's phenomenon: strong oscillations and errors are accumulated at the jump locations. This will greatly affect the accuracy of the scheme. So although spectral accuracy and speed is

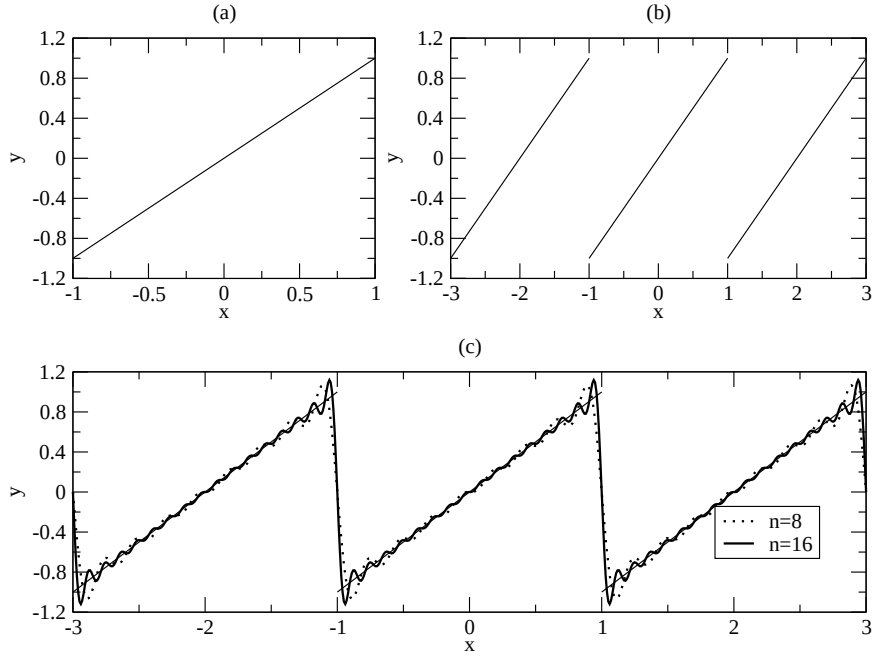

FIGURE 5. The function $y(x) = x$ for $x \in [-1, 1]$ (a) and its periodic extension (b). The FFT approximation is shown in (c) for $n = 8$ Fourier modes and $n = 16$ Fourier modes. Note the Gibb's oscillations.

retained away from discontinuities, the errors and the jumps will begin to propagate out to the rest of the computational domain.

To see this phenomenon, Fig. 5(a) considers a function which is periodically extended as shown in Fig. 5(b). The FFT approximation to this function is shown in Fig. 5(c) where the Gibb's oscillations are clearly seen at the jump locations. The oscillations clearly impact the usefulness of this periodic extension technique.

5.2. Method 2: Polynomial approximation with equi-spaced points. In moving away from an FFT basis expansion method, we can consider the most straightforward method available: polynomial approximation. Thus we simply discretize the given function $f(x)$ with $N + 1$ equally spaced points and fit an Nth degree polynomial through it. This amounts to letting

$$f(x) \approx a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^N \tag{1}$$

where the coefficients a_n are determined by an $(N+1) \times (N+1)$ system of equations. This method easily satisfies the prescribed nonperiodic boundary conditions. Further, differentiation of such an approximation is trivial. In this case, however, Runge phenomena (polynomial oscillations) generally occur. This is because a polynomial of degree n generally has $N - 1$ combined maxima and minima.

The Runge phenomena can easily be illustrated with the simple example function $f(x) = (1 + 16x^2)^{-1}$. Figure 6(a)–(b) illustrates the large oscillations which develop near the boundaries due to Runge phenomena for $N = 12$ and $N = 24$. As with the Gibb's oscillations generated by FFTs, the Runge oscillations render a simple polynomial approximation based upon equally spaced points useless.

5.3. Method 3: Polynomial approximation with clustered points. There exists a modification to the straightforward polynomial approximation given above. Specifically, this modification constructs a polynomial approximation on a clustered grid as opposed to the equal spacing

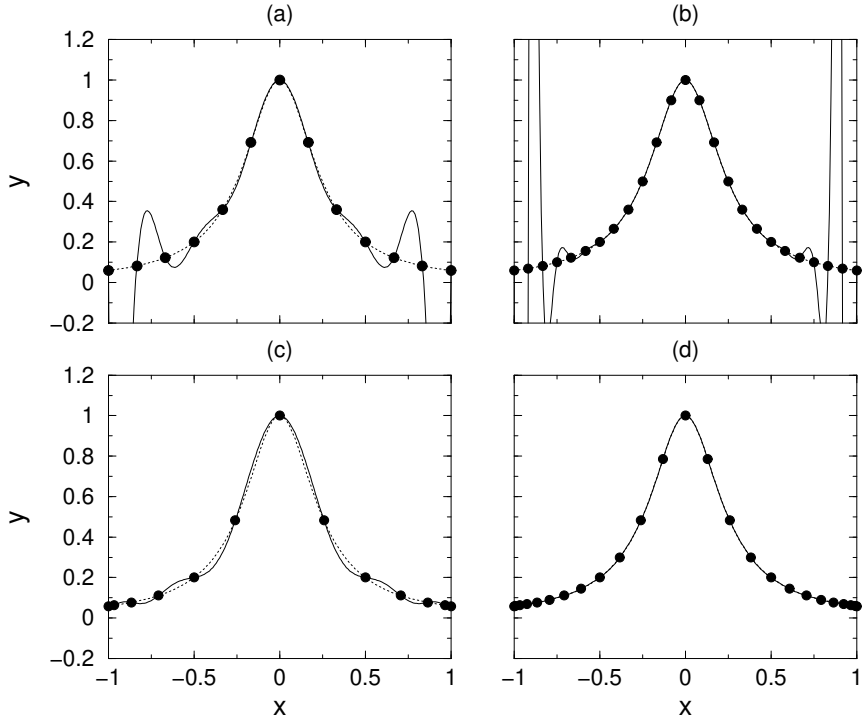

FIGURE 6. The function $y(x) = (1 + 16x^2)^{-1}$ for $x \in [-1, 1]$ (dotted line) with the bold points indicating the grid points for equi-spaced spaced points (a) $n = 12$ and (b) $n = 24$ and Chebychev clustered points (c) $n = 12$ and (d) $n = 24$. The solid lines are the polynomial approximations generated using the equi-spaced points (a)-(b) and clustered points (c)-(d) respectively.

which led to Runge phenomena. This polynomial approximation involves a clustered grid which is transformed to fit onto the unit circle, i.e. the Chebychev points. Thus we have the transformation

$$x_n = \cos(n\pi/N) \tag{2}$$

where $n = 0, 1, 2, \ldots, N$. This helps to greatly reduce the effects of the Runge phenomena.

The clustered grid approximation results in a polynomial approximation shown in Fig. 6(c)–(d). There are reasons for this great improvement using a clustered grid [**49**]. However, we will not discuss them since they are beyond the scope of this book. This clustered grid suggests an accurate and easy way to represent a function which does not have periodic boundaries.

5.4. Clustered points and Chebychev differentiation. The clustered grid given in method 3 above is on the Chebychev points. The resulting algorithm for constructing the polynomial approximation and differentiating it is as follows.

(1) Let p be a unique polynomial of degree $\leq N$ with $p(x_n) = V_n$, $0 \leq n \leq N$, where $V(x)$ is the function we are approximating.
(2) Differentiate by setting $w_n = p'(x_n)$.

The second step in the process of calculating the derivative is essentially the matrix multiplication

$$\mathbf{w} = \mathbf{D}_N \mathbf{v} \tag{3}$$

where $\mathbf{D}_N$ represents the action of differentiation. By using interpolation of the Lagrange form [**7**], the matrix elements of $p(x)$ can be constructed along with the $(N+1) \times (N+1)$ matrix $\mathbf{D}_N$. This

results in each matrix element $(D_N)_{ij}$ being given by

$$(D_N)_{00} = \frac{2N^2+1}{6} \tag{4a}$$

$$(D_N)_{NN} = -\frac{2N^2+1}{6} \tag{4b}$$

$$(D_N)_{jj} = -\frac{x_j}{2(1-x_j^2)} \quad j = 1, 2, \ldots, N-1 \tag{4c}$$

$$(D_N)_{ij} = \frac{c_i(-1)^{i+j}}{c_j(x_i - x_j)} \quad i, j = 0, 1, \ldots, N \ (i \neq j) \tag{4d}$$

where the parameter $c_j = 2$ for $j = 0$ or N or $c_j = 1$ otherwise.

Calculating the individual matrix elements results in the matrix $\mathbf{D}_N$

$$\mathbf{D}_N = \left(\begin{array}{c|ccc|c} \frac{2N^2+1}{6} & & \frac{2(-1)^j}{1-x_j} & & \frac{(-1)^N}{2} \\ \hline & \ddots & & \frac{(-1)^{i+j}}{x_i-x_j} & \\ \frac{-(-1)^i}{2(1-x_i)} & & \frac{-x_j}{2(1-x_j^2)} & & \frac{(-1)^{N+i}}{2(1+x_j)} \\ & \frac{(-1)^{i+j}}{x_i-x_j} & & \ddots & \\ \hline \frac{-(-1)^N}{2} & & \frac{-2(-1)^{N+j}}{1+x_j} & & -\frac{2N^2+1}{6} \end{array}\right) \tag{5}$$

which is a full matrix, i.e. it is not sparse. To calculate second, third, fourth and higher derivatives, simply raise the matrix to the appropriate power:

- $\mathbf{D}_N^2$ – second derivative
- $\mathbf{D}_N^3$ – third derivative
- $\mathbf{D}_N^4$ – fourth derivative
- $\mathbf{D}_N^m$ – mth derivative.

5.5. Boundaries. The construction of the differentiation matrix $\mathbf{D}_N$ does not explicitly include the boundary conditions. Thus the general differentiation given by (3) must be modified to include the given boundary conditions. Consider, for example, the simple boundary conditions

$$v(-1) = v(1) = 0. \tag{6}$$

The given differentiation matrix is then written as

$$\begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{N-1} \\ w_N \end{pmatrix} = \left(\begin{array}{c|c|c} & \text{top row} & \\ \hline \text{first column} & (D_N) & \text{last column} \\ \hline & \text{bottom row} & \end{array}\right) \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{N-1} \\ v_N \end{pmatrix}. \tag{7}$$

In order to satisfy the boundary conditions, we must manually set $v_0 = v_N = 0$. Thus only the interior points in the differentiation matrix are relevant. Note that for more general boundary conditions $v(-1) = \alpha$ and $v(1) = \beta$, we would simply set $v_0 = \alpha$ and $v_N = \beta$. The remaining $(N-1) \times (N-1)$ system is given by

$$\tilde{\mathbf{w}} = \tilde{\mathbf{D}}_N \tilde{\mathbf{v}} \tag{8}$$

where $\tilde{\mathbf{w}} = (w_1 w_2 \cdots w_{N-1})^T$, $\tilde{\mathbf{v}} = (v_1 v_2 \cdots v_{N-1})^T$ and we construct the matrix $\tilde{\mathbf{D}}_N$ with the simple python command:

```
tildeD=D[1:(N-1),1:(N-1)]
```

Note that the new matrix created is the old $\mathbf{D}_N$ matrix with the top and bottom rows and the first and last columns removed.

5.6. Connecting to the FFT. We have already discussed the connection of the Chebychev polynomial with the FFT algorithm. Thus we can connect the differentiation matrix with the FFT routine. After transforming via (2), then for real data the discrete Fourier transform can be used. For complex data, the regular FFT is used. Note that for the Chebychev polynomials

$$\frac{\partial T_n(\pm 1)}{\partial x} = 0 \tag{9}$$

so that no-flux boundaries are already satisfied. To impose pinned boundary conditions $v(\pm 1) = 0$, then the differentiation matrix must be imposed as shown above.

6. Implementing the Chebychev Transform

In order to make use of the Chebychev polynomials, we must generate our given function on the clustered grid given by

$$x_j = \cos(j\pi/N) \quad j = 0, 1, 2, \ldots, N. \tag{1}$$

This clustering will give higher resolution near the boundaries than the interior of the computational domain. The Chebychev differentiation matrix

$$\mathbf{D}_N \tag{2}$$

can then be constructed. Recall that the elements of this matrix are given by

$$(D_N)_{00} = \frac{2N^2+1}{6} \tag{3a}$$

$$(D_N)_{NN} = -\frac{2N^2+1}{6} \tag{3b}$$

$$(D_N)_{jj} = -\frac{x_j}{2(1-x_j^2)} \quad j = 1, 2, \ldots, N-1 \tag{3c}$$

$$(D_N)_{ij} = \frac{c_i(-1)^{i+j}}{c_j(x_i - x_j)} \quad i, j = 0, 1, \ldots, N \ (i \neq j) \tag{3d}$$

where the parameter $c_j = 2$ for $j = 0$ or N or $c_j = 1$ otherwise. Thus given the number of discretization points N, we can build the matrix $\mathbf{D}_N$ and also the associated clustered grid. The following python code simply required the number of points N to generate both of these fundamental quantities. Recall that it is assumed that the computational domain has been scaled to $x \in [-1, 1]$.

```
def cheb(N):
if N==0:
D = 0.; x = 1.
else:
n = arange(0,N+1)
x = cos(pi*n/N).reshape(N+1,1)
c = (hstack(( [2.], ones(N-1), [2.]))*(-1)**n).reshape(N+1,1)
X = tile(x,(1,N+1))
dX = X - X.T
```

```
D = dot(c,1./c.T)/(dX+eye(N+1))
D -= diag(sum(D.T,axis=0))
return D, x.reshape(N+1)
```

To test the differentiation, we consider two functions for which we know the exact values for the derivative and second derivative. Consider then

```
x = np.arange(-1, 1.01, 0.01)
u = np.exp(x) * np.sin(5 * x)
v = sech(x)
```

The first and second derivative of each of these functions is given by

```
ux = np.exp(x) * np.sin(5 * x) + 5 * np.exp(x) * np.cos(5 * x)
uxx = -24 * np.exp(x) * np.sin(5 * x) + 10 * np.exp(x) * np.cos(5 * x)
vx = -tanh(x) * sech(x)
vxx = sech(x) - 2 * (sech(x)**3)
```

We can also use the Chebychev differentiation matrix to numerically calculate the values of the first and second derivatives. All that is required is the number of discretation points N and the routine *cheb.m*.

```
N = 20
D, x2 = cheb(N)
D2 = np.dot(D, D)  # Second derivative matrix
```

Given the differentiation matrix $\mathbf{D}_N$ and clustered grid, the given function and its derivatives can be constructed numerically.

```
u2 = np.exp(x2) * np.sin(5 * x2)
v2 = sech(x2)
u2x = np.dot(D, u2)
v2x = np.dot(D, v2)
u2xx = np.dot(D2, u2)
v2xx = np.dot(D2, v2)
```

A comparison between the exact values for the differentiated functions and their approximations is shown in Fig. 7. As is expected the agreement is best for higher values of N. The python code for generating these graphical figures is given by

```
plt.plot(x, u, 'r', x2, u2, '.', x, v, 'g', x2, v2, '.')
plt.plot(x, ux, 'r', x2, u2x, '.', x, vx, 'g', x2, v2x, '.')
plt.plot(x, uxx, 'r', x2, u2xx, '.', x, vxx, 'g', x2, v2xx, '.')
```

6.1. Differentiation matrix in 2D. Unlike finite difference schemes which result in sparse differentiation matrices, the Chebychev differentiation matrix is full. The construction of 2D differentiation matrices thus would seem to be a complicated matter. However, the use of the *kron*

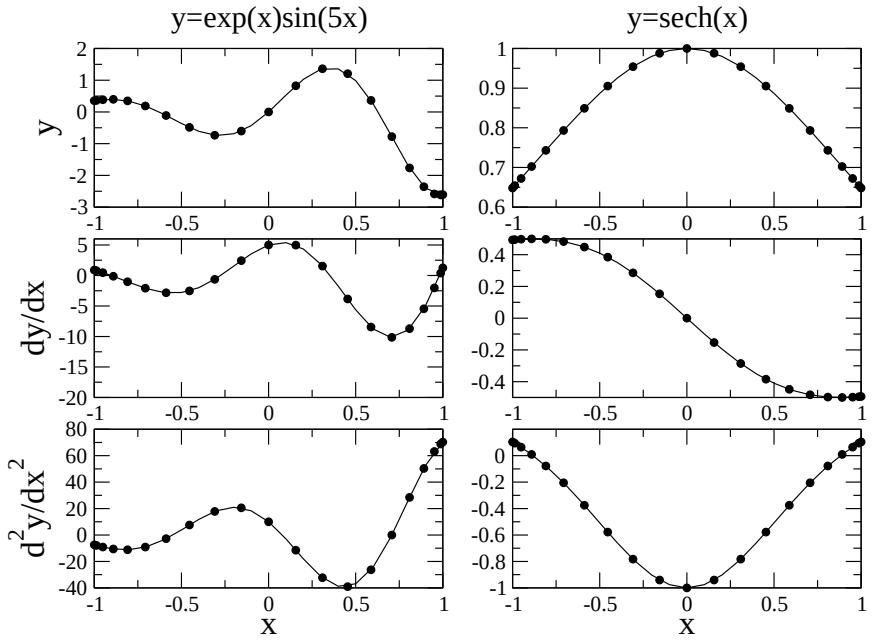

FIGURE 7. The function $y(x) = \exp(x)\sin 5x$ (left) and $y(x) = \mathrm{sech}x$ (right) for $x \in [-1, 1]$ and their first and second derivatives. The dots indicate the numerical values while the solid line is the exact solution. For these calculations, $N = 20$ in the differentiation matrix $\mathbf{D}_N$.

command in python makes this calculation trivial. In particular, the 2D Laplacian operator L given by

$$Lu = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{4}$$

can be constructed with

```
I = np.eye(len(D2))
L = kron(I, D2) + kron(D2, I)  # 2D Laplacian
```

where $D2 = D^2$ and I is the identity matrix of size $N \times N$. The $D2$ in each slot takes the x and y derivatives, respectively.

6.2. Solving PDEs with the Chebychev differentiation matrix. To illustrate the use of the Chebychev differentiation matrix, the two-dimensional heat equation is considered:

$$\frac{\partial u}{\partial t} = \nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{5}$$

on the domain $x, y \in [-1, 1]$ with the Dirichlet boundary conditions $u = 0$.

The number of Chebychev points is first chosen and the differentiation matrix constructed in one dimension:

```
N = 40
D, x = cheb(N)
D[N, :] = 0
D[0, :] = 0
```

```
D2 = np.dot(D, D)
```

The Neumann boundary conditions were imposed by modifying the first and last rows of the differentiation matrix D. Specifically, those rows were set to zero. The initial conditions for this simulation will be a Gaussian centered at the origin.

```
y = x
X, Y = np.meshgrid(x, y)
U = np.exp(-(X**2 + Y**2) / 0.1)
```

Note that the vectors x and y in the *meshgrid* command correspond to the Chebychev points for a given N. The final step is to build a loop which will advance and plot the solution in time. This requires the use of the *reshape* command, since we are in two dimensions, and an ODE solver such as *ode23*.

```
def heatrhs2D(u, t, L, mu):
    return mu*np.dot(L, u)

u = U.reshape((N + 1) ** 2)
mu = 1
for j in range(4):
    tspan = np.array([0, 0.05])
    ysol = odeint(heatrhs2D, u, tspan, args=(L,mu))
    u = ysol[-1]
    U = u.reshape(N + 1, N + 1)
```

This code will advance the solution $\Delta t = 0.05$ in the future and plot the solution. Figure 8 depicts the two-dimensional diffusive behavior given the Dirichlet boundary conditions using the Chebychev differentiation matrix.

7. Computing Spectra: The Floquet–Fourier–Hill Method

Spectral transforms, such as Fourier and Chebychev, have many advantages including their performance speed of $O(N \log N)$ and their spectral accuracy properties. Given these advantages, it is desirable to see where else such methods can play a role in practice. One area that we now return to is the idea of computing spectra of linear operators. This was introduced via finite difference discretization in Section 8. But just as finite difference discretization of PDEs can be replaced in certain cases by spectral discretization of PDEs, so can the computation of spectra of linearized operators be replaced with spectral based methods [**50**].

The methodology developed here will be applied to finding spectra (all eigenvalues) of the eigenvalue problem

$$\mathcal{L}v = \lambda v \tag{1}$$

where the linear operator is assumed to be of the form

$$\mathcal{L} = \sum_{k=0}^{M} f_k(x)\partial_x^k = f_0(x) + f_1(x)\partial_x + \cdots + f_M(x)\partial_x^M \tag{2}$$

with the periodic constraint $f_k(x + L) = f_k(x)$. Thus there are restrictions on how broadly the technique can be applied. Generically, it is for periodic problems only. However, for problems of infinite extent where the solutions $v(\pm\infty) \to 0$, the method is often successful when considering

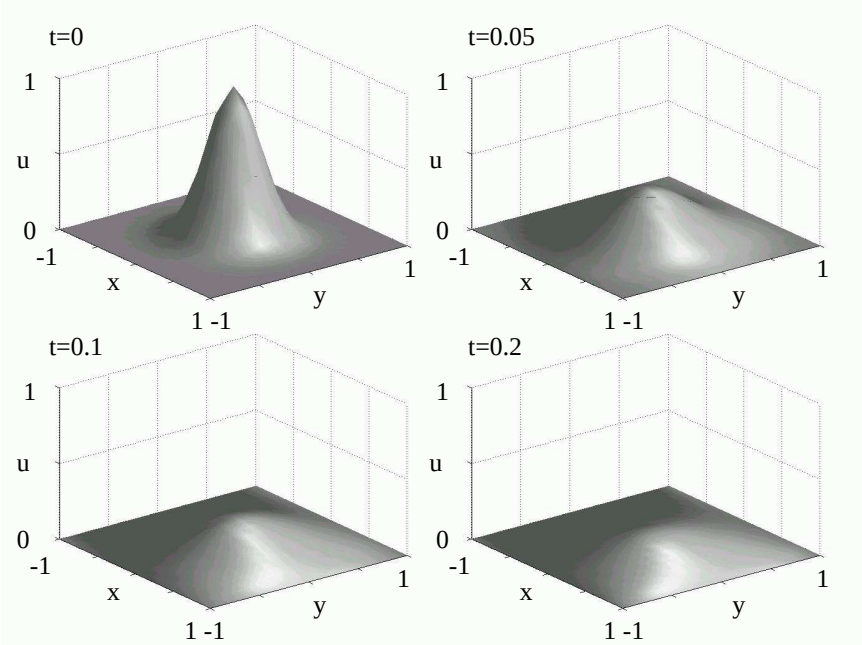

FIGURE 8. Evolution of an initial two dimensional Gaussian governed by the heat equation on the domain $x, y \in [-1, 1]$ with Dirichlet boundary conditions $u = 0$. The Chebychev differentiation matrix is used to calculate the Laplacian with $N = 30$.

a large, but finite sized domain, i.e. the periodic boundary conditions suffice to approximate the decaying to zero solutions at the boundary. The Floquet–Fourier–Hill method can also be generalized to a vector version [**50**].

The numerical technique developed here is based upon ideas of Fourier analysis and Floquet theory. George Hill first used variations of this technique in 1886 [**51**] on what is now called Hill's equation. One of the great open questions of his day concerned the stability of orbits in planetary and lunar motion. As such, the method will be called the Floquet–Fourier–Hill (FFH) method [**50**]. The method allows for the efficient computation of the spectra of a linear operator by exploiting numerous advantages over finite difference schemes, including accuracy and speed.

The coefficients of the operator $\mathcal{L}$ in (2) are periodic with period L, thus they can be represented by a Fourier series expansion

$$f_k(x) = \sum_{j=-\infty}^{\infty} \hat{f}_{k,j} \exp(i2\pi jx/L), \qquad k = 0, \cdots, M \tag{3}$$

where the Fourier coefficients are determined from the inner product integral

$$\hat{f}_{k,j} = \frac{1}{L} \int_{-L/2}^{L/2} f_k(x) \exp(-i2\pi jx/L) dx, \qquad k = 0, \cdots, M. \tag{4}$$

Recall that the variable i again represents the imaginary unit. Thus far, this is standard Fourier theory. In fact, the FFT algorithm computes such Fourier coefficients in $O(N \log N)$ time.

Floquet theory dictates that every bounded solution of the fundamental equation (1) is of the form [**52, 53**]

$$w(x) = \exp(i\mu x)\phi(x) \tag{5}$$

with $\phi(x + L) = \phi(x)$ for any fixed λ and $\mu \in [0, 2\pi/L)$. The factor $\exp(i\mu x)$ is referred to as a Floquet multiplier and $i\mu$ is the Floquet exponent [**52, 53**]. In different areas of science, Floquet

theory may be known as monodromy theory (Hamiltonian systems, etc.) or Bloch theory (solid state theory, etc). In the context of Bloch theory the Floquet exponent is often referred to as the quasi-momentum.

Since the function $\phi(x)$ is also periodic with a period L, it can also be expanded in a Fourier series so that

$$w(x) = \exp(i\mu x) \sum_{j=-\infty}^{\infty} \hat{\phi}_j \exp(i2\pi jx/L) = \sum_{j=-\infty}^{\infty} \hat{\phi}_j \exp(ix[\mu + 2\pi j/L]) \tag{6}$$

where

$$\hat{\phi}_j = \frac{1}{L} \int_{-L/2}^{L/2} \phi(x) \exp(-i2\pi jx/L) dx \tag{7}$$

is the jth Fourier coefficient of $\phi(x)$. Upon multiplying (1) by $\exp(-i\mu x)$ and noting that any term of the resulting equation is periodic, the nth Fourier coefficient of the eigenvalue equation becomes, after some manipulation and using orthogonality [**50**],

$$\sum_{m=-\infty}^{\infty} \left(\sum_{k=0}^{M} \hat{f}_{k,n-m} \left[i \left(\mu + \frac{2\pi m}{L} \right) \right]^k \right) \hat{\phi}_m = \lambda \hat{\phi}_m. \tag{8}$$

Thus the original eigenvalue problem has been transformed into the Fourier domain and the eigenvalue problem

$$\hat{\mathcal{L}}(\mu)\hat{\phi} = \lambda \hat{\phi} \tag{9}$$

with $\hat{\phi} = (\cdots, \hat{\phi}_{-2}, \hat{\phi}_{-1}, \hat{\phi}_0, \hat{\phi}_1, \hat{\phi}_2, \cdots)^T$ and where the μ-dependence of $\hat{\mathcal{L}}$ is explicitly indicated. This is a bi-infinite matrix with the linear operator in the spectral domain defined by

$$\hat{\mathcal{L}}(\mu)_{nm} = \sum_{k=0}^{M} \hat{f}_{k,n-m} \left[i \left(\mu + \frac{2\pi m}{L} \right) \right]^k. \tag{10}$$

Note that to this point, *no approximations were used*. The problem was simply transformed to the Fourier domain. To compute the eigenvalue spectra, the bi-infinite matrix (9) is approximated by limiting the number of Fourier modes. Specifically, a cut-off wavenumber, N, is chosen which results in a matrix system of size $2N + 1$:

$$\hat{\mathcal{L}}_N(\mu)\hat{\phi}_N = \lambda_N \hat{\phi}_N \tag{11}$$

where the λ_N are now numerical approximations to the true eigenvalues λ. Convergence of this scheme is considered further in Ref. [**50**]. Note that the **eigs** command still plays the underlying and fundamental role, but now for a transformed matrix and vector space.

7.1. Example: Schrödinger operators. To implement this scheme, an example is considered arising from Schrödinger operators. Specifically, the following two operators are considered:

$$\mathcal{L}_- v = -\frac{d^2v}{dx^2} + (1 - 2\,\mathrm{sech}^2 x)v = \lambda v \tag{12a}$$

$$\mathcal{L}_+ v = -\frac{d^2v}{dx^2} + (1 - 6\,\mathrm{sech}^2 x)v = \lambda v. \tag{12b}$$

These operators are chosen since their spectrum is known completely. In particular, the operator $\mathcal{L}_-$ has a discrete eigenvalue at zero with eigenfunction $v_{\lambda=0} = \mathrm{sech}\, x$. It also has a continuum of eigenvalues for $\lambda \in [1, \infty)$. The operator $\mathcal{L}_+$ has two discrete eigenvalues at $\lambda = -3$ and $\lambda = 0$ with eigenfunctions $v_{\lambda=-3} = \mathrm{sech}^2 x$ and $v_{\lambda=0} = \mathrm{sech}\, x \tanh x$. It also has a continuum of eigenvalues for $\lambda \in [1, \infty)$.

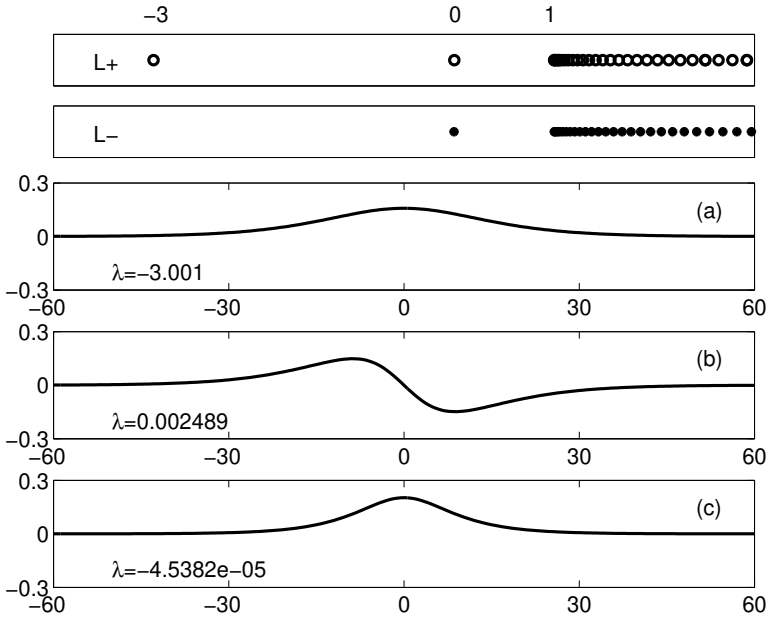

FIGURE 9. The top panels are the spectral of the operators $\mathcal{L}_+$ and $\mathcal{L}_-$. The operator $\mathcal{L}_+$ is known to have two discrete eigenvalues at $\lambda = -3$ and $\lambda = 0$ with eigenfunctions $v_{\lambda=-3} = \text{sech}^2 x$ and $v_{\lambda=0} = \text{sech}x \tanh x$. It also has a continuum of eigenvalues for $\lambda \in [1, \infty)$. The panels (a) and (b) demonstrate these two eigenfunctions computed numerically along with the approximate evaluation of the eigenvalue. The operator $\mathcal{L}_-$ has a discrete eigenvalue at zero with eigenfunction $v_{\lambda=0} = \text{sech}x$. It also has a continuum of eigenvalues for $\lambda \in [1, \infty)$. Its eigenfunction and computed eigenvalue is demonstrated in panel (c).

For these two examples, we have the following relations, respectively, back to (2)

$$f_0(x) = 1 - 2\,\text{sech}^2 x, \quad f_1(x) = 0, \quad f_2(x) = -1 \tag{13a}$$

$$f_0(x) = 1 - 6\,\text{sech}^2 x, \quad f_1(x) = 0, \quad f_2(x) = -1. \tag{13b}$$

The only Fourier expansion that needs to be done then is for the term $\text{sech}^2 x$. The following python code constructs the FFH eigenvalue problem.

```
from scipy.integrate import quad

L = 40  # box size  -L,L
n = 200  # modes
TOL = 1e-8

a = np.zeros(n + 1)
for j in range(1, n + 2):
    def integrand(x, j, L):
        return (1 / (2 * L)) * np.cos((j - 1) * np.pi * x / L) * sech(x) ** 2

    a[j - 1], _ = quad(integrand, -L, L, args=(j, L), epsabs=TOL, epsrel=TOL)

A = np.zeros((n + 1, n + 1))
for j in range(1, n + 2):
    A[j - 1, j - 1] = a[0]
```

```
for jj in range(2, n + 2):
    for j in range(jj, n + 2):
        A[j - jj, j - 1] = a[jj - 1]
        A[j - 1, j - jj] = a[jj - 1]

D2 = np.zeros((n + 1, n + 1))
for j in range(1, n + 2):
    D2[j - 1, j - 1] = -1 - ((np.pi / L) ** 2) * (n / 2 + 1 - j) ** 2

Lplus = D2 + 6 * A
lam, V = np.linalg.eig(Lplus)
lam = np.real(lam)

sorted_indices = np.argsort(np.real(lam))[::-1]
lamsort = lam[sorted_indices]
Vsort =V[:, sorted_indices]

Lminus = D2 + 2 * A
lam2, V2 = np.linalg.eig(Lminus)
lam2 = np.real(lam2)

sorted_indices = np.argsort(np.real(lam2))[::-1]
lam2sort = lam2[sorted_indices]
V2sort =V2[:, sorted_indices]
```

In this example where the operator is of the Sturm–Liouville form, the inner product only needs to be computed with respect to $\cos(n\pi x/L)$ versus $\exp(in\pi x/L)$. Figure 9 shows the computed spectrum of both the $\mathcal{L}_+$ and $\mathcal{L}_-$ operators. The eigenfunctions are also shown for the three discrete eigenvalues, two for $\mathcal{L}_+$ and one for $\mathcal{L}_-$, respectively. The computation uses $\Delta x = 0.6$ which would only give an accuracy of 10^{-1}. However, the computation with FFH is accurate to 10^{-3} or more. Note that using the **quad** command is a highly inefficient way to compute the Fourier coefficients. Using the FFT algorithm is much faster and ultimately the way to do the computation. However, the above does illustrate the FFH process in its entirety.

7.2. Example: Mathieu's equation. As a second example of computing spectra in a highly efficient way, consider the Mathieu equation of mathematical physics

$$\frac{d^2v}{dx^2} + [\lambda - 2q\cos(\omega x)]\, v = 0. \tag{14}$$

This gives the linear operator $\mathcal{L} = -d^2/dx^2 + 2q\cos(\omega x)$. The equation originates from the Helmholtz equation through the use of separation of variables. Here we are interested in determining all λ values for which bounded solutions exist. Many texts on perturbation methods use this equation as one of their prototypical examples. One of their goals is to determine the edges of the spectrum for varying q.

In this case, we have the following relations back to (2)

$$f_0(x) = 2q\cos(\omega x), \quad f_1(x) = 0, \quad f_2(x) = -1. \tag{15}$$

Thus a fairly simple structure exists with $f_0(x)$ already being represented as Fourier modes if we choose ω to be an integer. In what follows, ω is chosen to be an integer and the eigenvalues are found as a function of the variable q.

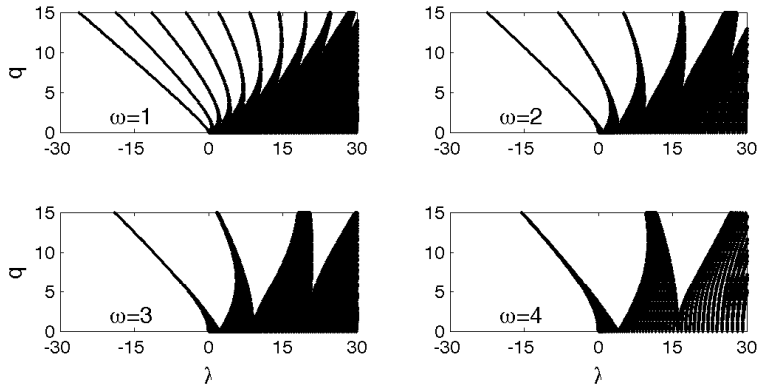

FIGURE 10. Spectrum of Mathieu's equation for four different values of ω. Every black point is an approximate point of the spectrum and no filling in of spectral regions was done. Standard perturbation theory is typically only able to approximate the spectrum near $q = 0$.

```
q_values = np.arange(0, 15.1, 0.1)  # q values to consider
n = 10  # number of Fourier modes
cuts = 40  # number of mu slices
freq = 2  # frequency omega

for q in q_values:   # Loop over q values
    lam = []
    for jcut in np.arange(-freq / 2, freq / 2 + freq / cuts, freq / cuts):
        A = np.zeros((2 * n + 1, 2 * n + 1))
        for j in range(1, 2 * n + 2):
            A[j - 1, j - 1] = (n + 1 - j + jcut) ** 2  # derivative elements

        for j in range(1, 2 * n + 2 - freq):
            A[j + freq - 1, j - 1] = q
            A[j - 1, j + freq - 1] = q

        W, V = np.linalg.eig(A)  # compute eigenvalues
        lam.extend(W.real)  # track all eigenvalues
```

Unlike the previous example, no Fourier mode projection needs to be done since $f_0(x) = 2q\cos(\omega x)$ is already in Fourier mode form. Thus only the coefficient $2q$ is needed at frequency ω. Figure 10 shows the resulting band-gap structure of eigenvalues for four different values of ω. All points in the plot are computed points and no filling-in of the spectrum was done.

To see how to generalize the FFH method to a vector model and/or higher dimensions, see Ref. [**50**]. In general, one should think about this spectral based FFH method as simply taking advantage of spectral accuracy and speed. Just as PDE based solvers aim to take advantage of these features for accurately and quickly solving a given problem, the FFH should always be used when either an infinite domain (with decaying boundary conditions) or periodic solutions are of interest.

8. Problems and Exercises

8.1. Mode-Locked Lasers. The invention of the laser in 1960 is a historical landmark of scientific innovation. In the decades since, the laser has evolved from a fundamental science phenomenon to a ubiquitous, cheap, and readily available commercial commodity. A large variety of laser technologies and applications exists from the commercial, industrial, medical, and academic arena. In many of these applications, the laser is required to produce ultrashort pulses (e.g., tens to hundreds femtoseconds), which are localized light waves, with nearly uniform amplitudes. The generation of such localized light pulses is the subject mode-locking theory.

As with all electromagnetic phenomena, the propagation of an optical field in a given medium is governed by Maxwell's equations:

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{16a}$$

$$\nabla \times \mathbf{H} = \mathbf{J}_f + \frac{\partial \mathbf{D}}{\partial t} \tag{16b}$$

$$\nabla \cdot \mathbf{D} = \rho_f \tag{16c}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{16d}$$

Here the electromagnetic field is denoted by the vector $\mathbf{E}(x, y, z, t)$ with the corresponding magnetic field, electromagnetic and magnetic flux densities being denoted by $\mathbf{H}(x, y, z, t)$, $\mathbf{D}(x, y, z, t)$ and $\mathbf{B}(x, y, z, t)$ respectively. In the absence of free charges, which is the case of interest here, the current density and free charge density are both zero so that $\mathbf{J}_f = 0$ and $\rho_f = 0$.

The flux densities $\mathbf{D}$ and $\mathbf{B}$ characterize the constitutive laws of a given medium. Therefore, any specific material of interest is characterized by the constitutive laws and their relationship to the electric and magnetic fields:

$$\mathbf{D} = \epsilon_0 \mathbf{E} + \mathbf{P} = \epsilon \mathbf{E} \tag{17a}$$

$$\mathbf{B} = \mu_0 \mathbf{H} + \mathbf{M} \tag{17b}$$

where ϵ_0 and μ_0 are the free space permittivity and free space permeability respectively, and $\mathbf{P}$ and $\mathbf{M}$ are the induced electric and magnetic polarizations. At optical frequencies, $\mathbf{M} = 0$. The nonlocal, nonlinear response of the medium to the electric field is captured by the induced polarization vector $\mathbf{P}$.

In what follows, interest will be given solely to the electric field and the induced polarization. Expressing Maxwell's equations in terms of these two fundamental quantities $\mathbf{E}$ and $\mathbf{P}$ can be easily accomplished by taking the curl of (16a) and using (16b), (17a) and (17b). This yields the electric field evolution

$$\nabla^2 \mathbf{E} - \nabla(\nabla \cdot \mathbf{E}) - \frac{1}{c^2}\frac{\partial^2 \mathbf{E}}{\partial t^2} = \mu_0 \frac{\partial^2 \mathbf{P}(\mathbf{E})}{\partial t^2} \tag{18}$$

For the quasi one-dimensional case of fiber propagation, the above expression for the electric field reduces to

$$\frac{\partial^2 E}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 E}{\partial t^2} = \mu_0 \frac{\partial^2 P(E)}{\partial t^2} \tag{19}$$

where now the electric field is expressed as the scalar quantity $E(x, t)$. Note that we could formally handle the transverse field structure as well in an asymptotic way, but for the purposes of this exercise, this is not necessary.

Only the linear propagation is considered at first. The nonlinearity will be added afterwards. Thus one considers the following linear function with linear polarization response function:

$$\frac{\partial^2 E}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 E}{\partial t^2} = \frac{1}{c^2}\frac{\partial^2}{\partial t^2}\left(\int_{-\infty}^{t} \chi^{(1)}(t-\tau)E(\tau, x)d\tau\right) \tag{20}$$

where $\chi^{(1)}$ is the linear response to the electric field that includes the time response and causality. It should be recalled that $c^2 = 1/(\epsilon_0\mu_0)$. The left hand side of the equation is the standard wave equation that can be solved analytically. It generically yields waves propagating left and right in the media at speed c. Here, we are interested in waves moving only in a single direction down the fiber.

The center frequency expansion is an asymptotic approach that assumes the electromagnetic field, to leading order, is at a single frequency. A slowly-varying envelope equation is then derived for the evolution of an envelope equation that contains many cycles of the electric field. Thus at leading order, one can think of this approach as considering a delta function in frequency and plane wave in time.

The center frequency expansion begins by assuming

$$E(x,t) = Q(x,t)e^{i(kx-\omega t)} + c.c. \tag{21}$$

where *c.c.* denotes complex conjugate and k and ω are the wavenumber and optical frequency respectively. These are both large quantities so that $k, \omega \gg 1$. Indeed, in this asymptotic expansion the small parameter is given by $\epsilon = 1/k \ll 1$. Note that if Q is constant, than this assumption amounts to assuming a plane wave solution, i.e. a delta function in frequency.

Inserting (21) into (20) yields the following:

$$\begin{aligned} &e^{i(kx-\omega t)}\left[Q_{xx} + 2ikQ_x - k^2Q\right] - \frac{1}{c^2}e^{i(kx-\omega t)}\left[Q_{tt} + 2i\omega Q_t - \omega^2 Q\right] \\ &= \frac{1}{c^2}\frac{\partial^2}{\partial t^2}\left(\int_{-\infty}^{t}\chi^{(1)}(t-\tau)Q(\tau,x)e^{i(kx-\omega\tau)}d\tau\right) \end{aligned} \tag{22}$$

where subscripts denote partial differentiation. In the integral on the right hand side, a change of variables is made so that $\sigma = t - \tau$ and

$$\begin{aligned} &e^{i(kx-\omega t)}\left[Q_{xx} + 2ikQ_x - k^2Q\right] - \frac{1}{c^2}e^{i(kx-\omega t)}\left[Q_{tt} + 2i\omega Q_t - \omega^2 Q\right] \\ &= \frac{1}{c^2}\frac{\partial^2}{\partial t^2}\left(e^{i(kx-\omega t)}\int_{0}^{\infty}\chi^{(1)}(\sigma)Q(t-\sigma,x)d\sigma\right). \end{aligned} \tag{23}$$

Using the following Taylor expansion

$$Q(t-\sigma,x) = Q(t,x) - \sigma Q_t(x,t) + \frac{1}{2}\sigma^2 Q_{tt}(x,t) + \cdots \tag{24}$$

in (23) results finally in the following equation for $Q(x,t)$:

$$\begin{aligned} &Q_{xx} + 2ikQ_x - k^2Q + \frac{1}{c^2}\left[\omega^2(1+\hat{\chi})Q + i[(\omega^2\hat{\chi})_\omega + 2\omega]Q_t\right. \\ &\qquad \left. -1/2[(\omega^2\hat{\chi})_{\omega\omega} + 2]Q_{tt} + \cdots\right] = 0 \end{aligned} \tag{25}$$

where $\hat{\chi} = \int_0^\infty \chi^{(1)}(\sigma)e^{i\omega\sigma}d\sigma$ is the Fourier transform of the linear susceptibility function $\chi^{(1)}$. It is at this point that the asymptotic balancing of terms begins. In particular, the leading order balance with the largest terms of size k^2 and ω^2 so that

$$k^2 = \frac{\omega^2}{c^2}(1+\hat{\chi}). \tag{26}$$

For now, we assume that the linear susceptibility is real. An imaginary part will lead to attenuation. This will be considered laser in the context of the laser cavity, but not now in the envelop approach.

Once the dominant balance (26) has been established, it is easy to show that the following two relations hold:

$$2kk' = \frac{1}{c^2}[(\omega^2\hat{\chi})_\omega + 2\omega] \tag{27a}$$

$$kk'' + (k')^2 = \frac{1}{2c^2}[(\omega^2\hat{\chi})_{\omega\omega} + 2] \tag{27b}$$

where the primes denote differentiation with respect to ω. Such relations can be found for higher-order terms in the Taylor expansion of the integral. Thus the equation for $Q(x,t)$ reduces to

$$2ik(Q_x + k'Q_t) + Q_{xx} - [kk'' + (k')^2]Q_{tt} + \sum_{n=3}^{\infty} \beta_n \partial_t^{(n)} Q = 0\,. \tag{28}$$

Noting that $k' = dk/d\omega$ is in fact the definition of the inverse group velocity in wave propagation problems, we can move into the group-velocity frame of reference by defining the new variables:

$$T = t - k'x \tag{29a}$$

$$Z = x\,. \tag{29b}$$

This yields the governing linear equations

$$iQ_Z - \frac{k''}{2}Q_{TT} + \sum_{n=3}^{\infty} \beta_n \partial_T^{(n)} Q = 0 \tag{30}$$

where the β_n are the coefficients of the higher-order dispersive terms (β_n for $n > 2$). If we ignore the higher-order dispersion, i.e. assume $\beta_n = 0$, then the resulting equation is just the linear Schrödinger equation we expect. Note that the Z and T act as the *time* and *space* variables respectively in the moving coordinate system.

The nonlinearity will now be included with the linear susceptibility response. At present, only an instantaneous response will be introduced so that the governing equation (20) is modified to

$$\frac{\partial^2 E}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 E}{\partial t^2} = \frac{1}{c^2}\frac{\partial^2}{\partial t^2}\left[\int_{-\infty}^{t} \chi^{(1)}(t-\tau)E(\tau,x)d\tau + \chi^{(3)}E^3\right] \tag{31}$$

where $\chi^{(3)}$ is the nonlinear (cubic) susceptibility. Note that it is assumed that the propagation is in a centro-symmetric material so that all $\chi^{(2)} = 0$. Both the center-frequency and short-pulse asymptotics proceed to balance the cubic response with the dispersive effects derived in the equations (30) and (31).

In the center frequency asymptotics, the cubic term can be carried through the derivation assuming the ansatz (21) and following the steps (22) through (29). The leading order contribution to (25) becomes

$$\frac{\omega^2}{c^2}\chi^{(3)}|Q|^2 Q + \cdots \tag{32}$$

where the dots represent higher-order terms. Note that because of the ansatz approximation (21), there are no derivatives on the cubic term. Thus the effects of the two derivatives applied to the nonlinear term in (31) produces at leading order the derivative of the plane wave ansatz (21), i.e. it simply produces a factor of ω^2 in front of the nonlinearity. This yields the governing linear equations

$$iQ_Z - \frac{k''}{2}Q_{TT} + \sum_{n=3}^{\infty} \beta_n \partial_T^{(n)} Q + \alpha|Q|^2 Q = 0 \tag{33}$$

where $\alpha = \omega^2\chi^{(3)}/2kc^2$.

With appropriate normalization and only including the second-order chromatic dispersion, the standard nonlinear Schrödinger equation is obtained:

$$iQ_Z \pm \frac{1}{2}Q_{TT} + |Q|^2 Q = 0 \tag{34}$$

Such a description holds if the envelope $Q(Z,T)$ contains many cycles of the electric field so that the asymptotic ordering based upon $1/k \ll 1$ hold. If only a few cycles of the electric field are underneath the envelope, than the asymptotic ordering that occurs in (25) cannot be applied and the reduction to the NLS description is suspect. Note that the dispersion can be either positive (anomalous dispersion) or negative (normal dispersion).

Saturating Gain Dynamics and Attenuation. In addition to the two dominant effects of dispersion and self-phase modulation as characterized by the NLS equation (34), a given laser cavity also must include the gain and loss dynamics. The loss is not only due to the attenuation in splicing the laser components together, but also from inclusion of an output coupler in the laser cavity that taps off a certain portion of the laser cavity energy every round trip. Thus gain must be applied to the laser system. A simple model that accounts for both gain saturation that is bandwidth limited and attenuation is the following:

$$Q_Z = -\Gamma Q + g(Z)\left(1 + \tau\partial_T^2\right) Q \tag{35}$$

where

$$g(Z) = \frac{2g_0}{1 + \|Q\|^2/E_0}\,. \tag{36}$$

Here Γ measures the attenuation rate of the cavity, the parameter τ measures the bandwidth of the gain (assumed to be parabolic in shape around the center-frequency), g_0 is the gain pumping strength, and E_0 is the cavity saturation energy. Note that $\|Q\|^2 = \int_{-\infty}^{\infty} |Q|^2 dT$ is the total cavity energy. Thus the gain dynamics $g(Z)$ provides the saturation behavior that is characteristic of any physically realistic gain media. Specifically, as the cavity energy grows, the effective gain $g(Z)$ decreases since the number of atoms in the inverted population state become depleted.

Intensity Discrimination. In addition to the gain and loss dynamics, one other physically important effect must be considered, namely the intensity discrimination (or saturable absorption) necessary for laser mode-locking. Essentially, there must be some physical mechanism in the cavity that favors high intensities over low intensities of the electric field, thus allowing for an intensity selection mechanism. Physically, such an effect has been achieved in a wide variety of physical scenarios including the nonlinear polarization laser, a laser mode-locked via nonlinear interferometry, or quantum saturable absorbers. A phenomenological approach to considering such a phenomena can be captured by the simple equation

$$Q_Z = \beta|Q|^2 Q - \sigma|Q|^4 Q \tag{37}$$

where β models a cubic gain and σ is a quintic saturation effect. Normally, the cubic gain would dominate and the quintic saturation is only proposed to prevent solutions from blowing up. Thus typically $\sigma \ll \beta$.

The Haus Master Mode-Locking Model. The late Hermann Haus of the Massachusetts Institute of Technology proposed that the dominant laser cavity effects should be included in a single phenomenological description of the electric field evolution [**54**]. As such, he proposed the master mode-locking model that includes the chromatic dispersion and nonlinearity of (34), the saturating gain and loss of (36), and the intensity discrimination of (37). Averaged together, they produce the Ginzburg-Landau like equation [**47**]

$$iQ_Z \pm \frac{1}{2}Q_{TT} + \alpha|Q|^2 Q = i\left[g(Z)\left(1 + \tau\partial_T^2\right) Q - \Gamma Q + \beta|Q|^2 Q - \sigma|Q|^4 Q\right] \tag{38}$$

where the gain $g(Z)$ is given by (36).

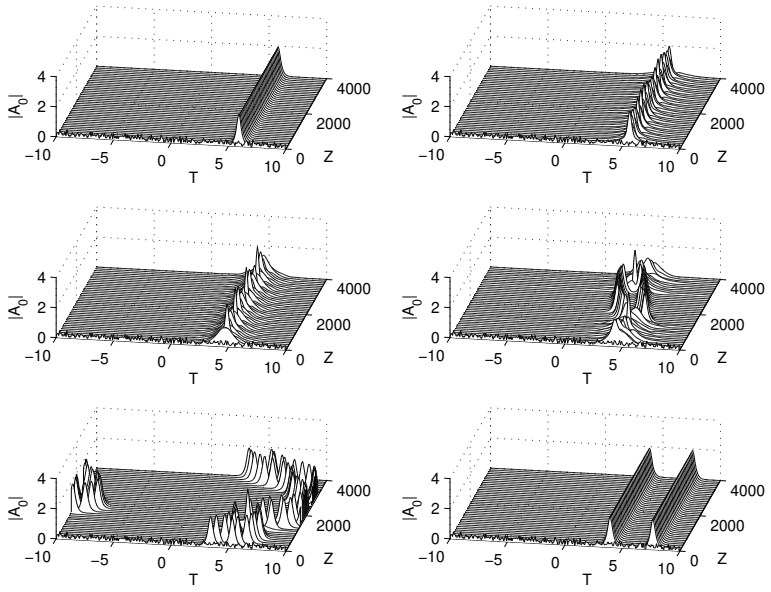

FIGURE 11. Temporal evolution and associated bifurcation structure of the transition from one pulse per round trip to two pulses per round trip [**57**]. The corresponding values of gain are $g_0 = 2.3, 2.35, 2.5, 2.55, 2.7$, and 2.75. For the lowest gain value only a single pulse is present. The pulse then becomes a periodic breather before undergoing a "chaotic" transition between a breather and two-pulse solution. Above a critical value ($g_0 \approx 2.75$), two pulse are stabilized. [from J. N. Kutz and B. Sandstede, *Theory of passive harmonic mode-locking using waveguide arrays*, Optics Express **16**, 636-650 (2008) ©OSA]

Mode-Locking with Waveguide Arrays. Another model of mode-locking is based on a more quantitative approach to the saturable absorption mechanism. The intensity discrimination in the specific laser considered here is provided by the nonlinear mode-coupling in the waveguides as described in the previous section [**55**]. When placed within an optical fiber cavity, the pulse shaping mechanism of the waveguide array leads to stable and robust mode-locking. The resulting approximate evolution dynamics describing the waveguide array mode-locking is given by [**56**]

$$i\frac{\partial Q}{\partial Z} \pm \frac{1}{2}\frac{\partial^2 Q}{\partial T^2} + \alpha|Q|^2 Q + CV + i\gamma_0 Q - ig(Z)\left(1+\tau\frac{\partial^2}{\partial T^2}\right)Q = 0 \tag{39a}$$

$$i\frac{\partial V}{\partial Z} + C(W+Q) + i\gamma_1 V = 0 \tag{39b}$$

$$i\frac{\partial W}{\partial Z} + CV + i\gamma_2 W = 0\,, \tag{39c}$$

where the gain $g(Z)$ is again given by (36) and the $V(Z,T)$ and $W(Z,T)$ fields model the electromagnetic energy in the neighboring channels of the waveguide array. Note that the equations governing these neighboring fields are ordinary differential equations.

Project and Application:

Consider the mode-locking equations (38) and (39) with the gain given by (36). Use the following parameter values for (38) $(E_0, \tau, \alpha, \Gamma, \beta, \sigma) = (1, 0.1, 1, 1, 0.5, 0.1)$ with a negative sign of dispersion, and consider (39) with a positive sign of dispersion and $(E_0, \tau, C, \alpha, \gamma_0, \gamma_1, \gamma_2) =$

$(1, 0.1, 5, 8, 0, 0, 10)$.

Initial Conditions: For both models, assume white-noise initial conditions at the initial time $Q(0, T)$. Recall that Z is the time-like variable and T is the space like variable in this moving coordinate frame.

Boundary Conditions: The cavity is assumed to be very long in relation to the width of the pulse. Therefore, it is usually assumed that the localized mode-locked pulses are essentially in an infinite domain. To approximate this, one can simply use a large domain and an FFT based solution method. Be sure to pick a domain large enough so that boundary effects do not play a role.

(a) For both governing models (38) and (39), explore the dynamics as a function of increasing gain g_0. In particular, determine the ranges of stability for which stable 1-pulse mode-locking occurs.

(b) Carefully explore the region near the transition regions where the 1-pulse solution undergoes instability to a 2-pulse solution. Specifically, identify the region where periodic oscillations occur via a Hopf bifurcation and determine if there are any chaotic regions of behavior near the transition point.

(c) Determine a parameter regime for positive dispersion in (38) where stable mode-locking can occur. Key parameters to adjust for this model are α, β, and σ. Additionally, determine the mode-locking regime of stability for (39) with negative dispersion. The key parameters to adjust for this model are α and C.

8.2. Bose-Einstein Condensates. To consider a specific model that allows the connection from the microscopic scale to the macroscopic scale, we examine mean-field theory of many-particle quantum mechanics with the particular application of BECs trapped in a standing light wave [**58**]. The classical derivation given here is included to illustrate how the local model and its nonlocal perturbation are related. The inherent complexity of the dynamics of N pairwise interacting particles in quantum mechanics often leads to the consideration of such simplified mean-field descriptions. These descriptions are a blend of symmetry restrictions on the particle wave function [**59**] and functional form assumptions on the interaction potential [**59, 60, 61**].

The dynamics of N identical pairwise interacting quantum particles is governed by the time-dependent, N-body Schrödinger equation

$$i\hbar\frac{\partial\Psi}{\partial t}=-\frac{\hbar^2}{2m}\Delta^N\Psi+\sum_{i,j=1,i\neq j}^{N}W(\mathbf{x}_i-\mathbf{x}_j)\Psi+\sum_{i=1}^{N}V(\mathbf{x_i})\Psi, \tag{40}$$

where $\mathbf{x_i} = (x_{i1}, x_{i2}, x_{i3})$, $\Psi = \Psi(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, ..., \mathbf{x}_N, t)$ is the wave function of the N-particle system, $\Delta^N = \left(\nabla^N\right)^2 = \sum_{i=1}^{N}\left(\partial_{x_{i1}}^2 + \partial_{x_{i2}}^2 + \partial_{x_{i3}}^2\right)$ is the kinetic energy or Laplacian operator for N-particles, $W(\mathbf{x_i} - \mathbf{x_j})$ is the symmetric interaction potential between the i-th and j-th particle, and $V(\mathbf{x}_i)$ is an external potential acting on the i-th particle. Also, $\hbar$ is Planck's constant divided by 2π and m is the mass of the particles under consideration.

One way to arrive at a mean-field description is by using the Lagrangian reduction technique [**62**], which exploits the Hamiltonian structure of Eq. (40). The Lagrangian of Eq. (40) is given by [**59**]

$$L=\int_{-\infty}^{\infty}\left\{i\frac{\hbar}{2}\left(\Psi\frac{\partial\Psi^*}{\partial t}-\Psi^*\frac{\partial\Psi}{\partial t}\right)+\frac{\hbar^2}{2m}\left|\nabla^N\Psi\right|^2+\right.$$
$$\left.\sum_{i=1}^{N}\left(\sum_{j\neq i}^{N}W(\mathbf{x}_i-\mathbf{x}_j)+V(\mathbf{x}_i)\right)|\Psi|^2\right\}d\mathbf{x}_1\cdots d\mathbf{x}_N\,. \tag{41}$$

The Hartree-Fock approximation (as used in [**59**]) for bosonic particles uses the separated wave function ansatz

$$\Psi = \psi_1(\mathbf{x}_1, t)\psi_2(\mathbf{x}_2, t)\cdots\psi_N(\mathbf{x}_N, t) \tag{42}$$

where each one-particle wave function $\psi(\mathbf{x}_i)$ is assumed to be normalized so that $\langle\psi(\mathbf{x}_i)|\psi(\mathbf{x}_i)\rangle^2 = 1$. Since identical particles are being considered,

$$\psi_1 = \psi_2 = \ldots = \psi_N = \psi, \tag{43}$$

enforcing total symmetry of the wave function. Note that for the case of BECs, assumption (42) is approximate if the temperature is not identically zero.

Integrating Eq. (41) using (42) and (43) and taking the variational derivative with respect to $\psi(\mathbf{x}_i)$ results in the Euler-Lagrange equation [**62**]

$$\begin{aligned} i\hbar\frac{\partial\psi(\mathbf{x},t)}{\partial t} = \; & -\frac{\hbar^2}{2m}\Delta\psi(\mathbf{x},t) + V(\mathbf{x})\psi(\mathbf{x},t) + \\ & (N-1)\psi(\mathbf{x},t)\int_{-\infty}^{\infty} W(\mathbf{x}-\mathbf{y})|\psi(\mathbf{y},t)|^2 d\mathbf{y}. \end{aligned} \tag{44}$$

Here, $\mathbf{x} = \mathbf{x}_i$, and Δ is the one-particle Laplacian in three dimensions. The Euler-Lagrange equation (44) is identical for all $\psi(\mathbf{x}_i, t)$. Equation (44) describes the nonlinear, nonlocal, mean-field dynamics of the wave function $\psi(\mathbf{x}, t)$ under the standard assumptions (42) and (43) of Hartree-Fock theory [**59**]. The coefficient of $\psi(\mathbf{x}, t)$ in the last term in Eq. (44) represents the effective potential acting on $\psi(\mathbf{x}, t)$ due to the presence of the other particles.

At this point, it is common to make an assumption on the functional form of the interaction potential $W(\mathbf{x}-\mathbf{y})$. This is done to render Eq. (44) analytically and numerically tractable. Although the qualitative features of this functional form may be available, for instance from experiment, its quantitative details are rarely known. One convenient assumption in the case of short-range potential interactions is $W(\mathbf{x} - \mathbf{y}) = \kappa\delta(\mathbf{x} - \mathbf{y})$ where δ is the Dirac delta function. This leads to the Gross-Pitaevskii [**60, 61**] mean-field description:

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\Delta\psi + \beta|\psi|^2\psi + V(\mathbf{x})\psi, \tag{45}$$

where $\beta = (N-1)\kappa$ reflects whether the interaction is repulsive ($\beta > 0$) or attractive ($\beta < 0$). The above string of assumptions is difficult to physically justify. Nevertheless, Lieb and Seiringer [**63**] show that Eq. (45) is the correct asymptotic description in the dilute-gas limit. In this limit, Eqs. (44) and (45) are asymptotically equivalent. Thus, although the nonlocal Eq. (44) is in no way a more valid model than the local Eq. (45), it can be interpreted as a perturbation to the local Eq. (45).

Project and Application:

In what follows, you will be asked to consider the dynamics of a BEC in one, two and three dimensions. The potential in (45) determines much of the allowed or enforced dynamics in the system.

One-dimensional condensate: In this initial investigation, the BEC cavity is assumed to trapped in a cigar shaped trapped where the longitudinal direction is orders of magnitude longer than the transverse direction. Therefore, it is usually assumed that the localized mode-locked pulses are essentially in an infinite domain. To approximate this, one can simply use a large domain and an FFT based solution method. Be sure to pick a domain large enough so that boundary effects do not play a role. The governing equations (45) is then reduced to nondimensional form

$$i\frac{\partial\psi}{\partial t} + \frac{1}{2}\frac{\partial^2\psi}{\partial x^2} + \alpha|\psi|^2\psi - \left[V_0\sin^2(\omega(x-\bar{x})) + V_1x^2\psi\right] = 0, \tag{46}$$

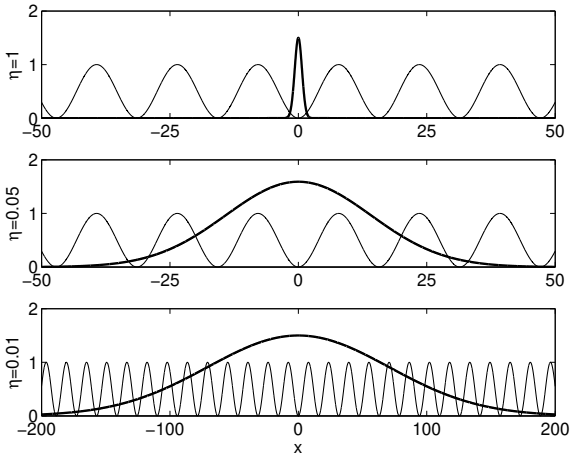

FIGURE 12. Localized condensate (bold line) and applied periodic potential (light line) for various ratios of period of the potential to a Gaussian input $\psi = 1.5\sqrt{\eta}\exp(-\eta^2 x^2)$. Here we choose $\omega = 0.2$ with the value of η given on the left of each panel. [from N. H. Berry and J. N. Kutz, *Dynamics of Bose-Einstein condensates under the influence of periodic and harmonic potentials*, Physical Review E **75**, 036214 (2007) ©APS]

where a periodic (sinusoidal) and harmonic (parabolic) potential have been assumed to be acting on the system.

(a) Investigate the dynamics of the system as a function of the parameters V_0 and V_1 and with $\alpha = 1$. Assume a localized initial condition, something like a Gaussian that can be generically imposed off-center ($x \neq 0$) from the harmonic trap (See, for instance, Fig. 12)). Also investigate the three scenarios where the width of the pulse is much wider, about the same width and narrower than the period of the oscillatory potential. Consider the stability of the BEC and the amplitude–width dynamics as the condensate propagates in time and slides along the combination of periodic and harmonic traps.

(b) For the case of a purely periodic potential, $V_1 = 0$ and $V_0 \neq 0$, consider wether standing wave solutions or periodic solutions are permissible. Assumed an initial periodic sinusoidal solution and investigate both an attractive ($\alpha = 1$) or repulsive condensate ($\alpha = -1$).

Two- and Three-dimensional condensate lattices: An interesting BEC configuration to consider is when a periodic lattice potential is imposed on the system. In two-dimensions, this gives the governing equations

$$i\frac{\partial\psi}{\partial t} + \frac{1}{2}\frac{\partial^2\psi}{\partial x^2} + \frac{1}{2}\frac{\partial^2\psi}{\partial y^2} + \alpha|\psi|^2\psi - \big[A_1\sin^2(\omega_1 x) + B_1\big]\big[A_2\sin^2(\omega_2 y) + B_2\big] = 0, \tag{47}$$

where A_i, ω_i and B_i completely characterize the nature of the periodic potential.

(c) For a symmetric lattice with $A_1 = A_2$, $B_1 = B_2$ and $\omega_1 = \omega_2$, consider the dynamics of both localized and periodic solutions on the lattice for both attractive and repulsive condensates. Are there relatively stable BEC configurations that allow for a long-time stable configuration of the BEC (See, for instance, Fig. 13). Consider solutions that are localized in the individual troughs of the periodic potential as well as those that localized on the peaks of the potential.

(d) Explore the dynamics of the lattice when symmetry is broken in the x and y direction.

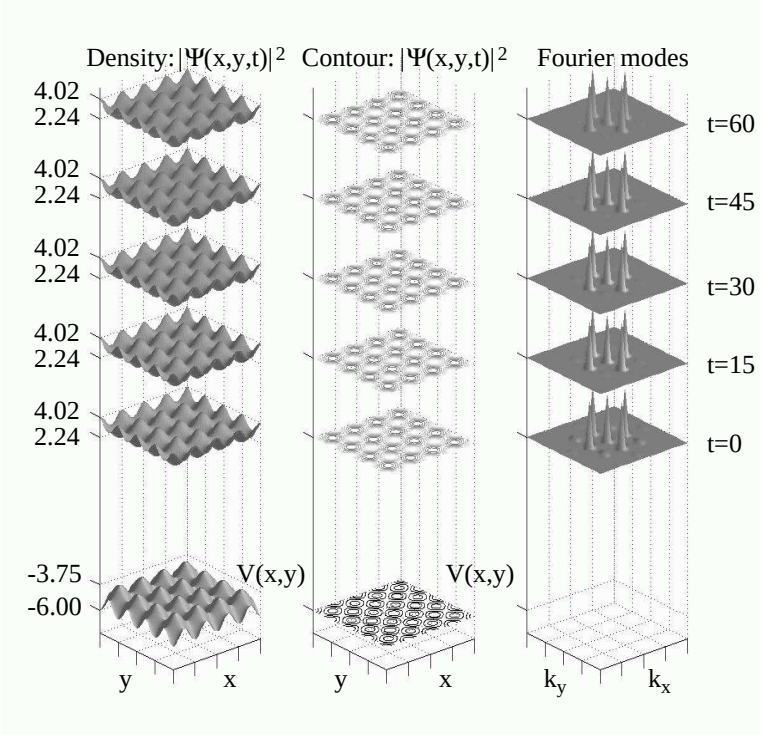

FIGURE 13. Evolution of stable two-dimensional BEC on a periodic lattice.

(e) Generalize (47) to three dimensions and consider (c) and (d) in the context of a three dimensional lattice structure. Use the **isosurface** and **slice** plotting routines to demonstrate the evolution dynamics.

8.3. Introduction to Reaction-Diffusion Systems. To begin a discussion of the need for generic reaction-diffusion equations, we consider a set of simplified models relating to *predator-prey* population dynamics. These models consider the interaction of two species: predators and their prey. It should be obvious that such species will have significant impact on one another. In particular, if there is an abundance of prey, then the predator population will grow due to the surplus of food. Alternatively, if the prey population is low, then the predators may die off due to starvation.

To model the interaction between these species, we begin by considering the predators and prey in the absence of any interaction. Thus the prey population (denoted by $x(t)$) is governed by

$$\frac{dx}{dt} = ax \tag{48}$$

where $a > 0$ is a net growth constant. The solution to this simple differential equation is $x(t) = x(0)\exp(at)$ so that the population grows without bound. We have assumed here that the food supply is essentially unlimited for the prey so that the unlimited growth makes sense since there is nothing to kill off the population.

Likewise, the predators can be modeled in the absence of their prey. In this case, the population (denoted by $y(t)$) is governed by

$$\frac{dy}{dt} = -cy \tag{49}$$

where $c > 0$ is a net decay constant. The reason for the decay is that the population starves off since there is no food (prey) to eat.

We now try to model the interaction. Essentially, the interaction must account for the fact the the predators eat the prey. Such an interaction term can result in the following system:

$$\frac{dx}{dt} = ax - \alpha xy \tag{50a}$$

$$\frac{dy}{dt} = -cx + \alpha xy\,, \tag{50b}$$

where $\alpha > 0$ is the interaction constant. Note that α acts as a decay to the prey population since the predators will eat them, and as a growth term to the predators since they now have a food supply. These nonlinear and autonomous equations are known as the *Lotka–Volterra equations*. There are two fundamental limitations of this model: the interaction is only heuristic in nature and there is no spatial dependence. Thus the validity of this simple modeling is certainly questionable.

Spatial Dependence. One way to model the dispersion of a species in a given domain is by assuming the dispersion is governed by a diffusion process. If in addition we assume that the prey population can saturate at a given level, then the governing population equations are

$$\frac{\partial x}{\partial t} = a\left(x - \frac{x^2}{k}\right) - \alpha xy + D_1\nabla^2 x \tag{51a}$$

$$\frac{\partial y}{\partial t} = -cx + \alpha xy + D_2\nabla^2 y\,, \tag{51b}$$

which are known as the *modified Lotka-Volterra* equations. It includes the species interaction with saturation, i.e. the *reaction* terms, and spatial spreading through diffusion, i.e. the *diffusion* term. Thus it is a simple reaction-diffusion equation.

Along with the governing equations, boundary conditions must be specified. A variety of conditions may be imposed, these include periodic boundaries, clamped boundaries such that x and y are known at the boundary, flux boundaries in which $\partial x/\partial n$ and $\partial y/\partial n$ are known at the boundaries, or some combination of flux and clamped. The boundaries are significant in determining the ultimate behavior in the system.

Spiral Waves. One of the many phenomena which can be observed in reaction-diffusion systems is spiral waves. An excellent system for studying this phenomena is the Fitzhugh-Nagumo model which provides a heuristic description of an excitable nerve potential:

$$\frac{\partial u}{\partial t} = u(a-u)(1-u) - v + D\nabla^2 u \tag{52a}$$

$$\frac{\partial v}{\partial t} = bu - \gamma v\,, \tag{52b}$$

where a, D, b, and γ are tunable parameters.

A basic understanding of the spiral wave phenomena can be achieved by considering this problem in the absence of the diffusion. Thus the reaction terms alone give

$$\frac{\partial u}{\partial t} = u(a-u)(1-u) - v \tag{53a}$$

$$\frac{\partial v}{\partial t} = bu - \gamma v\,. \tag{53b}$$

This reduces to a system of differential equations for which the fixed points can be considered. Fixed points occur when $\partial u/\partial t = \partial v/\partial t = 0$. The three fixed points for this system are given by

$$(u, v) = (0, 0) \tag{54a}$$

$$(u, v) = (u_\pm, (a-u_\pm)(1-u_\pm)u_\pm) \tag{54b}$$

where $u_\pm = [(a+1) \pm ((a+1)^2 - 4(a - b/\gamma))^{1/2}]/2$.

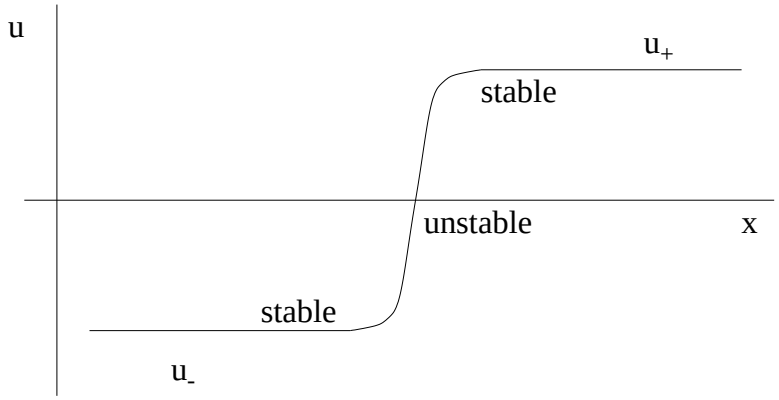

FIGURE 14. Front solution which connects the two stable branch of solutions $u_\pm$ through the unstable solution $u = 0$.

The stability of these three fixed points can be found by linearization [**29**]. In particular, consider the behavior near the steady-state solution $u = v = 0$. Thus let

$$u = 0 + \tilde{u} \tag{55a}$$

$$v = 0 + \tilde{v} \tag{55b}$$

where $\tilde{u}, \tilde{v} \ll 1$. Plugging in and discarding higher order terms gives the linearized equations

$$\frac{\partial \tilde{u}}{\partial t} = a\tilde{u} - \tilde{v} \tag{56a}$$

$$\frac{\partial \tilde{v}}{\partial t} = b\tilde{u} - \gamma\tilde{v}\,. \tag{56b}$$

This can be written as the linear system

$$\frac{d\mathbf{x}}{dt} = \begin{pmatrix} a & -1 \\ b & -\gamma \end{pmatrix} \mathbf{x}\,. \tag{57}$$

Assuming a solution of the form $\mathbf{x} = \mathbf{v}\exp(\lambda t)$ results in the eigenvalue problem

$$\begin{pmatrix} a-\lambda & -1 \\ b & -\gamma-\lambda \end{pmatrix} \mathbf{v} = 0 \tag{58}$$

which has the eigenvalues

$$\lambda_\pm = \frac{1}{2}\left[(a-\gamma) \pm \sqrt{(a-\gamma)^2 + 4(b-a\gamma)}\right]\,. \tag{59}$$

In the case where $b - a\gamma > 0$, the eigenvalues are purely real with one positive and one negative eigenvalue, i.e. it is a saddle node. Thus the steady-state solution in this case is unstable.

Further, if the condition $b - a\gamma > 0$ holds, then the two remaning fixed points occur for $u_- < 0$ and $u_+ > 0$. The stability of these points may also be found. For the parameter restrictions considered here, these two remaining points are found to be stable upon linearization.

The question which can then naturally arise: if the two stable solutions $u_\pm$ are connected, how will the front between the two stable solutions evolve? The scenario is depicted in one dimension in Fig. 14 where the two stable $u_\pm$ branches are connected through the unstable $u = 0$ solution. Not just an interesting mathematical phenomena, spiral waves are also exhibited in nature. Figure 15 illustrates an experimental observation in which spiral waves are exhibited in rat ventricular cells. Spiral waves are seen to be naturally generated in this and many other systems and are of natural interest for analytic study.

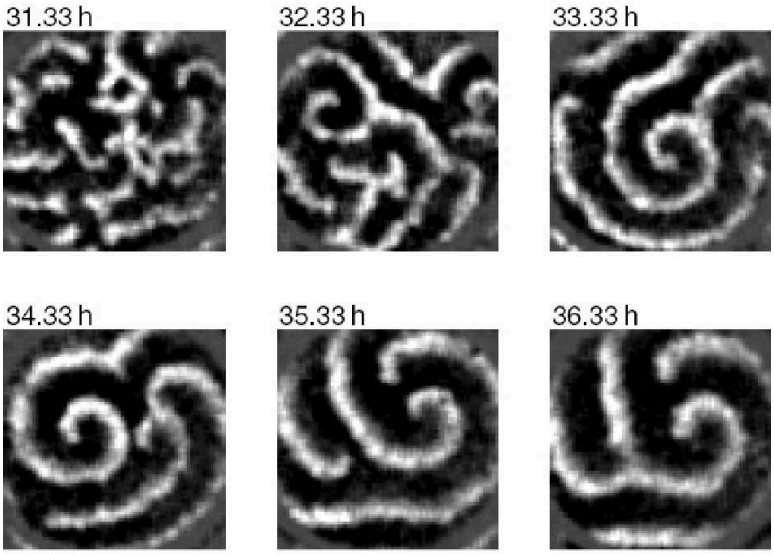

FIGURE 15. Experimental observations of typical spatiotemporal evolution of spiral waves in the early stage of a primary culture of dissociated rat ventricular cells. Few spiral cores survive at the end. [from S.-J. Woo, J. H. Hong, T. Y. Kim, B. W. Bae and K. J. Lee, *Spiral wave drift and complex-oscillatory spiral waves caused by heterogeneities in two-dimensional in vitro cardiac tissues*, New Journal of Physics **10**, 015005 (2008) ©IOP]

The $\lambda - \omega$ Reaction-Diffusion System. The general reaction-diffusion system can be written in the vector form

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{f}(\mathbf{u}) + D\nabla^2 \mathbf{u} \tag{60}$$

where the diffusion is proportional to the parameter D and the reaction terms are given by $\mathbf{f}(\mathbf{u})$. The specific case we consider is the $\lambda - \omega$ system which takes the form

$$\frac{\partial}{\partial t}\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \lambda(A) & -\omega(A) \\ \omega(A) & \lambda(A) \end{pmatrix}\begin{pmatrix} u \\ v \end{pmatrix} + D\nabla^2 \begin{pmatrix} u \\ v \end{pmatrix} \tag{61}$$

where $A = u^2 + v^2$. Thus the nonlinearity is cubic in nature. This allows for the possibility of supporting three steady-state solutions as in the Fitzhugh-Nagumo model which led to spiral wave behavior. The spirals to be investigated in this case can have one, two, three or more arms. An example of a spiral wave initial condition is given by

$$u_0(x, y) = \tanh|\mathbf{r}|\cos(m\theta - |\mathbf{r}|) \tag{62a}$$

$$v_0(x, y) = \tanh|\mathbf{r}|\sin(m\theta - |\mathbf{r}|) \tag{62b}$$

where $|\mathbf{r}| = \sqrt{x^2 + y^2}$ and $\theta = x + iy$. The parameter m determines the number of arms on the spiral. The stability of the spiral evolution under perturbation and in different parameter regimes is essential in understanding the underlying dynamics of the system. A wide variety of boundary conditions can also be applied. Namely, periodic, clamped, no-flux, or mixed. In each case, the boundaries have significant impact on the resulting evolution as the boundary effects creep into the middle of the computational domain.

Project and Application:

Consider the $\lambda - \omega$ reaction-diffusion system

$$U_t = \lambda(A)U - \omega(A)V + D_1\nabla^2 U \tag{63a}$$

$$V_t = \omega(A)U + \lambda(A)V + D_2\nabla^2 V \tag{63b}$$

where $A^2 = U^2 + V^2$ and $\nabla^2 = \partial_x^2 + \partial_y^2$.

Boundary Conditions: Consider the two boundary conditions in the $x-$ and $y-$directions:

- Periodic
- No flux: $\partial U/\partial n = \partial V/\partial n = 0$ on the boundaries

Numerical Integration Procedure: The following numerical integration procedures are to be investigated and compared.

- For the periodic boundaries, transform the right-hand with FFTs
- For the no flux boundaries, use the Chebychev polynomials

You can advance the solution in time using ode45 (or any one of the built in ODE solvers from the MATLAB suite).

Initial Conditions Start with spiral initial conditions in U and V.

```
[X,Y]=meshgrid(x,y);
m=1; % number of spirals
u=tanh(sqrt(X.^2+Y.^2)).*cos(m*angle(X+i*Y)-(sqrt(X.^2+Y.^2)));
v=tanh(sqrt(X.^2+Y.^2)).*sin(m*angle(X+i*Y)-(sqrt(X.^2+Y.^2)));
```

Consider the specific $\lambda - \omega$ system:

$$\lambda(A) = 1 - A^2 \tag{64a}$$

$$\omega(A) = -\beta A^2 \tag{64b}$$

Look to construct one- and two-armed spirals for this system. Also investigate when the solutions become unstable and "chaotic" in nature. Investigate the system for all three boundary conditions. Note $\beta > 0$ and further consider the diffusion to be not too large, but big enough to kill off Gibbs phenomena at the boundary, i.e. $D_1 = D_2 = 0.1$.

CHAPTER 12

Finite Element Methods

The numerical methods considered thus far, i.e. finite difference and spectral, are powerful tools for solving a wide variety of physical problems. However, neither method is well suited for complicated boundary configurations or complicated boundary conditions associated with such domains. The finite element method is ideally suited for complex boundaries and very general boundary conditions. This method, although perhaps not as fast as finite difference and spectral, is instrumental in solving a wide range of problems which is beyond the grasp of other techniques.

1. Finite Element Basis

The finite difference method approximates a solution by discretizing and solving the matrix problem $\mathbf{Ax} = \mathbf{b}$. Spectral methods use the FFT to expand the solution in global sine and consine basis functions. Finite elements are another form of finite difference in which the approximation to the solution is made by interpolating over patches of our solution region. Ultimately, to find the solution we will once again solve a matrix problem $\mathbf{Ax} = \mathbf{b}$. Five essential steps are required in the finite element method:

- Discretization of the computational domain
- Selection of the interpolating functions
- Derivation of characteristic matrices and vectors
- Assembly of characteristic matrices and vectors
- Solution of the matrix problem $\mathbf{Ax} = \mathbf{b}$.

As will be seen, each step in the finite element method is mathematically significant and relatively challenging. However, there are a large number of commercially available packages that take care of implementing the above ideas. Keep in mind that the primary reason to develop this technique is boundary conditions: both complicated domains and general boundary conditions.

1.1. Domain discretization. Finite element domain discretization usually involves the use of a commercial package. Two commonly used packages are the python PDE Toolbox and FEMLAB (which is built on the python platform). Writing your own code to generate an unstructured computational grid is a difficult task and well beyond the scope of this book. Thus we will use commercial packages to generate the computational mesh necessary for a given problem. The key idea is to discretize the domain with triangular elements (or another appropriate element function). Figure 1 shows the discretization of a domain with complicated boundaries inside the computationally relevant region. Note that the triangular discretization is such that it adaptively sizes the triangles so that higher resolution is automatically in place where needed. The key features of the discretization are as follows:

- The width and height of all discretization triangles should be similar.
- All shapes used to span the computational domain should be approximated by polygons.
- A commercial package is almost always used to generate the grid unless you are doing research in this area.

1.2. Interpolating functions. Each element in the finite element basis relies on a piecewise approximation. Thus the solution to the problem is approximated by a simple function in each

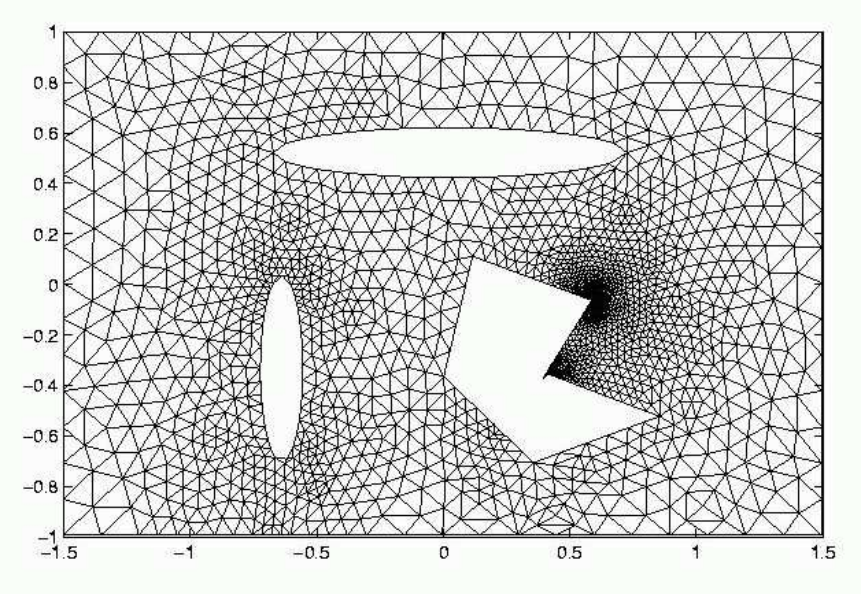

FIGURE 1. Finite element triangular discretization provided by the MATLAB PDE Toolbox. Note the increased resolution near corners and edges.

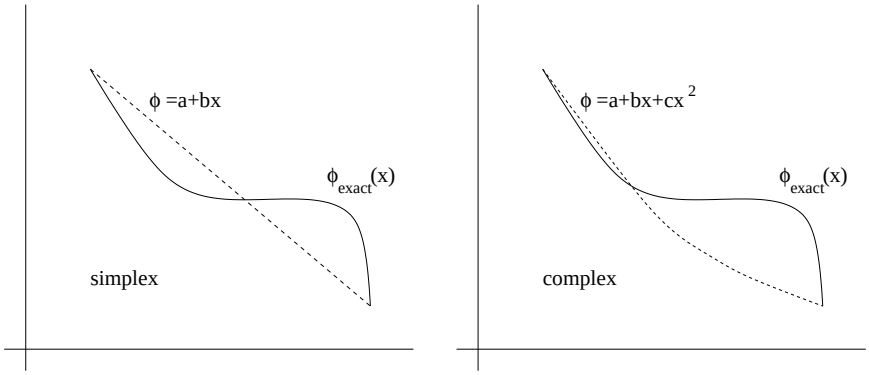

FIGURE 2. Finite element discretization for simplex and complex finite elements. Essentially the finite elements are approximating polynomials.

triangular (or polygonal) element. As might be expected, the accuracy of the scheme depends upon the choice of the approximating function chosen.

Polynomials are the most commonly used interpolating functions. The simpler the function, the less computationally intensive. However, accuracy is usually increased with the use of higher order polynomials. Three groups of elements are usually considered in the basic finite element scheme:

- Simplex elements: linear polynomials are used
- Complex elements: higher order polynomials are used
- Multiplex elements: rectangles are used instead of triangles.

An example of each of the three finite elements is given in Figs. 2 and 3 for one-dimensional and two-dimensional finite elements.

1.3. 1D simplex. To consider the finite element implementation, we begin by considering the one-dimensional problem. Figure 4 shows the linear polynomial used to approximate the solution

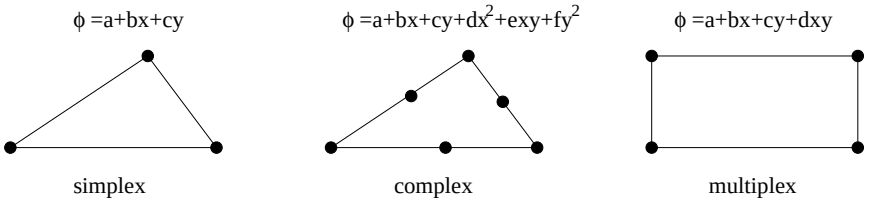

FIGURE 3. Finite element discretization for simplex, complex, and multiplex finite elements in two-dimensions. The finite elements are approximating surfaces.

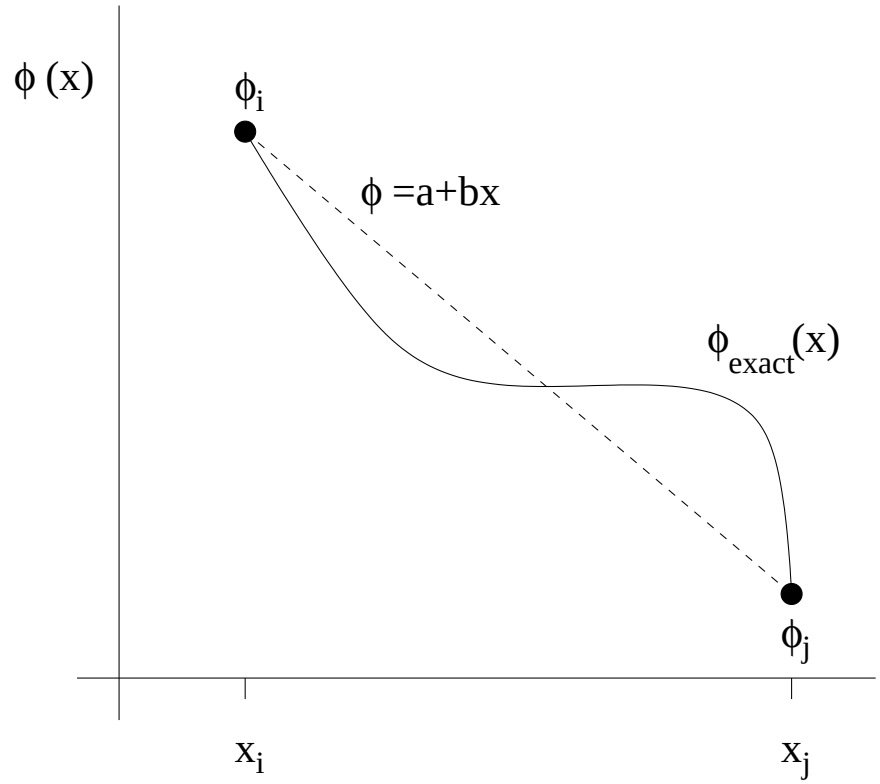

FIGURE 4. One dimensional approximation using finite elements.

between points x_i and x_j. The approximation of the function is thus

$$\phi(x) = a + bx = (1 \;\; x) \begin{pmatrix} a \\ b \end{pmatrix}. \tag{65}$$

The coefficients a and b are determined by enforcing that the function goes through the end points. This gives

$$\phi_i = a + bx_i \tag{66a}$$
$$\phi_j = a + bx_j \tag{66b}$$

which is a 2×2 system for the unknown coefficients. In matrix form this can be written

$$\phi = \mathbf{A}\mathbf{a} \;\rightarrow\; \phi = \begin{pmatrix} \phi_i \\ \phi_j \end{pmatrix}, \; \mathbf{A} = \begin{pmatrix} 1 & x_i \\ 1 & x_j \end{pmatrix}, \; \mathbf{a} = \begin{pmatrix} a \\ b \end{pmatrix}. \tag{67}$$

Solving for $\mathbf{a}$ gives

$$\mathbf{a} = \mathbf{A}^{-1}\phi = \frac{1}{x_j - x_i}\begin{pmatrix} x_j & -x_i \\ -1 & 1 \end{pmatrix}\begin{pmatrix} \phi_i \\ \phi_j \end{pmatrix} = \frac{1}{l}\begin{pmatrix} x_j & -x_i \\ -1 & 1 \end{pmatrix}\begin{pmatrix} \phi_i \\ \phi_j \end{pmatrix} \tag{68}$$

where $l = x_j - x_i$. Recalling that $\phi = (1 \;\; x)\mathbf{a}$ then gives

$$\phi = \frac{1}{l}(1 \;\; x)\begin{pmatrix} x_j & -x_i \\ -1 & 1 \end{pmatrix}\begin{pmatrix} \phi_i \\ \phi_j \end{pmatrix}. \tag{69}$$

Multiplying this expression out yields the approximate solution

$$\phi(x) = N_i(x)\phi_i + N_j(x)\phi_j \tag{70}$$

where $N_i(x)$ and $N_j(x)$ are the Lagrange polynomial coefficients [**7**] (shape functions)

$$N_i(x) = \frac{1}{l}(x_j - x) \tag{71a}$$

$$N_j(x) = \frac{1}{l}(x - x_i). \tag{71b}$$

Note the following properties: $N_i(x_i) = 1, N_i(x_j) = 0$ and $N_j(x_i) = 0, N_j(x_j) = 1$. This completes the generic construction of the polynomial which approximates the solution in one dimension.

1.4. 2D simplex. In two dimensions, the construction becomes a bit more difficult. However, the same approach is taken. In this case, we approximate the solution over a region with a plane (see Fig. 5)

$$\phi(x) = a + bx + cy. \tag{72}$$

The coefficients a, b and c are determined by enforcing that the function goes through the end points. This gives

$$\phi_i = a + bx_i + cy_i \tag{73a}$$

$$\phi_j = a + bx_j + cy_j \tag{73b}$$

$$\phi_k = a + bx_k + cy_k \tag{73c}$$

which is now a 3×3 system for the unknown coefficients. In matrix form this can be written

$$\phi = \mathbf{A}\mathbf{a} \;\rightarrow\; \phi = \begin{pmatrix} \phi_i \\ \phi_j \\ \phi_k \end{pmatrix}, \mathbf{A} = \begin{pmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_k & y_k \end{pmatrix}, \mathbf{a} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}. \tag{74}$$

The geometry of this problem is reflected in Fig. 5 where each of the points of the discretization triangle is illustrated.

Solving for $\mathbf{a}$ gives

$$\mathbf{a} = \mathbf{A}^{-1}\phi = \frac{1}{2S}\begin{pmatrix} x_jy_k - x_ky_j & x_ky_i - x_iy_k & x_iy_j - x_jy_i \\ y_j - y_k & y_k - y_i & y_i - y_j \\ x_k - x_j & x_i - x_k & x_j - x_i \end{pmatrix}\begin{pmatrix} \phi_i \\ \phi_j \\ \phi_k \end{pmatrix} \tag{75}$$

where S is the area of the triangle projected onto the $x-y$ plane (see Fig. 5). Recalling that $\phi = \mathbf{A}\mathbf{a}$ gives

$$\phi(x) = N_i(x,y)\phi_i + N_j(x,y)\phi_j + N_k(x,y)\phi_k \tag{76}$$

where $N_i(x,y)$, $N_j(x,y)$, $N_k(x,y)$ are the Lagrange polynomial coefficients [**7**] (shape functions)

$$N_i(x,y) = \frac{1}{2S}\left[(x_jy_k - x_ky_j) + (y_j - y_k)x + (x_k - x_j)y\right] \tag{77a}$$

$$N_j(x,y) = \frac{1}{2S}\left[(x_ky_i - x_iy_k) + (y_k - y_i)x + (x_i - x_k)y\right] \tag{77b}$$

$$N_k(x,y) = \frac{1}{2S}\left[(x_iy_j - x_jy_i) + (y_i - y_j)x + (x_j - x_i)y\right]. \tag{77c}$$

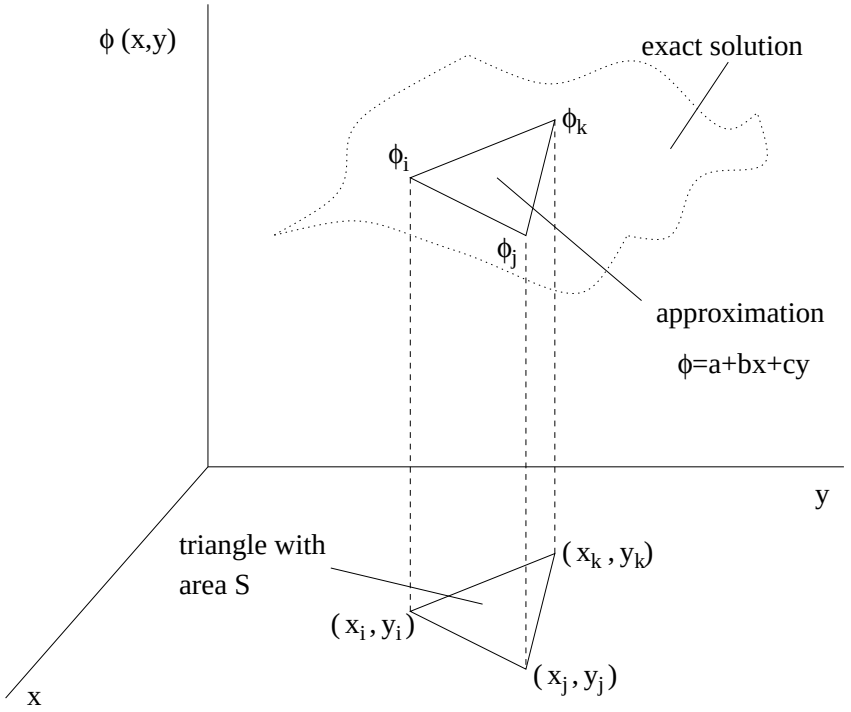

FIGURE 5. Two dimensional triangular approximation of the solution using the finite element method.

Note the following end point properties: $N_i(x_i, y_i) = 1$, $N_j(x_j, y_j) = N_k(x_k, y_k) = 0$ and $N_j(x_j, y_j) = 1$, $N_j(x_i, y_i) = N_j(x_k, y_k) = 0$ and $N_k(x_k, y_k) = 1$, $N_k(x_i, y_i) = N_k(x_j, y_j) = 0$. This completes the generic construction of the polynomial which approximates the solution in one dimension, and the generic construction of the surface which approximates the solution in two dimensions.

2. Discretizing with Finite Elements and Boundaries

In the previous section, an outline was given for the construction of the polynomials which approximate the solution over a finite element. Once constructed, however, they must be used to solve the physical problem of interest. The finite element solution method relies on solving the governing set of partial differential equations in the *weak formulation* of the problem, i.e. the integral formulation of the problem. This is because we will be using linear interpolation pieces whose derivatives don't necessarily match accross elements. In the integral formulation, this does not pose a problem.

We begin by considering the elliptic partial differential equation

$$\frac{\partial}{\partial x}\left(p(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + r(x,y)u = f(x,y) \tag{1}$$

where, over part of the boundary,

$$u(x,y) = g(x,y) \quad \text{on } S_1 \tag{2}$$

and

$$p(x,y)\frac{\partial u}{\partial x}\cos\theta_1 + q(x,y)\frac{\partial u}{\partial y}\cos\theta_2 + g_1(x,y)u = g_2(x,y) \quad \text{on } S_2. \tag{3}$$

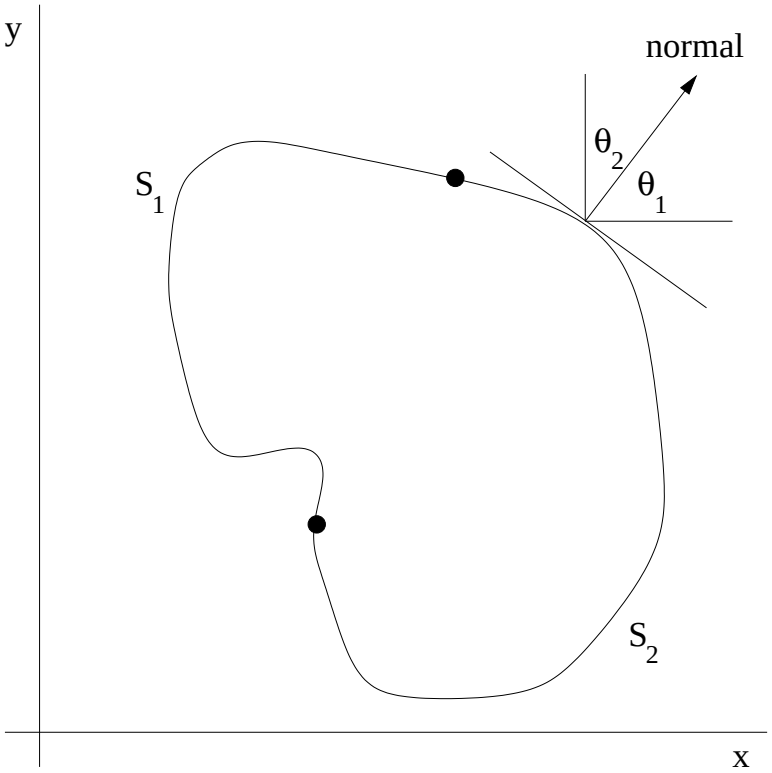

FIGURE 6. Computational domain of interest which has the two domain regions S_1 and S_2.

The boundaries and domain are illustrated in Fig. 6. Note that the normal derivative determines the angles θ_1 and θ_2. A domain such as this would be very difficult to handle with finite difference techniques. And further, because of the boundary conditions, spectral methods cannot be implemented. Only the finite element method renders the problem tractable.

2.1. The variational principle. To formulate the problem correctly for the finite element method, the governing equations and their associated boundary conditions are recast in an integral form. This recasting of the problem involves a variational principle. Although we will not discuss the calculus of variations here [**64**], the highlights of this method will be presented.

The variational method expresses the governing partial differential as an integral which is to be minimized. In particular, the functional to be minimized is given by

$$I(\phi) = \iiint_V F\left(\phi, \frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\phi}{\partial z}\right) dV + \iint_S g\left(\phi, \frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\phi}{\partial z}\right) dS, \tag{4}$$

where a three-dimensional problem is being considered. The derivation of the functions F, which captures the governing equation, and g, which captures the boundary conditions, follow the Euler–Lagrange equations for minimizing variations:

$$\frac{\delta F}{\delta\phi} = \frac{\partial}{\partial x}\left(\frac{\partial F}{\partial(\phi_x)}\right) + \frac{\partial}{\partial y}\left(\frac{\partial F}{\partial(\phi_y)}\right) + \frac{\partial}{\partial z}\left(\frac{\partial F}{\partial(\phi_z)}\right) - \frac{\partial F}{\partial\phi} = 0. \tag{5}$$

This is essentially a generalization of the concept of the zero derivative which minimizes a function. Here we are minimizing a functional.

For our governing elliptic problem (1) with boundary conditions given by those of S_2 (3), we find the functional $I(u)$ to be given by

$$\begin{aligned} I(u) &= \frac{1}{2}\int\int_D dxdy\left[p(x,y)\left(\frac{\partial u}{\partial x}\right)^2 + q(x,y)\left(\frac{\partial u}{\partial y}\right)^2 - r(x,y)u^2 + 2f(x,y)u\right] \\ &\quad + \int_{S_2} dS\left[-g_2(x,y)u + \frac{1}{2}g_1(x,y)u^2\right]. \end{aligned} \tag{6}$$

Thus the interior domain over which we integrate D has the integrand

$$I_D = \frac{1}{2}\left[p(x,y)\left(\frac{\partial u}{\partial x}\right)^2 + q(x,y)\left(\frac{\partial u}{\partial y}\right)^2 - r(x,y)u^2 + 2f(x,y)u\right]. \tag{7}$$

This integrand was derived via variational calculus as the minimization over the functional space of interest. To confirm this, we apply the variational derivative as given by (5) and find

$$\begin{aligned} \frac{\delta I_D}{\delta u} &= \frac{\partial}{\partial x}\left(\frac{\partial I_D}{\partial(u_x)}\right) + \frac{\partial}{\partial y}\left(\frac{\partial I_D}{\partial(u_y)}\right) - \frac{\partial I_D}{\partial u} = 0 \\ &= \frac{\partial}{\partial x}\left(\frac{1}{2}p(x,y)\cdot 2\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{1}{2}q(x,y)\cdot 2\frac{\partial u}{\partial y}\right) - \left(-\frac{1}{2}r(x,y)\cdot 2u + f(x,y)\right) \\ &= \frac{\partial}{\partial x}\left(p(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + r(x,y)u - f(x,y) = 0. \end{aligned} \tag{8}$$

Upon rearranging, this gives back the governing equation (1)

$$\frac{\partial}{\partial x}\left(p(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + r(x,y)u = f(x,y). \tag{9}$$

A similar procedure can be followed on the boundary terms to derive the appropriate boundary conditions (2) or (3). Once the integral formulation has been achieved, the finite element method can be implemented with the following key ideas:

- The solution is expressed in the *weak* (integral) form since the linear (simplex) interpolating functions give that the second derivates (e.g. ∂_x^2 and ∂_y^2) are zero. Thus the elliptic operator (1) would have no contribution from a finite element of the form $\phi = a_1 + a_2x + a_3y$. However, in the integral formulation, the second derivative terms are proportional to $(\partial u/\partial x)^2 + (\partial u/\partial y)^2$ which gives $\phi = a_2^2 + a_3^2$.
- A solution is sought which takes the form

$$u(x,y) = \sum_{i=1}^{m} \gamma_i \phi_i(x,y) \tag{10}$$

  where $\phi_i(x,y)$ are the linearly independent piecewise-linear polynomials and $\gamma_1, \gamma_2, \ldots, \gamma_m$ are constants.
  - $\gamma_{n+1}, \gamma_{n+2}, \ldots, \gamma_m$ ensure that the boundary conditions are satisfied, i.e. those elements which touch the boundary must meet certain restrictions.
  - $\gamma_1, \gamma_2, \ldots, \gamma_n$ ensure that the integrand $I(u)$ in the interior of the computational domain is minimized, i.e. $\partial I/\partial\gamma_i = 0$ for $i = 1, 2, 3, \ldots, n$.

2.2. Solution method. We begin with the governing equation in integral form (6) and assume an expansion of the form (10). This gives

$$\begin{aligned} I(u) &= I\left(\sum_{i=1}^{m}\gamma_i\phi_i(x,y)\right) \\ &= \frac{1}{2}\iint_D dxdy\left[p(x,y)\left(\sum_{i=1}^{m}\gamma_i\frac{\partial\phi_i}{\partial x}\right)^2 + q(x,y)\left(\sum_{i=1}^{m}\gamma_i\frac{\partial\phi_i}{\partial y}\right)^2\right. \\ &\qquad \left. - r(x,y)\left(\sum_{i=1}^{m}\gamma_i\phi_i(x,y)\right)^2 + 2f(x,y)\left(\sum_{i=1}^{m}\gamma_i\phi_i(x,y)\right)\right] \\ &\quad + \int_{S_2} dS\left[-g_2(x,y)\sum_{i=1}^{m}\gamma_i\phi_i(x,y) + \frac{1}{2}g_1(x,y)\left(\sum_{i=1}^{m}\gamma_i\phi_i(x,y)\right)^2\right]. \end{aligned} \tag{11}$$

This completes the expansion portion.

The functional $I(u)$ must now be differentiated with respect to all the interior points and minimized. This involves differentiating the above with respect to γ_i where $i = 1, 2, 3, \ldots, n$. This results in the rather complicated expression

$$\begin{aligned} \frac{\partial I}{\partial\gamma_j} &= \iint_D dxdy\left[p(x,y)\sum_{i=1}^{m}\gamma_i\frac{\partial\phi_i}{\partial x}\frac{\partial\phi_j}{\partial x} + q(x,y)\sum_{i=1}^{m}\gamma_i\frac{\partial\phi_i}{\partial y}\frac{\partial\phi_j}{\partial y}\right. \\ &\qquad \left. - r(x,y)\sum_{i=1}^{m}\gamma_i\phi_i\phi_j + f(x,y)\phi_j\right] \\ &\quad + \int_{S_2} dS\left[-g_2(x,y)\phi_j + g_1(x,y)\sum_{i=1}^{m}\gamma_i\phi_i\phi_j\right] = 0. \end{aligned} \tag{12}$$

Term by term, this results in an expression for the γ_i given by

$$\begin{aligned} &\sum_{i=1}^{m}\left\{\iint_D dxdy\left[p\frac{\partial\phi_i}{\partial x}\frac{\partial\phi_j}{\partial x} + q\frac{\partial\phi_i}{\partial y}\frac{\partial\phi_j}{\partial y} - r\phi_i\phi_j\right] + \int_{S_2} dSg_1\phi_i\phi_j\right\}\gamma_i \\ &\quad + \iint_D dxdyf\phi_j - \int_{S_2} dSg_2(x,y)\phi_j = 0. \end{aligned} \tag{13}$$

But this expression is simply a matrix solve $\mathbf{Ax} = \mathbf{b}$ for the unknowns γ_i. In particular, we have

$$\mathbf{x} = \begin{pmatrix}\gamma_1\\ \gamma_2\\ \vdots\\ \gamma_n\end{pmatrix} \quad \mathbf{b} = \begin{pmatrix}\beta_1\\ \beta_2\\ \vdots\\ \beta_n\end{pmatrix} \quad \mathbf{A} = (\alpha_{ij}) \tag{14}$$

where

$$\beta_i = -\iint_D dxdyf\phi_i + \int_{S_2} dSg_2\phi_i - \sum_{k=n+1}^{m}\alpha_{ik}\gamma_k \tag{15a}$$

$$\alpha_{ij} = \iint_D dxdy\left[p\frac{\partial\phi_i}{\partial x}\frac{\partial\phi_j}{\partial x} + q\frac{\partial\phi_i}{\partial y}\frac{\partial\phi_j}{\partial y} - r\phi_i\phi_j\right] + \int_{S_2} dSg_1\phi_i\phi_j. \tag{15b}$$

Recall that each element ϕ_i is given by the simplex approximation

$$\phi_i = \sum_{i=1}^{3} N_j^{(i)}(x,y)\phi_j^{(i)} = \sum_{i=1}^{3} \left(a_j^{(i)} + b_j^{(i)}x + c_j^{(i)}y\right)\phi_j^{(i)}. \tag{16}$$

This concludes the construction of the approximate solution in the finite element basis. From an algorithmic point of view, the following procedure would be carried out to generate the solution.

- Discretize the computational domain into triangles. $T_1, T_2, \ldots, T_k$ are the interior triangles and $T_{k+1}, T_{k+2}, \ldots, T_m$ are triangles that have at least one edge which touches the boundary.
- For $l = k+1, k+2, \ldots, m$, determine the values of the vertices on the triangles which touch the boundary.
- Generate the shape functions
$$N_j^{(i)} = a_j^{(i)} + b_j^{(i)}x + c_j^{(i)}y \tag{17}$$
where $i = 1, 2, 3, \ldots, m$ and $j = 1, 2, 3, \ldots, m$.
- Compute the integrals for matrix elements α_{ij} and vector elements β_j in the interior and boundary.
- Construct the matrix $\mathbf{A}$ and vector $\mathbf{b}$.
- Solve $\mathbf{Ax} = \mathbf{b}$.
- Plot the solution $u(x,y) = \sum_{i=1}^{m} \gamma_i \phi_i(x,y)$.

For a wide variety of problems, the above procedures can simply be automated. That is exactly what commercial packages do. Thus once the coefficients and boundary conditions associated with (1), (2) and (3) are known, the solution procedure is straightforward.

Open source software

Although one could develop code for mesh construction and PDE solutions with finite elements (or finite volumes), there exist well developed computing platforms offering stable and robust implementations of both. Software like GOMA, FEniCS, or openFOAM offer incredible computing packages which allow for the simulation of complex, multi-scale physics problems without having to develop the infrastructure of geometry, adaptive meshing, and differentiation. These platforms allow for research level computing and are highly encouraged if the need for finite elements arises. As such, no homework exercises are proposed here.