Contents

For Pierre-Luigi
International Man of Mystery

Acknowledgments

The idea of this course began as a series of conversations with Dave Muraki. It has since grown into this scientific computing course whose ambition is to provide a truly versatile and useful course for students in the engineering, biological and physical sciences. I've also benefitted greatly from discussions with James Rossmanith, and with implementation ideas with Peter Blossey and Sorin Mitran. Leslie Butson, Sarah Hewitt and Jennifer O'Neil have been very helpful in editing the current set of notes so that it is more readable, useful, and error-free.

Prolegomenon

Scientific computing is ubiquitous in the physical, biological, and engineering sciences. Today, proficiency with computational methods, or lack thereof, can have a major impact on a researcher's ability to effectively analyze a given problem. Although a host of numerical analysis courses are traditionally offered in the mathematical sciences, the typical audience is the professional mathematician. Thus the emphasis is on establishing proven techniques and working through rigorous stability arguments. No doubt, this vision of numerical analysis is essential and provides the groundwork for this course on practical scientific computing. This more traditional approach to the teaching of numerical methods generally requires more than a year in coursework to achieve a level of proficiency necessary for solving practical problems.

The goal of this course is to embark on a new tradition: establishing computing proficiency as the first and foremost priority above rigorous analysis. Thus the major computational methods established over the past few decades are considered with emphasis on their use and implementation versus their rigorous analytic framework. A terse timeframe is also necessary in order to effectively augment the education of students from a wide variety of scientific departments. The three major techniques for solving partial differential equations are all considered: finite differences, finite elements, and spectral methods.

MATLAB has establised itself as the leader in scientific computing software. The built-in algorithms developed by MATLAB allow the computational focus to shift from technical details to overall implementation and solution techniques. Heavy and repeated use is made of MATLAB's linear algebra packages, Fast Fourier Transform routines, and finite element (partial differential equations) package. These routines are the workhorses for most solution techniques and are treated to a large extent as blackbox operations. Of course, cursory explanations are given of the underlying principles in any of the routines utilized, but it is largely left as reference material in order to focus on the application of the routine.

The end goal is for the student to develop a sense of confidence about implementing computational techniques. Specifically, at the end of the course, the student should be able to solve almost any 1D, 2D, or 3D problem of the elliptic, hyperbolic, or parabolic type. Or at the least, they should have a great deal of knowledge about how to solve the problem and should have enough information and references at their disposal to circumvent any implementation difficulties.

1 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.

1.1 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

$$\frac{d\mathbf{y}}{dt} = f(\mathbf{y}, t) \tag{1.1.1}$$

where $\mathbf{y}$ represents a vector and the initial conditions are given by

$$\mathbf{y}(0) = \mathbf{y}_0 \tag{1.1.2}$$

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 [1].

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:

$$\frac{d\mathbf{y}}{dt} = \lim_{\Delta t \to 0} \frac{\Delta \mathbf{y}}{\Delta t}\,. \tag{1.1.3}$$

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

$$\frac{d\mathbf{y}}{dt} = f(\mathbf{y}, t) \quad \Rightarrow \quad \frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{\Delta t} \approx f(\mathbf{y}_n, t_n)\,. \tag{1.1.4}$$

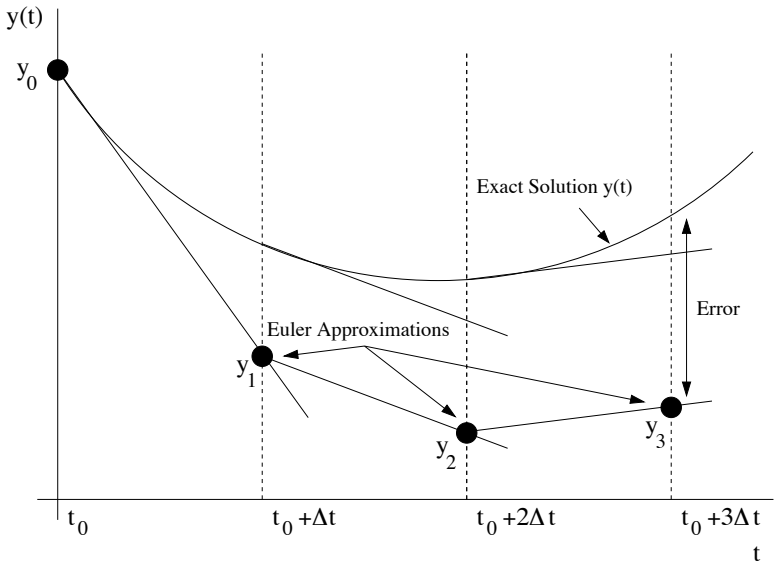

Figure 1: Graphical description of the iteration process used in the Euler method. Note that each subsequent approximation is generated from the slope of the previous point. This graphical illustration suggests that smaller steps Δt should be more accurate.

The approximation can easily be rearranged to give

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(\mathbf{y}_n, t_n)\,. \tag{1.1.5}$$

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

$$\mathbf{y}(t_{n+1}) = F(\mathbf{y}(t_n)) \tag{1.1.6}$$

where $F(\mathbf{y}(t_n)) = \mathbf{y}(t_n) + \Delta t \cdot f(\mathbf{y}(t_n), t_n)$ 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:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot \phi\,. \tag{1.1.7}$$

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 mid-point 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:

$$\mathbf{y}(t+\Delta t) = \mathbf{y}(t) + \Delta t\left[Af(t,\mathbf{y}(t)) + Bf(t+P\cdot\Delta t, \mathbf{y}(t)+Q\Delta t\cdot f(t,\mathbf{y}(t)))\right] \tag{1.1.8}$$

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

$$\begin{aligned} &f(t+P\cdot\Delta t, \mathbf{y}(t)+Q\Delta t\cdot f(t,\mathbf{y}(t))) = \\ &\quad f(t,\mathbf{y}(t)) + P\Delta t\cdot f_t(t,\mathbf{y}(t)) + Q\Delta t\cdot f_{\mathbf{y}}(t,\mathbf{y}(t))\cdot f(t,\mathbf{y}(t)) + O(\Delta t^2) \end{aligned} \tag{1.1.9}$$

where f_t and $f_{\mathbf{y}}$ denote differentiation with respect to t and $\mathbf{y}$ respectively, use has been made of (1.1.1), 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 (1.1.8) results in the following:

$$\begin{aligned} \mathbf{y}(t+\Delta t) &= \mathbf{y}(t) + \Delta t(A+B)f(t,\mathbf{y}(t)) \\ &+PB\Delta t^2\cdot f_t(t,\mathbf{y}(t)) + BQ\Delta t^2\cdot f_{\mathbf{y}}(t,\mathbf{y}(t))\cdot f(t,\mathbf{y}(t)) + O(\Delta t^3) \end{aligned} \tag{1.1.10}$$

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:

$$\begin{aligned} \mathbf{y}(t+\Delta t) = \mathbf{y}(t) + \Delta t\cdot f(t,\mathbf{y}(t)) + \frac{1}{2}\Delta t^2\cdot f_t(t,\mathbf{y}(t)) \\ +\frac{1}{2}\Delta t^2\cdot f_{\mathbf{y}}(t,\mathbf{y}(t))f(t,\mathbf{y}(t)) + O(\Delta t^3)\,. \end{aligned} \tag{1.1.11}$$

Comparing this Taylor expansion with (1.1.10) gives the following relations:

$$A + B = 1 \tag{1.1.12a}$$

$$PB = \frac{1}{2} \tag{1.1.12b}$$

$$BQ = \frac{1}{2} \tag{1.1.12c}$$

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:

$$\mathbf{y}(t+\Delta t) = \mathbf{y}(t) + \frac{\Delta t}{2}\left[f(t,\mathbf{y}(t)) + f(t+\Delta t, \mathbf{y}(t)+\Delta t\cdot f(t,\mathbf{y}(t)))\right] \tag{1.1.13a}$$

$$\mathbf{y}(t+\Delta t) = \mathbf{y}(t) + \Delta t\cdot f\left(t+\frac{\Delta t}{2}, \mathbf{y}(t)+\frac{\Delta t}{2}\cdot f(t,\mathbf{y}(t))\right)\,. \tag{1.1.13b}$$

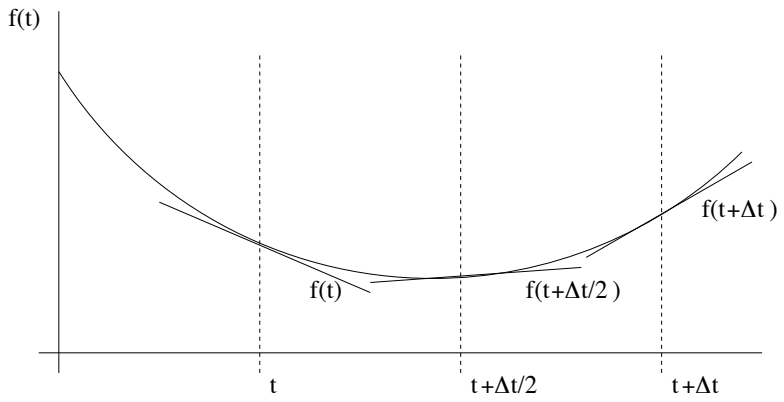

Figure 2: Graphical description of the initial, intermediate, and final slopes used in the 4th order Runge-Kutta iteration scheme over a time Δt.

Generally speaking, these methods for iterating forward in time given a single initial point are known as *Runge-Kutta methods*. By generalizing the assumption (1.1.8), 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.

4th-order Runge-Kutta

Perhaps the most popular general stepping scheme used in practice is known as the *4th order Runge-Kutta method*. The term "4th 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 "4th order". The scheme is as follows:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{6}\left[f_1 + 2f_2 + 2f_3 + f_4\right] \tag{1.1.14}$$

where

$$f_1 = f(t_n, \mathbf{y}_n) \tag{1.1.15a}$$
$$f_2 = f\left(t_n + \frac{\Delta t}{2}, \mathbf{y}_n + \frac{\Delta t}{2} f_1\right) \tag{1.1.15b}$$
$$f_3 = f\left(t_n + \frac{\Delta t}{2}, \mathbf{y}_n + \frac{\Delta t}{2} f_2\right) \tag{1.1.15c}$$
$$f_4 = f\left(t_n + \Delta t, \mathbf{y}_n + \Delta t \cdot f_3\right) . \tag{1.1.15d}$$

This scheme gives a local truncation error which is $O(\Delta t^5)$. The cumulative (global) error in this case case is fourth order so that for $t \sim O(1)$ then 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. For the 4th order scheme presented here, a graphical representation of this derivative sampling at intermediate time-steps is shown in Fig. 2.

Adams method: multi-stepping techniques

The development of the Runge-Kutta schemes rely on the definition of the derivative and Taylor expansions. Another approach to solving (1.1.1) is to start with the fundamental theorem of calculus [2]. Thus the differential equation can be integrated over a time-step Δt to give

$$\frac{d\mathbf{y}}{dt} = f(\mathbf{y}, t) \quad \Rightarrow \quad \mathbf{y}(t + \Delta t) - \mathbf{y}(t) = \int_t^{t+\Delta t} f(t, y)dt\,. \tag{1.1.16}$$

And once again using our iteration notation we find

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} f(t, y)dt\,. \tag{1.1.17}$$

This iteration relation is simply a restatement of (1.1.7) with $\Delta t \cdot \phi = \int_{t_n}^{t_{n+1}} f(t, y)dt$. However, at this point, no approximations have been made and (1.1.17) is exact. The numerical solution will be found by approximating $f(t, y) \approx p(t, y)$ where $p(t, y)$ is a polynomial. Thus the iteration scheme in this instance will be given by

$$\mathbf{y}_{n+1} \approx \mathbf{y}_n + \int_{t_n}^{t_{n+1}} p(t, y)dt\,. \tag{1.1.18}$$

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 $p(t, y)$. A first-order scheme can easily be constructed by allowing

$$p_1(t) = \text{constant} = f(t_n, \mathbf{y}_n)\,, \tag{1.1.19}$$

where the present point and no past points are used to determine the value of the polynomial. Inserting this first-order approximation into (1.1.18) results in the previously found Euler scheme

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(t_n, \mathbf{y}_n)\,. \tag{1.1.20}$$

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

$$p_2(t) = f_{n-1} + \frac{f_n - f_{n-1}}{\Delta t}(t - t_{n-1})\,. \tag{1.1.21}$$

When inserted into (1.1.18), this linear polynomial yields

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} \left(f_{n-1} + \frac{f_n - f_{n-1}}{\Delta t}(t - t_n) \right) dt\,. \tag{1.1.22}$$

Upon integration and evaluation at the upper and lower limits, we find the following 2nd order Adams-Bashforth scheme

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2}\left[3f(t_n, \mathbf{y}_n) - f(t_{n-1}, \mathbf{y}_{n-1})\right]. \tag{1.1.23}$$

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 *boot strap* to generate a second "initial condition" for the solution iteration process.

The Adams-Bashforth scheme uses current and past points to approximate the polymial $p(t, y)$ in (1.1.18). 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

$$p_1(t) = \text{constant} = f(t_{n+1}, \mathbf{y}_{n+1})\,, \tag{1.1.24}$$

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 (1.1.18) results in the *backward Euler scheme*

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(t_{n+1}, \mathbf{y}_{n+1})\,. \tag{1.1.25}$$

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

$$p_2(t) = f_n + \frac{f_{n+1} - f_n}{\Delta t}(t - t_n)\,. \tag{1.1.26}$$

Inserted into (1.1.18), this linear polynomial yields

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} \left(f_n + \frac{f_{n+1} - f_n}{\Delta t}(t - t_n) \right) dt\,. \tag{1.1.27}$$

Upon integration and evaluation at the upper and lower limits, we find the following 2nd order Adams-Moulton scheme

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2}\left[f(t_{n+1}, \mathbf{y}_{n+1}) + f(t_n, \mathbf{y}_n)\right]. \tag{1.1.28}$$

Once again this is a two-step algorithm. However, it is categorically different than 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 (1.1.28). 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. This is explored further in the notes.

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 (1.1.28) 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 (1.1.28) 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 (1.1.28) and explicitly find the corrected value of $\mathbf{y}_{n+1}$. The second-order predictor-corrector steps are then as follows:

$$\text{predictor (Adams-Bashforth): } \mathbf{y}^P_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2}[3f_n - f_{n-1}] \tag{1.1.29a}$$

$$\text{corrector (Adams-Moulton): } \mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2}[f(t_{n+1}, \mathbf{y}^P_{n+1}) + f(t_n, \mathbf{y}_n)]. \tag{1.1.29b}$$

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

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

$$\frac{d^3u}{dt^3} + u^2\frac{du}{dt} + \cos t \cdot u = g(t). \tag{1.1.30}$$

By defining

$$y_1 = u \tag{1.1.31a}$$

$$y_2 = \frac{du}{dt} \tag{1.1.31b}$$

$$y_3 = \frac{d^2u}{dt^2}\,, \tag{1.1.31c}$$

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

$$\frac{dy_1}{dt} = y_2 \tag{1.1.32a}$$

$$\frac{dy_2}{dt} = y_3 \tag{1.1.32b}$$

$$\frac{dy_3}{dt} = \frac{d^3u}{dt^3} = -u^2\frac{du}{dt} - \cos t \cdot u + g(t) = -y_1^2 y_2 - \cos t \cdot y_1 + g(t) \tag{1.1.32c}$$

which results in the original differential equation (1.1.1) considered previously

$$\frac{d\mathbf{y}}{dt} = \frac{d}{dt}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_2 \\ y_3 \\ -y_1^2 y_2 - \cos t \cdot y_1 + g(t) \end{pmatrix} = f(\mathbf{y}, t)\,. \tag{1.1.33}$$

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.

MATLAB commands

The time-stepping schemes considered here are all available in the MATLAB 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 [3, 4]

1.2 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 often are 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

$$\mathbf{y}(t+\Delta t) = \mathbf{y}(t) + \Delta t \cdot \frac{d\mathbf{y}(t)}{dt} + \frac{\Delta t^2}{2} \cdot \frac{d^2\mathbf{y}(c)}{dt^2} \tag{1.2.1}$$

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

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(t_n, \mathbf{y}_n) + O(\Delta t^2)\,. \tag{1.2.2}$$

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

$$E_k = \mathbf{y}(t_k) - \mathbf{y}_k \tag{1.2.3}$$

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

$$\epsilon_{k+1} = \mathbf{y}(t_{k+1}) - (\mathbf{y}(t_k) + \Delta t \cdot \phi) \tag{1.2.4}$$

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

$$\text{local:} \quad \epsilon_k = \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(c_k)}{dt^2} \sim O(\Delta t^2) \tag{1.2.5a}$$

$$\begin{aligned}\text{global:} \quad E_k &= \sum_{j=1}^{K} \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(c_j)}{dt^2} \approx \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(\mathbf{c})}{dt^2}\cdot K \\ &= \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(\mathbf{c})}{dt^2}\cdot\frac{b-a}{\Delta t} = \frac{b-a}{2}\Delta t\cdot\frac{d^2\mathbf{y}(\mathbf{c})}{dt^2} \sim O(\Delta t)\end{aligned} \tag{1.2.5b}$$

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.

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.

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 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

$$\frac{d\mathbf{y}}{dt} \approx \frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{\Delta t} + \epsilon(\mathbf{y}_n, \Delta t) \tag{1.2.6}$$

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

$$\mathbf{y}_{n+1} = \mathbf{Y}_{n+1} + \mathbf{e}_{n+1}\,. \tag{1.2.7}$$

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

$$\frac{d\mathbf{y}}{dt} = \frac{\mathbf{Y}_{n+1} - \mathbf{Y}_n}{\Delta t} + E_n(\mathbf{y}_n, \Delta t) \tag{1.2.8}$$

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

$$E_n = E_{\text{round}} + E_{\text{trunc}} = \frac{\mathbf{e}_{n+1} - \mathbf{e}_n}{\Delta t} - \frac{\Delta t^2}{2}\frac{d^2\mathbf{y}(c)}{dt^2}\,. \tag{1.2.9}$$

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

$$|\mathbf{e}_{n+1}| \le e_r \tag{1.2.10a}$$

$$|-\mathbf{e}_n| \le e_r \tag{1.2.10b}$$

$$M = \max_{c\in[t_n, t_{n+1}]} \left\{ \left| \frac{d^2\mathbf{y}(c)}{dt^2} \right| \right\}\,. \tag{1.2.10c}$$

This then gives the maximum error to be

$$|E_n| \leq \frac{e_r + e_r}{\Delta t} + \frac{\Delta t^2}{2} M = \frac{2e_r}{\Delta t} + \frac{\Delta t^2 M}{2} \,. \tag{1.2.11}$$

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

$$\frac{\partial |E_n|}{\partial(\Delta t)} = -\frac{2e_r}{\Delta t^2} + M\Delta t = 0 \,, \tag{1.2.12}$$

so that

$$\Delta t = \left(\frac{2e_r}{M} \right)^{1/3} \,. \tag{1.2.13}$$

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.

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

$$\frac{dy}{dt} = \lambda y \tag{1.2.14}$$

with

$$y(0) = y_0 \,. \tag{1.2.15}$$

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

$$y_{n+1} = y_n + \Delta t \cdot \lambda y_n = (1 + \lambda \Delta t) y_n \,. \tag{1.2.16}$$

After N steps, we find this iteration scheme yields

$$y_N = (1 + \lambda \Delta t)^N y_0 \,. \tag{1.2.17}$$

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

$$y_N = (1 + \lambda \Delta t)^N (y_0 + e) \,. \tag{1.2.18}$$

The error then associated with this scheme is given by

$$E = (1 + \lambda \Delta t)^N e \,. \tag{1.2.19}$$

At this point, the following observations can be made. For $\lambda > 0$, the solution $y_N \to \infty$ in Eq. (1.2.18) 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. (1.2.18) for $\lambda < 0$ is markedly different. For this case, $y_N \to 0$ in Eq. (1.2.18) as $N \to \infty$. The error, however, can dominate in this case. In particular, we have the two following cases for the error given by (1.2.19):

$$\text{I:} \quad |1 + \lambda\Delta t| < 1 \quad \text{then} \quad E \to 0 \tag{1.2.20a}$$

$$\text{II:} \quad |1 + \lambda\Delta t| > 1 \quad \text{then} \quad E \to \infty\,. \tag{1.2.20b}$$

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

$$\mathbf{y}_{n+1} = \mathbf{A}\mathbf{y}_n\,. \tag{1.2.21}$$

After N steps, the algorithm yields the solution

$$\mathbf{y}_N = \mathbf{A}^N\mathbf{y}_0\,, \tag{1.2.22}$$

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

$$\mathbf{A}^N = \mathbf{S}^{-1}\mathbf{\Lambda}^N\mathbf{S} \tag{1.2.23}$$

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

$$\mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \lambda_M \end{pmatrix} \;\to\; \mathbf{\Lambda}^N = \begin{pmatrix} \lambda_1^N & 0 & \cdots & 0 \\ 0 & \lambda_2^N & 0 & \cdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \lambda_M^N \end{pmatrix} \tag{1.2.24}$$

is a diagnonal 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 $\Re\{\lambda_i\} > 1$ for $i = 1, 2, ..., M$. This 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 (1.2.16)-(1.2.19). The backward Euler displays significant differences in stability. If we again consider (1.2.14) with (1.2.15), the backward Euler method gives the iteration scheme

$$y_{n+1} = y_n + \Delta t \cdot \lambda y_{n+1}\,, \tag{1.2.25}$$

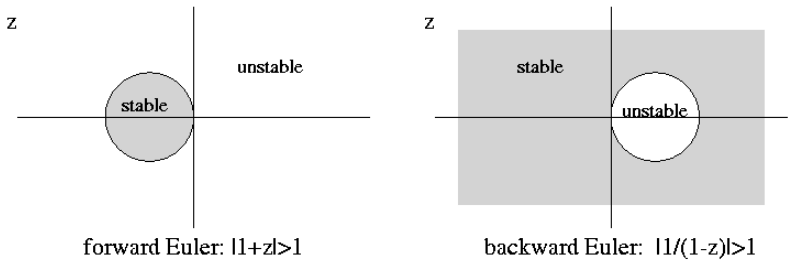

Figure 3: Regions for stable stepping for the forward Euler and backward Euler schemes.

which after N steps leads to

$$y_N = \left(\frac{1}{1-\lambda\Delta t}\right)^N y_0\,. \tag{1.2.26}$$

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

$$E = \left(\frac{1}{1-\lambda\Delta t}\right)^N e\,. \tag{1.2.27}$$

By letting $z = \lambda\Delta t$ be a complex number, we find the following criteria to yield unstable behavior based upon (1.2.19) and (1.2.27)

$$\text{forward Euler:} \quad |1+z| > 1 \tag{1.2.28a}$$

$$\text{backward Euler:} \quad \left|\frac{1}{1-z}\right| > 1\,. \tag{1.2.28b}$$

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.

1.3 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

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \tag{1.3.1}$$

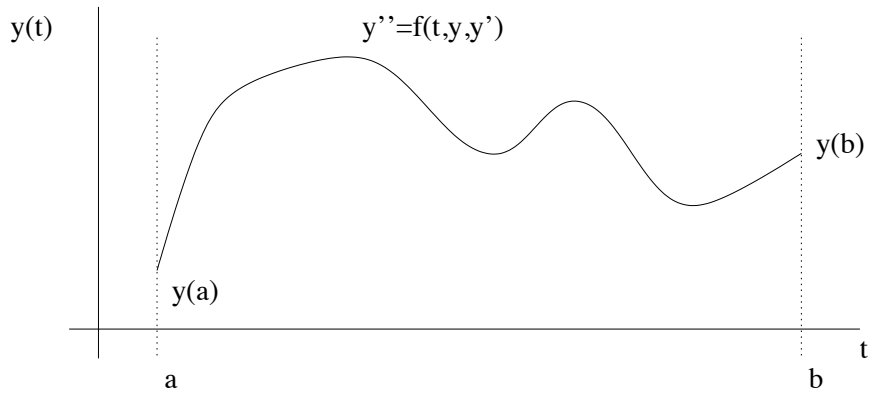

Figure 4: Graphical depiction of the structure of a typical solution to a boundary value problem with constraints at $t = a$ and $t = b$.

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

$$\alpha_1 y(a) + \beta_1 \frac{dy(a)}{dt} = \gamma_1 \tag{1.3.2a}$$

$$\alpha_2 y(b) + \beta_2 \frac{dy(b)}{dt} = \gamma_2 \,. \tag{1.3.2b}$$

Thus the solution is defined over a specific interval and must satisfy the relations (1.3.2) 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.

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

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \tag{1.3.3}$$

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

$$y(a) = \alpha \tag{1.3.4a}$$

$$y(b) = \beta \,. \tag{1.3.4b}$$

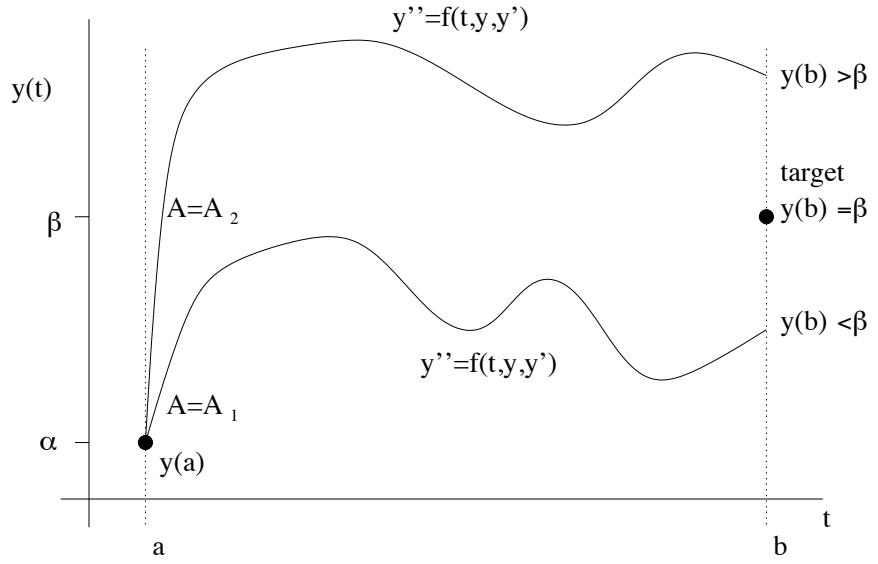

Figure 5: Solutions to the boundary value problem with $y(a) = \alpha$ and $y'(a) = A$. Here, two values of A are used to illustrate the solution behavior and its lack of matching the correct boundary value $y(b) = \beta$. However, the two solutions suggest that a bisection scheme could be used to find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

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

$$y(a) = \alpha \tag{1.3.5a}$$

$$\frac{dy(a)}{dt} = A\,, \tag{1.3.5b}$$

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 (1.3.4), whereas the value of $A = A_2$ is too large to satisfy (1.3.4).

Computational Algorithm

The above example demonstrates that adjusting the value of A in (1.3.5b) can lead to a solution which satisfies (1.3.4b). We can solve this using a self-

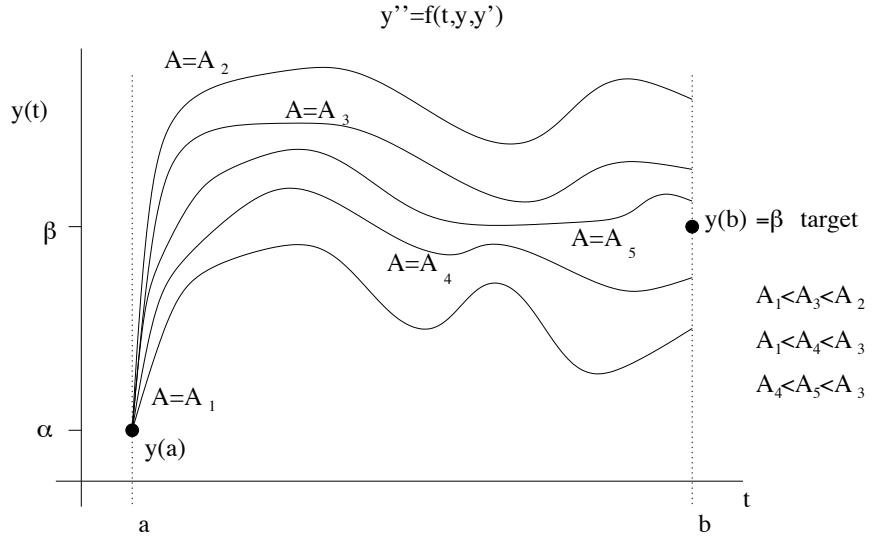

Figure 6: Graphical illustration of the shooting process which uses a bisection scheme to converge to the appropriate value of A for which $y(b) = \beta$.

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 amount of iterations provided the differential equation is well behaved.

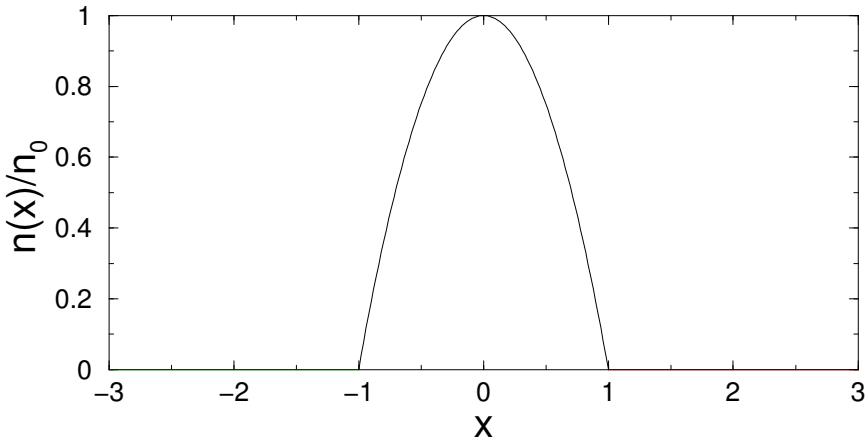

Figure 7: Plot of the spatial function $n(x)$.

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

$$\frac{d^2\psi_n}{dx^2} + [n(x) - \beta_n]\,\psi_n = 0 \tag{1.3.6}$$

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

$$n(x) = n_0 \begin{cases} 1 - |x|^2 & 0 \le |x| \le 1 \\ 0 & |x| > 1 \end{cases} \tag{1.3.7}$$

with n_0 being an arbitrary constant. Figure 7 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

$$\frac{d^2\psi_n}{dx^2} - \beta_n\psi_n = 0 \quad |x| > 1 \tag{1.3.8}$$

which has the general solution

$$\psi_n = c_1 \exp(\sqrt{\beta_n}x) + c_2 \exp(-\sqrt{\beta_n}x) \tag{1.3.9}$$

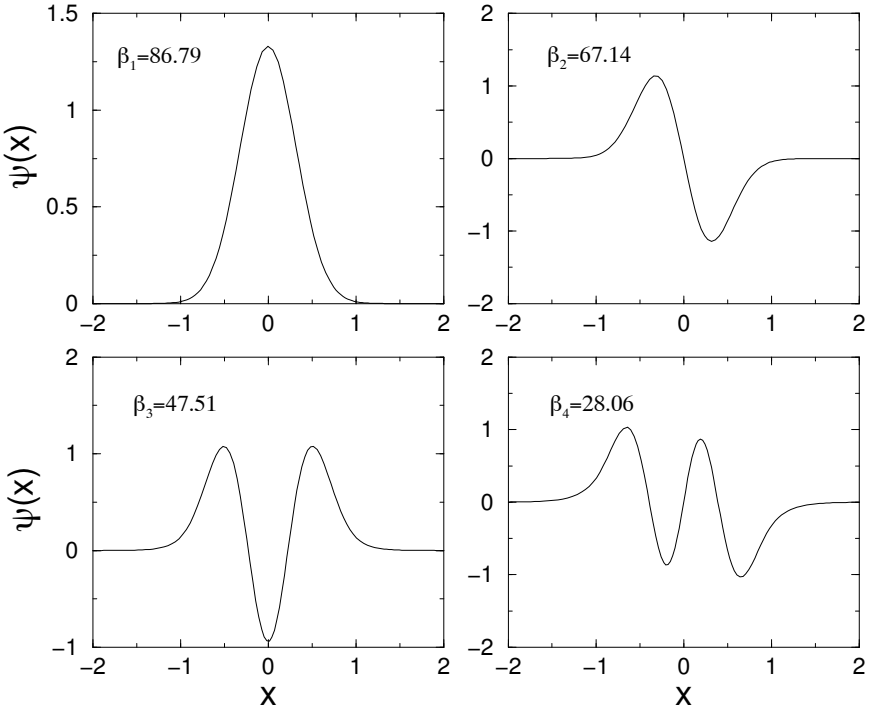

Figure 8: Plot of the first four eigenfunctions along with their eigenvalues β_n. For this example, $L = 2$ and $n_0 = 100$. These eigenmode structures are typical of those found in quantum mechanics and electromagnetic waveguides.

for $\beta_n \geq 0$. Note that we can only consider values of $\beta_n \geq 0$ 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 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

$$x \to \infty : \qquad \psi_n = c_2 \exp(-\sqrt{\beta_n}x) \tag{1.3.10a}$$

$$x \to -\infty : \qquad \psi_n = c_1 \exp(\sqrt{\beta_n}x)\,. \tag{1.3.10b}$$

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 (1.3.10) can equivalently

be thought of as solutions to the following first order equations:

$$x \to \infty : \qquad \frac{d\psi_n}{dx} + \sqrt{\beta_n}\psi_n = 0 \tag{1.3.11a}$$

$$x \to -\infty : \qquad \frac{d\psi_n}{dx} - \sqrt{\beta_n}\psi_n = 0\,. \tag{1.3.11b}$$

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

$$x = L : \qquad \frac{d\psi_n(L)}{dx} = -\sqrt{\beta_n}\psi_n(L) \tag{1.3.12a}$$

$$x = -L : \qquad \frac{d\psi_n(-L)}{dx} = \sqrt{\beta_n}\psi_n(-L)\,. \tag{1.3.12b}$$

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

$$x_1' = \psi_n' = x_2 \tag{1.3.13a}$$

$$x_2' = \psi_n'' = [\beta_n - n(x)]\,\psi_n = [\beta_n - n(x)]\,x_1\,. \tag{1.3.13b}$$

In matrix form, we can write the governing system as

$$\mathbf{x}' = \begin{pmatrix} 0 & 1 \\ \beta_n - n(x) & 0 \end{pmatrix} \mathbf{x} \tag{1.3.14}$$

where $\mathbf{x} = (x_1 \;\; x_2)^T = (\psi_n \;\; d\psi_n/dx)^T$. The boundary conditions (1.3.12) are

$$x = L : \qquad x_2 = -\sqrt{\beta_n}x_1 \tag{1.3.15a}$$

$$x = -L : \qquad x_2 = \sqrt{\beta_n}x_1\,. \tag{1.3.15b}$$

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 8 illustrates the first four eigenfunctions and their associate eigenvalues for $n_0 = 100$ and $L = 2$.

1.4 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 lecture. We consider

$$\mathbf{x}' = \begin{pmatrix} 0 & 1 \\ \beta_n - n(x) & 0 \end{pmatrix} \mathbf{x} \tag{1.4.1}$$

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

$$x = 1: \qquad \psi_n(1) = x_1(1) = 0 \tag{1.4.2a}$$
$$x = -1: \qquad \psi_n(-1) = x_1(-1) = 0\,. \tag{1.4.2b}$$

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.

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.
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: $\Delta t, \Delta t/2, \Delta t/4, \Delta t/8, \cdots$ 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.

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 hierarchy 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 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 9 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 MATLAB programming language. We begin with the initialization of the parameters.

Initialization

```
clear all;  % clear all previously defined variables
close all;  % clear all previously defined figures

tol=10^(-4);  % define a tolerance level to be achieved
           % by the shooting algorithm
col=['r','b','g','c','m','k'];  % eigenfunction colors

n0=100;    % define the parameter n0
A=1;       % define the initial slope at x=-1
x0=[0 A];  % initial conditions: x1(-1)=0, x1'(-1)=A
xp=[-1 1]; % define the span of the computational domain
```

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

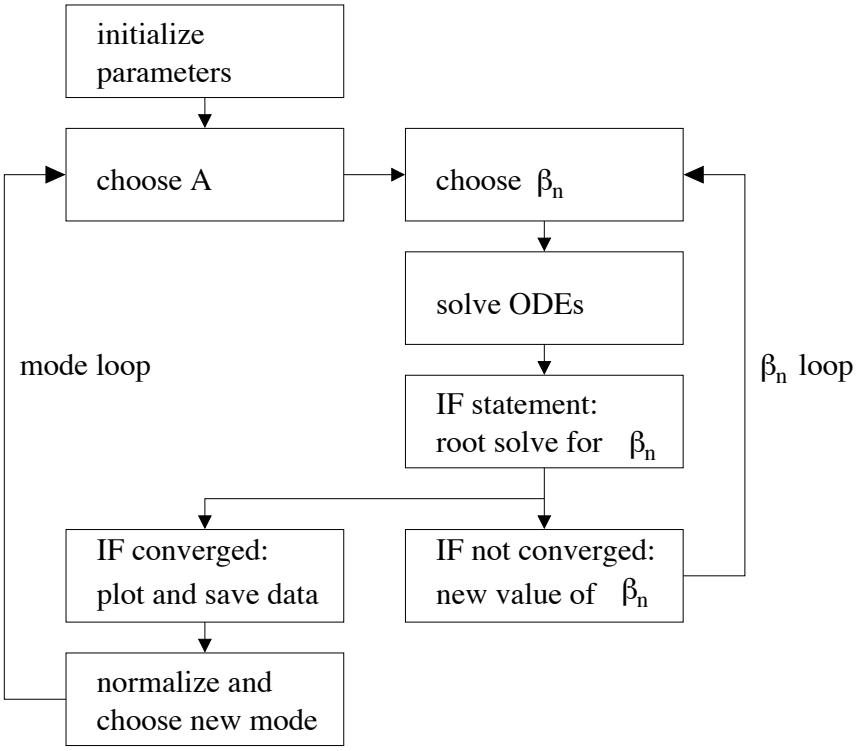

Figure 9: Basic algorithm structure for solving the boundary value problem. Two FOR loops are required to step through the values of β_n and A along with a single IF statement block to check for convergence of the solution

searches out a specified number of eigenmodes. Embedded in this FOR loop is a second FOR loop which attemps 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

```
beta_start=n0;  % beginning value of beta
for modes=1:5   %  begin mode loop
  beta=beta_start;   % initial value of eigenvalue beta
  dbeta=n0/100;      % default step size in beta
  for j=1:1000  % begin convergence loop for beta
    [t,y]=ode45('shoot2',xp,x0,[],n0,beta); % solve ODEs
    if abs(y(end,1)-0) < tol  % check for convergence
       beta                   % write out eigenvalue
```

```
      break                  % get out of convergence loop
    end
    if (-1)^(modes+1)*y(end,1)>0  %  this IF statement block
      beta=beta-dbeta;            %  checks to see if beta
    else                          %  needs to be higher or lower
      beta=beta+dbeta/2;          %  and uses bisection to
      dbeta=dbeta/2;              %  converge to the solution
    end                           %
  end   % end convergence loop
  beta_start=beta-0.1;  % after finding eigenvalue, pick
                        % new starting value for next mode
  norm=trapz(t,y(:,1).*y(:,1))  % calculate the normalization
  plot(t,y(:,1)/sqrt(norm),col(modes)); hold on  % plot modes
end   % end mode loop
```

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. For the differential equation considered here, the function *shoot2.m* would be the following:

shoot2.m

```
function rhs=shoot2(xspan,x,dummy,n0,beta)
rhs=[ x(2)
     (beta-n0)*x(1) ];
```

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 10 illustrates several characteristic features of this boundary value problem. In Fig. 10(a) and 10(b), the behavior of the solution near the first even and first odd solution is exhibited. From Fig. 10(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 10(c) illustrates the first four normalized eigenmodes along with their corresponding eigenvalues.

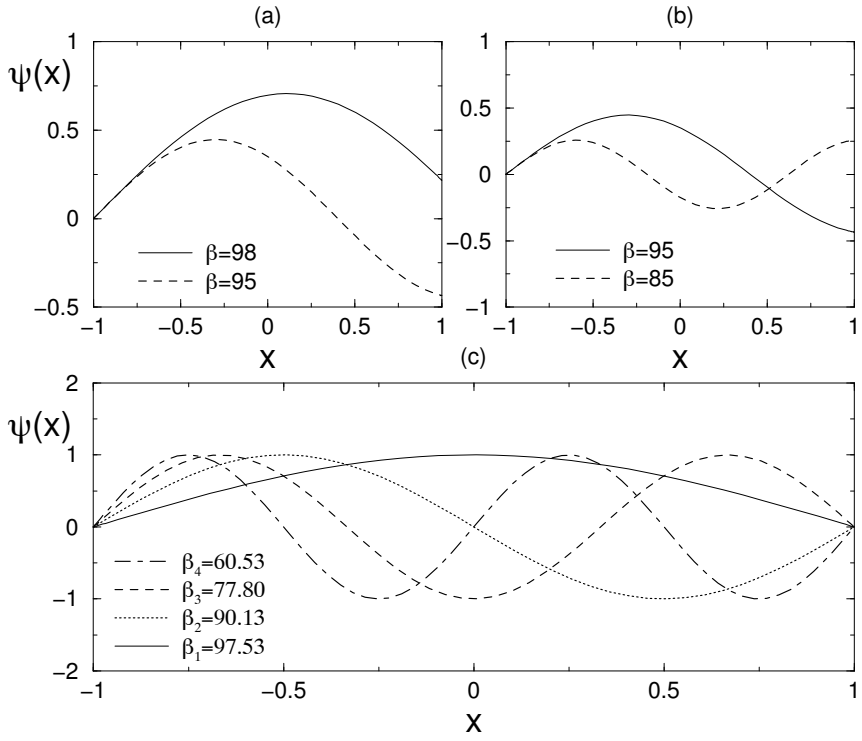

Figure 10: In (a) and (b) the behavior of the solution near the first even and first odd solution is depicted. Note that for the even modes increasing values of β bring the solution from $\psi_n(1) > 0$ to $\psi_n(1) < 0$. In contrast, odd modes go from $\psi_n(1) < 0$ to $\psi_n(1) > 0$ as β is increased. In (c) the first four normalized eigenmodes along with their corresponding eigenvalues are illustrated for $n_0 = 100$.

1.5 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

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \tag{1.5.1}$$

$O(\Delta t^2)$ center-difference schemes
$f'(t) = [f(t+\Delta t) - f(t-\Delta t)]/2\Delta t$
$f''(t) = [f(t+\Delta t) - 2f(t) + f(t-\Delta t)]/\Delta t^2$
$f'''(t) = [f(t+2\Delta t) - 2f(t+\Delta t) + 2f(t-\Delta t) - f(t-2\Delta t)]/2\Delta t^3$
$f''''(t) = [f(t+2\Delta t) - 4f(t+\Delta t) + 6f(t) - 4f(t-\Delta t) + f(t-2\Delta t)]/\Delta t^4$

Table 2: Second-order accurate center-difference formulas.

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

$$\alpha_1 y(a) + \beta_1 \frac{dy(a)}{dt} = \gamma_1 \tag{1.5.2a}$$

$$\alpha_2 y(b) + \beta_2 \frac{dy(b)}{dt} = \gamma_2 \,. \tag{1.5.2b}$$

Thus the solution is defined over a specific interval and must satisfy the relations (1.5.2) at the end points of the interval.

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

$$\frac{d^2y}{dt^2} = p(t)\frac{dy}{dt} + q(t)y + r(t) \tag{1.5.3}$$

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

$$y(a) = \alpha \tag{1.5.4a}$$

$$y(b) = \beta \,. \tag{1.5.4b}$$

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:

$$f(t+\Delta t) = f(t) + \Delta t \frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2f(t)}{dt^2} + \frac{\Delta t^3}{3!}\frac{d^3f(c_1)}{dt^3} \tag{1.5.5a}$$

$$f(t-\Delta t) = f(t) - \Delta t \frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2f(t)}{dt^2} - \frac{\Delta t^3}{3!}\frac{d^3f(c_2)}{dt^3} \tag{1.5.5b}$$

where $c_i \in [a, b]$. Subtracting these two expressions gives

$$f(t+\Delta t) - f(t-\Delta t) = 2\Delta t \frac{df(t)}{dt} + \frac{\Delta t^3}{3!}\left(\frac{d^3f(c_1)}{dt^3} + \frac{d^3f(c_2)}{dt^3}\right) . \tag{1.5.6}$$

$O(\Delta t^4)$ center-difference schemes
$f'(t) = [-f(t+2\Delta t) + 8f(t+\Delta t) - 8f(t-\Delta t) + f(t-2\Delta t)]/12\Delta t$
$f''(t) = [-f(t+2\Delta t) + 16f(t+\Delta t) - 30f(t) + 16f(t-\Delta t) - f(t-2\Delta t)]/12\Delta t^2$
$f'''(t) = [-f(t+3\Delta t) + 8f(t+2\Delta t) - 13f(t+\Delta t) + 13f(t-\Delta t) - 8f(t-2\Delta t) + f(t-3\Delta t)]/8\Delta t^3$
$f''''(t) = [-f(t+3\Delta t) + 12f(t+2\Delta t) - 39f(t+\Delta t) + 56f(t) - 39f(t-\Delta t) + 12f(t-2\Delta t) - f(t-3\Delta t)]/6\Delta t^4$

Table 3: Fourth-order accurate center-difference formulas.

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:

$$\frac{df(t)}{dt} = \frac{f(t+\Delta t) - f(t-\Delta t)}{2\Delta t} - \frac{\Delta t^2}{6}\frac{d^3 f(c)}{dt^3} \tag{1.5.7}$$

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 2-4 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 2 for the second and first derivatives. The boundary value problem then becomes

$$\frac{y(t+\Delta t)-2y(t)+y(t-\Delta t)}{\Delta t^2} = p(t)\frac{y(t+\Delta t)-y(t-\Delta t)}{2\Delta t} + q(t)y(t) + r(t) \tag{1.5.8}$$

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

$$\left[1 - \frac{\Delta t}{2}p(t)\right] y(t+\Delta t) - \left[2 + \Delta t^2 q(t)\right] y(t) + \left[1 + \frac{\Delta t}{2}\right] y(t-\Delta t) = \Delta t^2 r(t)\,. \tag{1.5.9}$$

$O(\Delta t^2)$ forward- and backward-difference schemes
$f'(t) = [-3f(t) + 4f(t+\Delta t) - f(t+2\Delta t)]/2\Delta t$
$f'(t) = [3f(t) - 4f(t-\Delta t) + f(t-2\Delta t)]/2\Delta t$
$f''(t) = [2f(t) - 5f(t+\Delta t) + 4f(t+2\Delta t) - f(t+3\Delta t)]/\Delta t^3$
$f''(t) = [2f(t) - 5f(t-\Delta t) + 4f(t-2\Delta t) - f(t-3\Delta t)]/\Delta t^3$

Table 4: Second-order accurate forward- and backward-difference formulas.

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

$$y(t_0) = y(a) = \alpha \tag{1.5.10a}$$
$$y(t_N) = y(b) = \beta\,. \tag{1.5.10b}$$

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

$$\mathbf{A} = \begin{bmatrix} 2+\Delta t^2 q(t_1) & -1+\frac{\Delta t}{2}p(t_1) & 0 & \cdots & & 0 \\ -1-\frac{\Delta t}{2}p(t_2) & 2+\Delta t^2 q(t_2) & -1+\frac{\Delta t}{2}p(t_2) & 0 & \cdots & \vdots \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & & & \ddots & \ddots & -1+\frac{\Delta t}{2}p(t_{N-2}) \\ 0 & \cdots & & 0 & -1-\frac{\Delta t}{2}p(t_{N-1}) & 2+\Delta t^2 q(t_{N-1}) \end{bmatrix} \tag{1.5.11}$$

and

$$\mathbf{x} = \begin{bmatrix} y(t_1) \\ y(t_2) \\ \vdots \\ y(t_{N-2}) \\ y(t_{N-1}) \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} -\Delta t^2 r(t_1) + (1 + \Delta t p(t_1)/2) y(t_0) \\ -\Delta t^2 r(t_2) \\ \vdots \\ -\Delta t^2 r(t_{N-2}) \\ -\Delta t^2 r(t_{N-1}) + (1 - \Delta t p(t_{N-1})/2) y(t_N) \end{bmatrix}. \tag{1.5.12}$$

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

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:

$$y'' = f(t, y, y') \rightarrow \frac{y(t+\Delta t) - 2y(t) + y(t-\Delta t)}{\Delta t^2} = f\left(t, y(t), \frac{y(t+\Delta t) - y(t-\Delta t)}{2\Delta t}\right). \tag{1.5.13}$$

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.

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

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 these 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.

Solving Nonlinear Systems: Newton-Raphson Iteration

The only built-in MATLAB command which solves nonlinear system of equations is FSOLVE. However, this command is now packaged within the optimization toolbox. Most users of MATLAB do not have access to this toolbox and alternatives must be sought. We therefore develop the basic ideas of the Newton-Raphson Iteration method, commonly known as a Newton's method. We begin by considering a single nonlinear equation

$$f(x_r) = 0 \tag{1.5.15}$$

where x_r is the root to the equation and the value being sought. We would like to develop a scheme which systematically determines the value of x_r. The Newton-Raphson method is an iterative scheme which relies on an initial guess, x_0, for the value of the root. From this guess, subsequent guesses are determined until

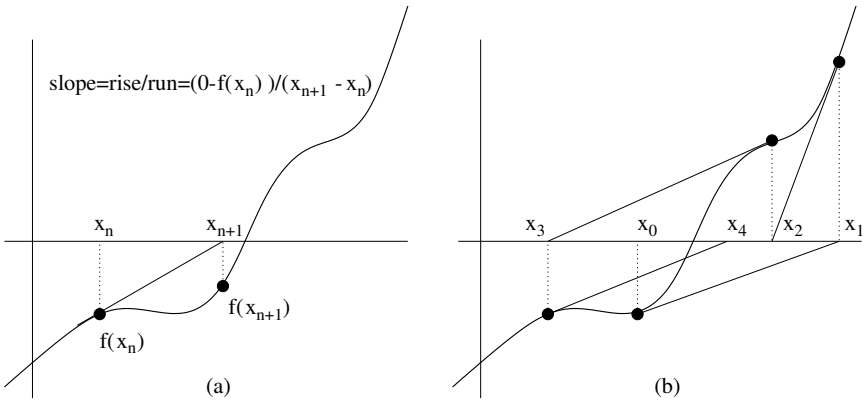

Figure 11: Construction and implementation of the Newton-Raphson iteration formula. In (a), the slope is the determining factor in deriving the Newton-Raphson formula. In (b), a graphical representation of the iteration scheme is given.

the scheme either converges to the root x_r or the scheme diverges and another initial guess is used. The sequence of guesses $(x_0, x_1, x_2, ...)$ is generated from the slope of the function $f(x)$. The graphical procedure is illustrated in Fig. 11. In essence, everything relies on the slope formula as illustrated in Fig. 11(a):

$$\text{slope} = \frac{df(x_n)}{dx} = \frac{\text{rise}}{\text{run}} = \frac{0 - f(x_n)}{x_{n+1} - x_n}\,. \tag{1.5.16}$$

Rearranging this gives the Newton-Raphson iterative relation

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\,. \tag{1.5.17}$$

A graphical example of how the iteration procedure works is given in Fig. 11(b) where a sequence of iterations is demonstrated. Note that the scheme fails if $f'(x_n) = 0$ since then the slope line never intersects $y = 0$. Further, for certain guesses the iterations may diverge. Provided the initial guess is sufficiently close, the scheme usually will converge. Conditions for convergence can be found in Burden and Faires [5].

The Newton method can be generalized for system of nonlinear equations. The details will not be discussed here, but the Newton iteration scheme is similar to that developed for the single function case. Given a system:

$$\mathbf{F}(\mathbf{x_n}) = \begin{bmatrix} f_1(x_1, x_2, x_3, ..., x_N) \\ f_2(x_1, x_2, x_3, ..., x_N) \\ \vdots \\ f_N(x_1, x_2, x_3, ..., x_N) \end{bmatrix} = \mathbf{0}\,, \tag{1.5.18}$$

the iteration scheme is

$$\mathbf{x_{n+1}} = \mathbf{x_n} + \mathbf{\Delta x_n} \tag{1.5.19}$$

where

$$\mathbf{J}(\mathbf{x_n})\mathbf{\Delta x_n} = -\mathbf{F}(\mathbf{x_n}) \tag{1.5.20}$$

and $\mathbf{J}(\mathbf{x_n})$ is the Jacobian matrix

$$\mathbf{J}(\mathbf{x_n}) = \begin{bmatrix} f_{1_{x_1}} & f_{1_{x_2}} & \cdots & f_{1_{x_N}} \\ f_{2_{x_1}} & f_{2_{x_2}} & \cdots & f_{2_{x_N}} \\ \vdots & \vdots & & \vdots \\ f_{N_{x_1}} & f_{N_{x_2}} & \cdots & f_{N_{x_N}} \end{bmatrix} \tag{1.5.21}$$

This algorithm relies on initially guessing values for $x_1, x_2, ..., x_N$. As before, there is no guarantee that the algorithm will converge. Thus a good initial guess is critical to its success. Further, the determinant of the Jacobian cannot equal zero, $\det \mathbf{J}(\mathbf{x_n}) \neq \mathbf{0}$, in order for the algorithm to work.

2 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.

2.1 Finite difference discretization

To discuss the solution of a given problem with the finite difference method, we consider a specific example from atmospheric sciences. 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 $\psi(x, y, t)$:

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega \tag{2.1.1a}$$

$$\nabla^2 \psi = \omega \tag{2.1.1b}$$

where

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} \tag{2.1.2}$$

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

$$\omega(x, y, t = 0) = \omega_0(x, y)\,. \tag{2.1.3}$$

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

$$\omega(-L, y, t) = \omega(L, y, t) \tag{2.1.4a}$$
$$\omega(x, -L, t) = \omega(x, L, t) \tag{2.1.4b}$$
$$\psi(-L, y, t) = \psi(L, y, t) \tag{2.1.4c}$$
$$\psi(x, -L, t) = \psi(x, L, t) \tag{2.1.4d}$$

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:

$$\psi(x, y, t) \quad \text{streamfunction} \tag{2.1.5a}$$
$$\omega(x, y, t) \quad \text{vorticity}\,. \tag{2.1.5b}$$

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

$$\nabla^2\psi = \frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} = \omega \tag{2.1.6}$$

Using the central difference formulas of Sec. 1.5 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:

$$\begin{aligned}\frac{\psi(x+\Delta x, y, t) - 2\psi(x, y, t) + \psi(x-\Delta x, y, t)}{\Delta x^2} \\ + \frac{\psi(x, y+\Delta y, t) - 2\psi(x, y, t) + \psi(x, y-\Delta y, t)}{\Delta y^2} = \omega(x, y, t)\end{aligned} \tag{2.1.7}$$

Thus the solution at each point depends upon itself and four neighboring points. This creates a five point stencil for solving this equation. Figure 12 illustrates the stencil which arises from discretization. For convenience we denote

$$\psi_{mn} = \psi(x_m, y_n)\,. \tag{2.1.8}$$

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

$$-4\psi_{mn} + \psi_{(m-1)n} + \psi_{(m+1)n} + \psi_{m(n-1)} + \psi_{m(n+1)} = \delta^2\omega_{mn} \tag{2.1.9}$$

with periodic boundary conditions imposing the following constraints

$$\psi_{1n} = \psi_{(N+1)n} \tag{2.1.10a}$$

$$\psi_{m1} = \psi_{m(N+1)} \tag{2.1.10b}$$

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

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

which results in the sparse matrix (banded matrix) system

$$\mathbf{A}\psi = \delta^2\omega \tag{2.1.12}$$

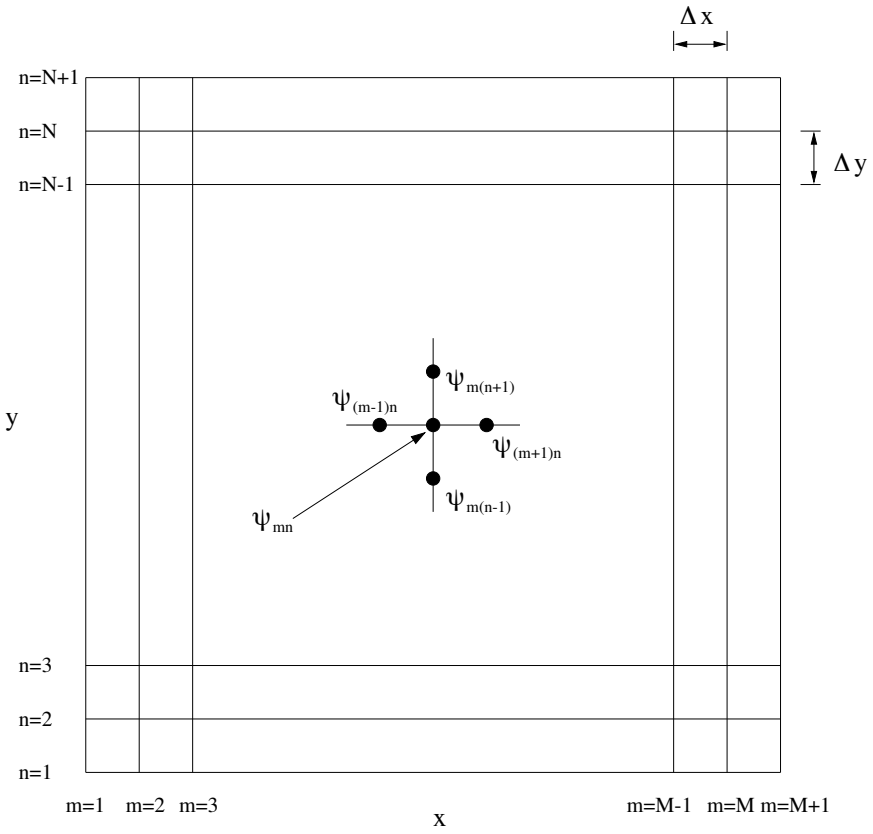

Figure 12: Discretization stencil for solving for the streamfunction with second-order accurate central difference schemes. Note that $\psi_{mn} = \psi(x_m, y_n)$.

where

$$\mathbf{A} = \begin{bmatrix}
-4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -4
\end{bmatrix} \tag{2.1.13}$$

and

$$\psi = (\psi_{11}\ \psi_{12}\ \psi_{13}\ \psi_{14}\ \psi_{21}\ \psi_{22}\ \psi_{23}\ \psi_{24}\ \psi_{31}\ \psi_{32}\ \psi_{33}\ \psi_{34}\ \psi_{41}\ \psi_{42}\ \psi_{43}\ \psi_{44})^{\mathrm{T}} \tag{2.1.14a}$$

$$\omega = \delta^2(\omega_{11}\,\omega_{12}\,\omega_{13}\,\omega_{14}\,\omega_{21}\,\omega_{22}\,\omega_{23}\,\omega_{24}\,\omega_{31}\,\omega_{32}\,\omega_{33}\,\omega_{34}\,\omega_{41}\,\omega_{42}\,\omega_{43}\,\omega_{44})^{\mathrm{T}} \quad (2.1.14b)$$

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 $\mathbf{A}$ 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:

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega\,. \quad (2.1.15)$$

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

$$\frac{\partial \omega}{\partial t} = \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} - \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} + \nu\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right). \quad (2.1.16)$$

Second order central-differencing discretization then yields

$$\begin{aligned}\frac{\partial \omega}{\partial t} \;=\; & \left(\frac{\psi(x,y+\Delta y,t)-\psi(x,y-\Delta y,t)}{2\Delta y}\right)\left(\frac{\omega(x+\Delta x,y,t)-\omega(x-\Delta x,y,t)}{2\Delta x}\right) \\ & -\left(\frac{\psi(x+\Delta x,y,t)-\psi(x-\Delta x,y,t)}{2\Delta x}\right)\left(\frac{\omega(x,y+\Delta y,t)-\omega(x,y-\Delta y,t)}{2\Delta y}\right) \\ & +\nu\left\{\frac{\omega(x+\Delta x,y,t)-2\omega(x,y,t)+\omega(x-\Delta x,y,t)}{\Delta x^2}\right. \\ & \left.+\frac{\omega(x,y+\Delta y,t)-2\omega(x,y,t)+\omega(x,y-\Delta y,t)}{\Delta y^2}\right\}. \end{aligned} \quad (2.1.17)$$

This is simply a large system of differential equations which can be stepped forward in time with any convenient time-stepping algorithm such as 4th 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 streamfunction to once again update the vorticity. This loop continues until the solution at the desired future time is achieved. Figure 13 illustrates how the five-point, two-dimensional stencil advances the solution.

The behavior of the vorticity is illustrated in Fig. 14 where the solution is advanced for eight time units. The initial condition used in this simulation is

$$\omega_0 = \omega(x,y,t=0) = \exp\left(-2x^2 - \frac{y^2}{20}\right). \quad (2.1.18)$$

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

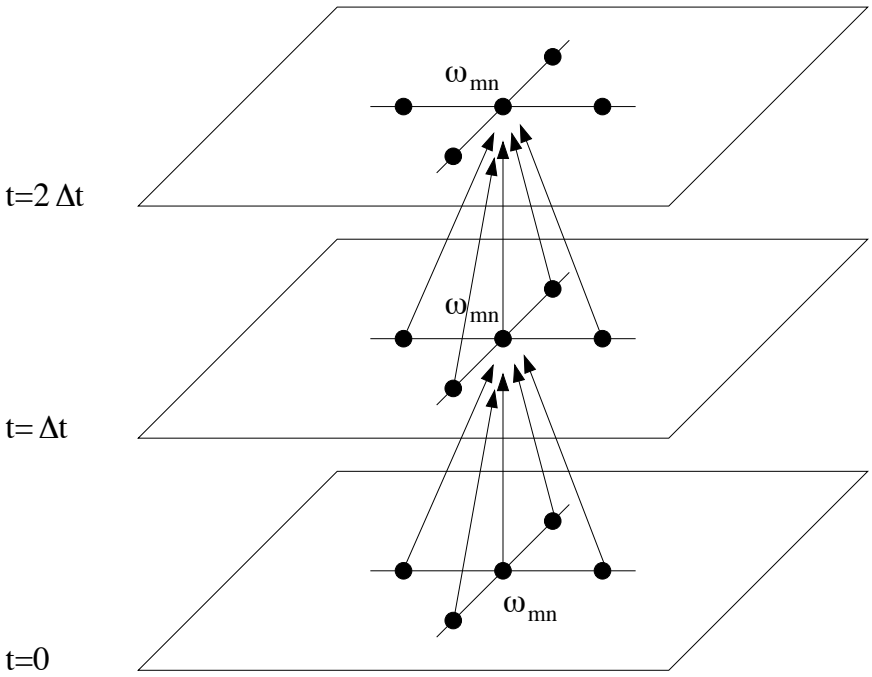

Figure 13: Discretization stencil resulting from center-differencing of the advection-diffusion equations. Note that for explicit stepping schemes the future solution only depends upon the present. Thus we are not required to solve a large linear system of equations.

2.2 Direct solution methods for Ax=b

A central concern in almost any computational strategy is a fast and efficient computational method for achieving a solution of a large system of equations $\mathbf{Ax} = \mathbf{b}$. In trying to render a computation tractable, it is crucial to minimize the operations it takes in solving such a system. There are a variety of direct methods for solving $\mathbf{Ax} = \mathbf{b}$: Gaussian elimination, LU decomposition, and inverting the matrix $\mathbf{A}$. In addition to these direct methods, iterative schemes can also provide efficient solution techniques. Some basic iterative schemes will be discussed in what follows.

The standard beginning to discussions of solution techniques for $\mathbf{Ax} = \mathbf{b}$ involves Gaussian elimination. We will consider a very simple example of a 3×3 system in order to understand the operation count and numerical procedure

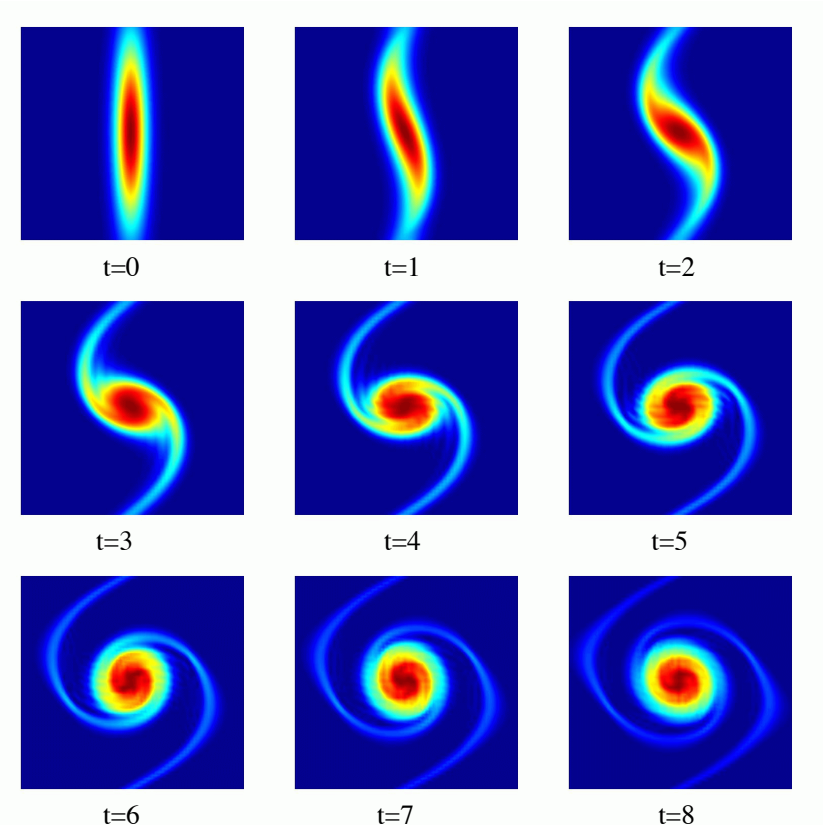

Figure 14: Time evolution of the vorticity $\omega(x, y, t)$ over eight time units with $\nu = 0.001$ and a spatial domain $x \in [-10, 10]$ and $y \in [-10, 10]$. The initial condition was a stretched Gaussian of the form $\omega(x, y, 0) = \exp(-2x^2 - y^2/20)$.

involved in this technique. Thus consider $\mathbf{Ax} = \mathbf{b}$ with

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix}. \tag{2.2.1}$$

The Gaussian elimination procedure begins with the construction of the augmented matrix

$$[\mathbf{A}|\mathbf{b}] = \left[\begin{array}{ccc|c} \underline{\mathbf{1}} & 1 & 1 & 1 \\ 1 & 2 & 4 & -1 \\ 1 & 3 & 9 & 1 \end{array}\right]$$

$$
\begin{aligned}
&= \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & -2 \\ 0 & 2 & 8 & 0 \end{array}\right] \\
&= \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & \underline{\mathbf{1}} & 3 & -2 \\ 0 & 1 & 4 & 0 \end{array}\right] \\
&= \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & -2 \\ 0 & 0 & 1 & 2 \end{array}\right] \qquad (2.2.2)
\end{aligned}
$$

where we have underlined and bolded the pivot of the augmented matrix. Back substituting then gives the solution

$$
\begin{aligned}
x_3 = 2 \quad &\rightarrow \; x_3 = 2 && (2.2.3a) \\
x_2 + 3x_3 = -2 \quad &\rightarrow \; x_2 = -8 && (2.2.3b) \\
x_1 + x_2 + x_3 = 1 \quad &\rightarrow \; x_1 = 7\,. && (2.2.3c)
\end{aligned}
$$

This procedure can be carried out for any matrix $\mathbf{A}$ which is nonsingular, i.e. $\det \mathbf{A} \neq 0$. In this algorithm, we simply need to avoid these singular matrices and occasionally shift the rows around to avoid a zero pivot. Provided we do this, it will always yield an answer.

The fact that this algorithm works is secondary to the concern of the time required in generating a solution in scientific computing. The operation count for the Gaussian elimination can easily be estimated from the algorithmic procedure for an $N \times N$ matrix:

1. Movement down the N pivots
2. For each pivot, perform N additions/subtractions across a given row.
3. For each pivot, perform the addition/subtraction down the N rows.

In total, this results in a scheme whose operation count is $O(N^3)$. The back substitution algorithm can similarly be calculated to give an $O(N^2)$ scheme.

LU Decomposition

Each Gaussian elimination operation costs $O(N^3)$ operations. This can be computationally prohibitive for large matrices when repeated solutions of $\mathbf{Ax} = \mathbf{b}$ must be found. When working with the same matrix $\mathbf{A}$ however, the operation count can easily be brought down to $O(N^2)$ using LU factorization which splits the matrix $\mathbf{A}$ into a lower triangular matrix L, and an upper triangular matrix

U. For a 3×3 matrix, the LU factorization scheme splits $\mathbf{A}$ as follows:

$$\mathbf{A}=\mathbf{LU} \rightarrow \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & m_{32} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}. \tag{2.2.4}$$

Thus the $\mathbf{L}$ matrix is lower triangular and the $\mathbf{U}$ matrix is upper triangular. This then gives

$$\mathbf{Ax} = \mathbf{b} \;\rightarrow\; \mathbf{LUx} = \mathbf{b} \tag{2.2.5}$$

where by letting $\mathbf{y} = \mathbf{Ux}$ we find the coupled system

$$\mathbf{Ly} = \mathbf{b} \;\text{ and }\; \mathbf{Ux} = \mathbf{y} \tag{2.2.6}$$

where the system $\mathbf{Ly} = \mathbf{b}$

$$y_1 = b_1 \tag{2.2.7a}$$
$$m_{21}y_1 + y_2 = b_2 \tag{2.2.7b}$$
$$m_{31}y_1 + m_{32}y_2 + y_3 = b_3 \tag{2.2.7c}$$

can be solved by $O(N^2)$ forward substitution and the system $\mathbf{Ux} = \mathbf{y}$

$$u_{11}x_1 + u_{12}x_2 + x_3 = y_1 \tag{2.2.8a}$$
$$u_{22}x_2 + u_{23}x_3 = y_2 \tag{2.2.8b}$$
$$u_{33}x_3 = y_3 \tag{2.2.8c}$$

can be solved by $O(N^2)$ back substitution. Thus once the factorization is accomplished, the LU results in an $O(N^2)$ scheme for arriving at the solution. The factorization itself is $O(N^3)$, but you only have to do this once. Note, you should **always** use LU decomposition if possible. Otherwise, you are doing far more work than necessary in achieving a solution.

As an example of the application of the LU factorization algorithm, we consider the 3×3 matrix

$$\mathbf{A}=\begin{pmatrix} 4 & 3 & -1 \\ -2 & -4 & 5 \\ 1 & 2 & 6 \end{pmatrix}. \tag{2.2.9}$$

The factorization starts from the matrix multiplication of the matrix $\mathbf{A}$ and the identity matrix $\mathbf{I}$

$$\mathbf{A} = \mathbf{IA} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \underline{\mathbf{4}} & 3 & -1 \\ -2 & -4 & 5 \\ 1 & 2 & 6 \end{pmatrix}. \tag{2.2.10}$$

The factorization begins with the pivot element. To use Gaussian elimination, we would multiply the pivot by $-1/2$ to eliminate the first column element in

the second row. Similarly, we would multiply the pivot by 1/4 to eliminate the first column element in the third row. These multiplicative factors are now part of the first matrix above:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/4 & 0 & 1 \end{pmatrix} \begin{pmatrix} 4 & 3 & -1 \\ 0 & \underline{\mathbf{-2.5}} & 4.5 \\ 0 & 1.25 & 6.25 \end{pmatrix}. \tag{2.2.11}$$

To eliminate on the third row, we use the next pivot. This requires that we multiply by $-1/2$ in order to eliminate the second column, third row. Thus we find

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/4 & -1/2 & 1 \end{pmatrix} \begin{pmatrix} 4 & 3 & -1 \\ 0 & -2.5 & 4.5 \\ 0 & 0 & 8.5 \end{pmatrix}. \tag{2.2.12}$$

Thus we find that

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/4 & -1/2 & 1 \end{pmatrix} \quad \text{and} \quad \mathbf{U} = \begin{pmatrix} 4 & 3 & -1 \\ 0 & -2.5 & 4.5 \\ 0 & 0 & 8.5 \end{pmatrix}. \tag{2.2.13}$$

It is easy to verify by direct substitution that indeed $\mathbf{A} = \mathbf{LU}$. Just like Gaussian elimination, the cost of factorization is $O(N^3)$. However, once $\mathbf{L}$ and $\mathbf{U}$ are known, finding the solution is an $O(N^2)$ operation.

The Permutation Matrix

As will often happen with Gaussian elimination, following the above algorithm will at times result in a zero pivot. This is easily handled in Gaussian elimination by shifting rows in order to find a non-zero pivot. However, in LU decomposition, we must keep track of this row shift since it will effect the right hand side vector $\mathbf{b}$. We can keep track of row shifts with a row permutation matrix $\mathbf{P}$. Thus if we need to permute two rows, we find

$$\mathbf{Ax} = \mathbf{b} \rightarrow \mathbf{PAx} = \mathbf{Pb} \rightarrow \mathbf{PLUx} = \mathbf{Pb} \tag{2.2.14}$$

thus $\mathbf{PA} = \mathbf{LU}$. To shift rows one and two, for instance, we would have

$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 & \cdots \\ 1 & 0 & 0 & \cdots \\ 0 & 0 & 1 & \cdots \\ \vdots & & & \end{pmatrix}. \tag{2.2.15}$$

If permutation is necessary, MATLAB can supply the permutation matrix associated with the LU decomposition.

MATLAB: A \ b

Given the alternatives for solving the linear system $\mathbf{Ax} = \mathbf{b}$, it is important to know how the MATLAB command structure for $\mathbf{A} \setminus \mathbf{b}$ works. The following is an outline of the algorithm performed.

1. It first checks to see if $\mathbf{A}$ is triangular, or some permutation thereof. If it is, then all that is needed is a simple $O(N^2)$ substitution routine.
2. It then checks if $\mathbf{A}$ is symmetric, i.e. Hermitian or Self-Adjoint. If so, a Cholesky factorization is attempted. If $\mathbf{A}$ is positive definite, the Cholesky algorithm is always succesful and takes half the run time of LU factorization.
3. It then checks if $\mathbf{A}$ is Hessenberg. If so, it can be written as an upper triangular matrix and solved by a substitution routine.
4. If all the above methods fail, then LU factorization is used and the forward and backward substitution routines generate a solution.
5. If $\mathbf{A}$ is not square, a QR (Householder) routine is used to solve the system.
6. If $\mathbf{A}$ is not square and sparse, a least squares solution using QR factorization is performed.

Note that solving by $\mathbf{b} = \mathbf{A}^{-1}\mathbf{x}$ is the slowest of all methods, taking 2.5 times longer or more than $\mathbf{A} \setminus \mathbf{b}$. It is not recommended. However, just like LU factorization, once the inverse is known it need not be calculated again. Care must also be taken when the $\det \mathbf{A} \approx 0$, i.e. the matrix is ill-conditioned.

MATLAB commands

The commands for executing the linear system solve are as follows

- **$\mathbf{A} \setminus \mathbf{b}$:** Solve the system in the order above.
- **$[L, U] = lu(A)$:** Generate the L and U matrices.
- **$[L, U, P] = lu(A)$:** Generate the L and U factorization matrices along with the permutation matrix P.

2.3 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. These iterative techniques can often go under the name of Krylov space methods [6]. The idea is to start with an initial guess for the solution and develop an iterative procedure that will converge to the solution. The simplest example

of this method is known as a Jacobi iteration scheme. The implementation of this scheme is best illustrated with an example. We consider the linear system

$$4x - y + z = 7 \tag{2.3.1a}$$
$$4x - 8y + z = -21 \tag{2.3.1b}$$
$$-2x + y + 5z = 15\,. \tag{2.3.1c}$$

We can rewrite each equation as follows

$$x = \frac{7 + y - z}{4} \tag{2.3.2a}$$
$$y = \frac{21 + 4x + z}{8} \tag{2.3.2b}$$
$$z = \frac{15 + 2x - y}{5}\,. \tag{2.3.2c}$$

To solve the system iteratively, we can define the following Jacobi iteration scheme based on the above

$$x_{k+1} = \frac{7 + y_k - z_k}{4} \tag{2.3.3a}$$
$$y_{k+1} = \frac{21 + 4x_k + z_k}{8} \tag{2.3.3b}$$
$$z_{k+1} = \frac{15 + 2x_k - y_k}{5}\,. \tag{2.3.3c}$$

An algorithm is then easily implemented computationally. In particular, we would follow the structure:

1. Guess initial values: (x_0, y_0, z_0).
2. Iterate the Jacobi scheme: $\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k$.
3. Check for convergence: $\| \mathbf{x}_{k+1} - \mathbf{x}_k \|$<tolerance.

Note that the choice of an initial guess is often critical in determining the convergence to the solution. Thus the more that is known about what the solution is supposed to look like, the higher the chance of successful implementation of the iterative scheme. Table 5 shows the convergence of this scheme for this simple example.

Given the success of this example, it is easy to conjecture that such a scheme will always be effective. However, we can reconsider the original system by interchanging the first and last set of equations. This gives the system

$$-2x + y + 5z = 15 \tag{2.3.4a}$$
$$4x - 8y + z = -21 \tag{2.3.4b}$$
$$4x - y + z = 7\,. \tag{2.3.4c}$$

k	x_k	y_k	z_k
0	1.0	2.0	2.0
1	1.75	3.375	3.0
2	1.84375	3.875	3.025
⋮	⋮	⋮	⋮
15	1.99999993	3.99999985	2.9999993
⋮	⋮	⋮	⋮
19	2.0	4.0	3.0

Table 5: Convergence of Jacobi iteration scheme to the solution value of $(x, y, z) = (2, 4, 3)$ from the initial guess $(x_0, y_0, z_0) = (1, 2, 2)$.

k	x_k	y_k	z_k
0	1.0	2.0	2.0
1	-1.5	3.375	5.0
2	6.6875	2.5	16.375
3	34.6875	8.015625	-17.25
⋮	⋮	⋮	⋮
	$\pm\infty$	$\pm\infty$	$\pm\infty$

Table 6: Divergence of Jacobi iteration scheme from the initial guess $(x_0, y_0, z_0) = (1, 2, 2)$.

To solve the system iteratively, we can define the following Jacobi iteration scheme based on this rearranged set of equations

$$x_{k+1} = \frac{y_k + 5z_k - 15}{2} \tag{2.3.5a}$$

$$y_{k+1} = \frac{21 + 4x_k + z_k}{8} \tag{2.3.5b}$$

$$z_{k+1} = y_k - 4x_k + 7\,. \tag{2.3.5c}$$

Of course, the solution should be exactly as before. However, Table 6 shows that applying the iteration scheme leads to a set of values which grow to infinity. Thus the iteration scheme quickly fails.

Strictly Diagonal Dominant

The difference in the two Jacobi schemes above involves the iteration procedure being strictly diagonal dominant. We begin with the definition of strict diagonal

dominance. A matrix $\mathbf{A}$ is strictly diagonal dominant if for each row, the sum of the absolute values of the off-diagonal terms is less than the absolute value of the diagonal term:

$$|a_{kk}| > \sum_{j=1, j\neq k}^{N} |a_{kj}| \,. \tag{2.3.6}$$

Strict diagonal dominance has the following consequence; given a stricly diagonal dominant matrix $\mathbf{A}$, then $\mathbf{Ax} = \mathbf{b}$ has a unique solution $\mathbf{x} = \mathbf{p}$. Jacobi iteration produces a sequence $\mathbf{p}_k$ that will converge to $\mathbf{p}$ for any $\mathbf{p}_0$. For the two examples considered here, this property is crucial. For the first example (2.3.1), we have

$$\mathbf{A} = \begin{pmatrix} 4 & -1 & 1 \\ 4 & -8 & 1 \\ -2 & 1 & 5 \end{pmatrix} \rightarrow \begin{array}{l} \text{row 1: } |4| > |-1| + |1| = 2 \\ \text{row 2: } |-8| > |4| + |1| = 5 \\ \text{row 3: } |5| > |2| + |1| = 3 \end{array} , \tag{2.3.7}$$

which shows the system to be strictly diagonal dominant and guaranteed to converge. In contrast, the second system (2.3.4) is not stricly diagonal dominant as can be seen from

$$\mathbf{A} = \begin{pmatrix} -2 & 1 & 5 \\ 4 & -8 & 1 \\ 4 & -1 & 1 \end{pmatrix} \rightarrow \begin{array}{l} \text{row 1: } |-2| < |1| + |5| = 6 \\ \text{row 2: } |-8| > |4| + |1| = 5 \\ \text{row 3: } |1| < |4| + |-1| = 5 \end{array} . \tag{2.3.8}$$

Thus this scheme is not guaranteed to converge. Indeed, it diverges to infinity.

Modification and Enhancements: Gauss-Seidel

It is sometimes possible to enhance the convergence of a scheme by applying modifications to the basic Jacobi scheme. For instance, the Jacobi scheme given by (2.3.3) can be enhanced by the following modifications

$$x_{k+1} = \frac{7 + y_k - z_k}{4} \tag{2.3.9a}$$

$$y_{k+1} = \frac{21 + 4x_{k+1} + z_k}{8} \tag{2.3.9b}$$

$$z_{k+1} = \frac{15 + 2x_{k+1} - y_{k+1}}{5} \,. \tag{2.3.9c}$$

Here use is made of the supposedly improved value x_{k+1} in the second equation and x_{k+1} and y_{k+1} in the third equation. This is known as the Gauss-Seidel scheme. Table 7 shows that the Gauss-Seidel procedure converges to the solution in half the number of iterations used by the Jacobi scheme.

Unlike the Jacobi scheme, the Gauss-Seidel method is not guaranteed to converge even in the case of strict diagonal dominance. Further, the Gauss-Seidel modification is only one of a large number of possible changes to the iteration

k	x_k	y_k	z_k
0	1.0	2.0	2.0
1	1.75	3.75	2.95
2	1.95	3.96875	2.98625
⋮	⋮	⋮	⋮
10	2.0	4.0	3.0

Table 7: Convergence of Gauss-Seidel iteration scheme to the solution value of $(x, y, z) = (2, 4, 3)$ from the initial guess $(x_0, y_0, z_0) = (1, 2, 2)$.

scheme which can be implemented in an effort to enhance convergence. It is also possible to use several previous iterations to achieve convergence. Krylov space methods [6] are often high end iterative techniques especially developed for rapid convergence. Included in these iteration schemes are conjugant gradient methods and generalized minimum residual methods which we will discuss and implement [6].

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

$$-4\psi_{mn} + \psi_{(m+1)n} + \psi_{(m-1)n} + \psi_{m(n+1)} + \psi_{m(n-1)} = \delta^2\omega_{mn}\,. \qquad (2.3.10)$$

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

$$\psi^{k+1}_{mn} = \frac{\psi^k_{(m+1)n} + \psi^k_{(m-1)n} + \psi^k_{m(n+1)} + \psi^k_{m(n-1)} - \delta^2\omega_{mn}}{4}\,. \qquad (2.3.11)$$

Note that the diagonal term had 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 non-zero diagonals D.

2. For each ψ_{mn}, perform the appropriate addition and subtractions. In this case there are 5 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.

2.4 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

$$\nabla^2 \psi = \omega \tag{2.4.1}$$

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 are defined as

$$F(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} f(x) dx \tag{2.4.2a}$$

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} F(k) dk \,. \tag{2.4.2b}$$

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

$$\widehat{f'(x)} = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} f'(x) dx = f(x) e^{ikx}|_{-\infty}^{\infty} - \frac{ik}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} f(x) dx \,. \quad (2.4.3)$$

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

$$\widehat{f'(x)} = -\frac{ik}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} f(x) dx = -ik\widehat{f(x)} \,. \quad (2.4.4)$$

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

$$\widehat{f^{(n)}} = (-ik)^n \widehat{f} \,. \quad (2.4.5)$$

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

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

$$y'' - \omega^2 y = -f(x) \quad x \in [-\infty, \infty] \,. \quad (2.4.6)$$

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

$$\begin{aligned} &\widehat{y''} - \omega^2 \widehat{y} = -\widehat{f} \\ &-k^2 \widehat{y} - \omega^2 \widehat{y} = -\widehat{f} \\ &(k^2 + \omega^2)\widehat{y} = \widehat{f} \\ &\widehat{y} = \frac{\widehat{f}}{k^2 + \omega^2} \,. \end{aligned} \quad (2.4.7)$$

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

$$y(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} \frac{\widehat{f}}{k^2 + \omega^2} dk \,. \quad (2.4.8)$$

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

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 [7]. Their algorithm was named one of the top ten algorithms of the 20th 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 $\exp(ikx)$ 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 [7] for a broader overview.

The Streamfunction

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

$$\nabla^2 \psi = \omega \tag{2.4.9}$$

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

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = \omega \,. \tag{2.4.10}$$

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:

$$\widehat{\frac{\partial^2 \psi}{\partial x^2}} + \widehat{\frac{\partial^2 \psi}{\partial y^2}} = \widehat{\omega} \quad \rightarrow \quad -k_x^2 \widehat{\psi} + \frac{\partial^2 \widehat{\psi}}{\partial y^2} = \widehat{\omega} \,, \tag{2.4.11}$$

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

$$-k_x^2 \widetilde{\widehat{\psi}} + \widetilde{\frac{\partial^2 \widehat{\psi}}{\partial y^2}} = \widetilde{\widehat{\omega}} \quad \rightarrow \quad -k_x^2 \widetilde{\widehat{\psi}} + -k_y^2 \widetilde{\widehat{\psi}} = \widetilde{\widehat{\omega}} \,. \tag{2.4.12}$$

This can be rearranged to obtain the final result

$$\widetilde{\widehat{\psi}} = -\frac{\widetilde{\widehat{\omega}}}{k_x^2 + k_y^2} \,. \tag{2.4.13}$$

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.

MATLAB commands

The commands for executing the Fast Fourier Transform and its inverse are as follows

- $fft(x)$**:** Forward Fourier transform a vector **x**.
- $ifft(x)$**:** Inverse Fourier transform a vector **x**.

2.5 Comparison of solution techniques for Ax=b: rules of thumb

The practical implementation of the mathematical tools available in MATLAB is crucial. This lecture will focus on the use of some of the more sophisticated routines in MATLAB 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.

Fast Fourier Transform: FFT, IFFT, FFTSHIFT, IFFTSHIFT2

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 **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 $x \in [0, L] \rightarrow [-L, 0]$

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:

$$f(x) = \exp(-\alpha x^2) \quad \to \quad \widehat{f}(k) = \frac{1}{\sqrt{2\alpha}} \exp\left(-\frac{k^2}{4\alpha}\right). \tag{2.5.1}$$

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

```
clear all; close all;  % clear all variables and figures

L=20;   % define the computational domain [-L/2,L/2]
n=128;  % define the number of Fourier modes 2^n

x2=linspace(-L/2,L/2,n+1);  % define the domain discretization
x=x2(1:n);   % consider only the first n points:  periodicity

u=exp(-x.*x);         % function to take a derivative of
ut=fft(u);            % FFT the function
utshift=fftshift(ut); % shift FFT

figure(1), plot(x,u)     % plot initial gaussian
figure(2), plot(abs(ut))      % plot unshifted transform
figure(3), plot(abs(utshift)) % plot shifted transform
```

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. 15 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 $f'(x) = -\text{sech}(x)\tanh(x)$.

```
clear all; close all;  % clear all variables and figures
```

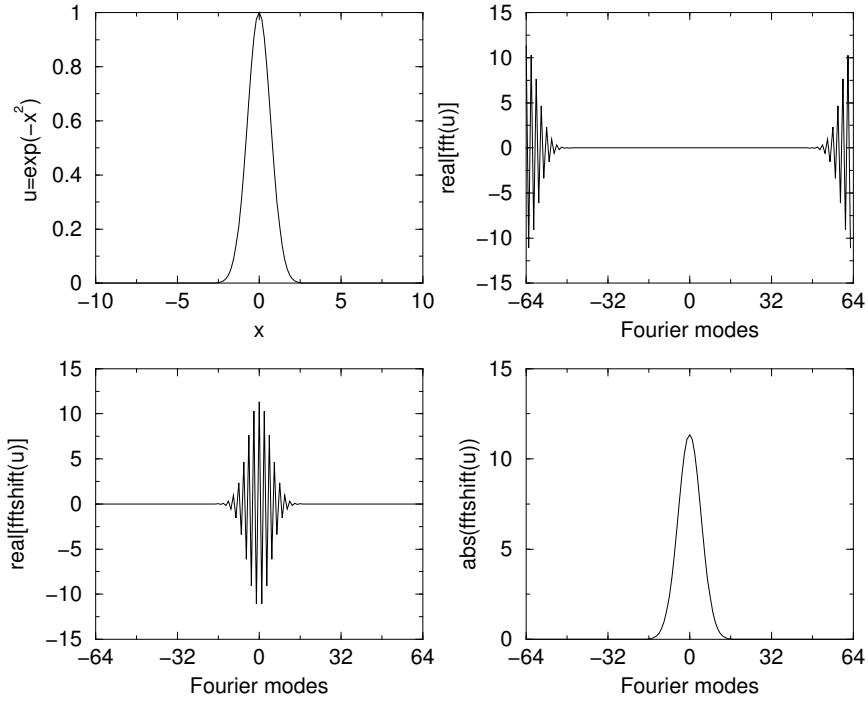

Figure 15: Fast Fourier Transform of Gaussian data illustrating the shifting properties of the FFT routine. Note that the *fftshift* command restores the transform to its mathematically correct, unshifted state.

```
L=20;   % define the computational domain [-L/2,L/2]
n=128;  % define the number of Fourier modes 2^n

x2=linspace(-L/2,L/2,n+1);  % define the domain discretization
x=x2(1:n);   % consider only the first n points:  periodicity
u=sech(x);     % function to take a derivative of
ut=fft(u);     % FFT the function
k=(2*pi/L)*[0:(n/2-1) (-n/2):-1]; % k rescaled to 2pi domain

ut1=i*k.*ut;         % first derivative
ut2=-k.*k.*ut;       % second derivative
ut3=-i*k.*k.*k.*ut;  % third derivative

u1=ifft(ut1); u2=ifft(ut2); u3=ifft(ut3); % inverse transform
u1exact=-sech(x).*tanh(x);  % analytic first derivative
```

```
figure(1)
plot(x,u,'r',x,u1,'g',x,u1exact,'go',x,u2,'b',x,u3,'c') % plot
```

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 MATLAB 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 direction 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.

Sparse Matrices: SPDIAG, SPY

Under discretization, most physical problems yield sparse matrices, i.e. matrices which are largely composed of zeros. For instance, the matrices (1.5.11) and (2.1.13) 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 at the nonzero components of the matrix structure. As an example, we construct the matrix given by (2.1.13) for the case of $N = 5$ in both the x and y directions.

```
clear all; close all;  % clear all variables and figures

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

e0=zeros(n,1);  % vector of zeros
e1=ones(n,1);   % vector of ones

e2=e1;    % copy the one vector
e4=e0;    % copy the zero vector
for j=1:m
 e2(m*j)=0;  % overwrite every m^th value with zero
 e4(m*j)=1;  % overwirte every m^th value with one
end

e3(2:n,1)=e2(1:n-1,1); e3(1,1)=e2(n,1); % shift to correct
e5(2:n,1)=e4(1:n-1,1); e5(1,1)=e4(n,1); % positions

% place diagonal elements
```

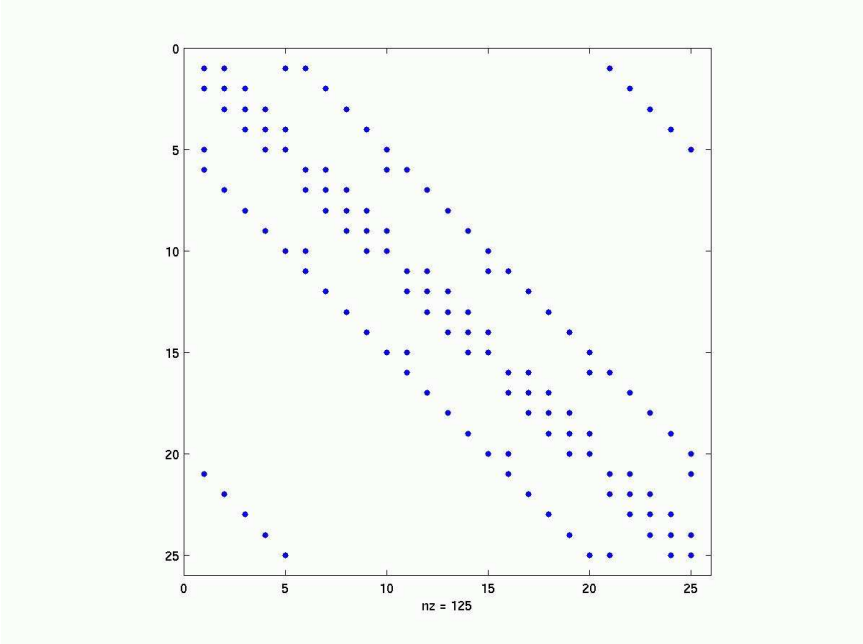

Figure 16: Sparse matrix structure for the Laplacian operator using second order discretization. The output is generated via the *spy* command in MATLAB.

```
matA=spdiags([e1 e1 e5 e2 -4*e1 e3 e4 e1 e1], ...
      [-(n-m) -m -m+1 -1 0 1 m-1 m (n-m)],n,n);
spy(matA) % view the matrix structure
```

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.

Iterative Methods: BICGSTAB, GMRES

Iterative techniques need also to be considered. There are a wide variety of built-in iterative techniques in MATLAB. 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 MATLAB.

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

```
>>x=gmres(A,b);
```

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

```
>>x=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,maxit,M1,M2,x0);
>>[x,flag,relres,iter]=gmres(A,b,restart,tol,maxit,M1,M2,x0);
```

We already know that the matrix $\mathbf{A}$ and vector $\mathbf{b}$ come from the original problem. The remaining parameters are as follows:

tol = specified tolerance for convergence
maxit = maximum number of iterations
M1, M2 = pre-conditioning 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.

2.6 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.

Streamfunction Equations: Nonuniqueness

We return to the consideration of the streamfunction equation

$$\nabla^2\psi = \omega \tag{2.6.1}$$

with the periodic boundary conditions

$$\psi(-L, y, t) = \psi(L, y, t) \tag{2.6.2a}$$
$$\psi(x, -L, t) = \psi(x, L, t)\,. \tag{2.6.2b}$$

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

$$\psi = \psi_0 + c\,, \tag{2.6.3}$$

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 (2.1.13), 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

$$u = -\tfrac{\partial\psi}{\partial y} \quad \text{(x-component of velocity)} \tag{2.6.4a}$$
$$v = \tfrac{\partial\psi}{\partial x} \quad \text{(y-component of velocity)}\,. \tag{2.6.4b}$$

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

$$\frac{\partial\omega}{\partial t} + [\psi, \omega] = \nu\nabla^2\omega \tag{2.6.5}$$

where

$$[\psi, \omega] = \frac{\partial\psi}{\partial x}\frac{\partial\omega}{\partial y} - \frac{\partial\psi}{\partial y}\frac{\partial\omega}{\partial x}\,, \tag{2.6.6}$$

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 (2.1.13). And in fact, this is the simplest thing to do. For instance, given the construction of the matrix (2.1.13), we could simply add the following line of MATLAB code:

```
A(1,1)=0;
```

Then the $\det \mathbf{A} \neq 0$ 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 $A(1,1) \neq -4$.

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

$$\nabla^2 \psi = \omega \tag{2.6.7}$$

to the equation (2.4.13)

$$\widetilde{\widehat{\psi}} = -\frac{\widetilde{\widehat{\omega}}}{k_x^2 + k_y^2}\,. \tag{2.6.8}$$

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 two.

An observation concerning (2.6.8) 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 (2.6.8) so that

$$\widetilde{\widehat{\psi}} = -\frac{\widetilde{\widehat{\omega}}}{k_x^2 + k_y^2 + eps} \tag{2.6.9}$$

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(1)=10^(-6);
ky(1)=10^(-6);
```

The values of $kx(1) = ky(1) = 0$ 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.

Sparse Derivative Matrices: Advection Terms

The sparse matrix (2.1.13) represents the discretization of the Laplacian which is accurate to second order. However, when calculating the advection terms given by

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x}\,, \tag{2.6.10}$$

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

$$\frac{\partial \omega}{\partial x} = \frac{\omega(x + \Delta x, y) - \omega(x - \Delta x, y)}{2\Delta x}\,. \tag{2.6.11}$$

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

$$\frac{\partial \omega_{mn}}{\partial x} = \frac{\omega_{(m+1)n} - \omega_{(m-1)n}}{2\Delta x}\,, \tag{2.6.12}$$

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

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

From these individual terms, the linear system $\partial\omega/\partial x = (1/2\Delta x)\mathbf{B}\omega$ can be constructed. The critical component is the sparse matrix $\mathbf{B}$. Note that we have

used the usual definition of the vector ω.

$$\omega = \begin{pmatrix} \omega_{11} \\ \omega_{12} \\ \vdots \\ \omega_{1n} \\ \omega_{21} \\ \omega_{22} \\ \vdots \\ \omega_{n(n-1)} \\ \omega_{nn} \end{pmatrix}. \tag{2.6.14}$$

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

$$\mathbf{B} = \begin{bmatrix} \mathbf{0} & \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} & -\mathbf{I} \\ -\mathbf{I} & \mathbf{0} & \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & \mathbf{0} & -\mathbf{I} & \mathbf{0} & \mathbf{I} \\ \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} & -\mathbf{I} & \mathbf{0} \end{bmatrix}, \tag{2.6.15}$$

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

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} = (\mathbf{B}\psi)(\mathbf{C}\omega) - (\mathbf{C}\psi)(\mathbf{B}\omega)\,. \tag{2.6.16}$$

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

3 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.

3.1 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

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} \tag{3.1.1}$$

with the periodic boundary conditions $u(-L) = u(L)$. We discretize the spatial derivative with a second-order scheme (see Table 2) so that

$$\frac{\partial u}{\partial t} = \frac{\kappa}{\Delta x^2}\left[u(x+\Delta x) - 2u(x) + u(x-\Delta x)\right]. \tag{3.1.2}$$

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.

The ODE System

To define the system of ODEs, we discretize and define the values of the vector $\mathbf{u}$ in the following way.

$$\begin{aligned}
&u(-L) = u_1\\
&u(-L+\Delta x) = u_2\\
&\vdots\\
&u(L-2\Delta x) = u_{n-1}\\
&u(L-\Delta x) = u_n\\
&u(L) = u_{n+1}\,.
\end{aligned}$$

Recall from periodicity that $u_1 = u_{n+1}$. Thus our system of differential equations solves for the vector

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}. \tag{3.1.3}$$

The governing equation (3.1.2) is then reformulated as the differential equations system

$$\frac{d\mathbf{u}}{dt} = \frac{\kappa}{\Delta x^2}\mathbf{A}\mathbf{u}\,, \tag{3.1.4}$$

where $\mathbf{A}$ is given by the sparse matrix

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 & 1 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & 0 & 1 & -2 & 1 \\ 1 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix}, \tag{3.1.5}$$

and the values of one on the upper right and lower left of the matrix result from the periodic boundary conditions.

MATLAB 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=ones(n,1);  % build a vector of ones
A=spdiags([e1 -2*e1 e1],[-1 0 1],n,n);  % diagonals
A(1,n)=1;  A(n,1)=1;  % periodic boundaries
```

2. Generate the desired initial condition vector $\mathbf{u} = \mathbf{u}_0$.

3. Call an ODE solver from the MATLAB suite. The matrix $\mathbf{A}$, the diffusion constant κ and spatial step Δx need to be passed into this routine.

```
[t,y]=ode45('rhs',tspan,u0,[],k,dx,A);
```

   The function *rhs.m* should be of the following form

```
function rhs=rhs(tspan,u,dummy,k,dx,A)
rhs=(k/dx^2)*A*u;
```

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 MATLAB.

2D MATLAB implementation

In the case of two dimensions, the calculation becomes slightly more difficult since the 2D data is represented by a matrix and the ODE solvers require a vector input for the initial data. For this case, the governing equation is

$$\frac{\partial u}{\partial t} = \kappa \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{3.1.6}$$

Discretizing in x and y gives a right hand side which takes on the form of (2.1.7). Provided $\Delta x = \Delta y = \delta$ are the same, the system can be reduced to the linear system

$$\frac{d\mathbf{u}}{dt} = \frac{\kappa}{\delta^2} \mathbf{A}\mathbf{u}\,, \tag{3.1.7}$$

where we have arranged the vector $\mathbf{u}$ in a similar fashion to (2.1.14) so that

$$\mathbf{u} = \begin{pmatrix} u_{11} \\ u_{12} \\ \vdots \\ u_{1n} \\ u_{21} \\ u_{22} \\ \vdots \\ u_{n(n-1)} \\ u_{nn} \end{pmatrix}, \tag{3.1.8}$$

where we have defined $u_{jk} = u(x_j, y_k)$. The matrix $\mathbf{A}$ is a nine diagonal matrix given by (2.1.13). 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 $\mathbf{A}$ (2.1.13).

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=linspace(-Lx/2,Lx/2,nx+1); % account for periodicity
x=x2(1:nx);      % x-vector
y2=linspace(-Ly/2,Ly/2,ny+1); % account for periodicity
y=y2(1:ny);      % y-vector

[X,Y]=meshgrid(x,y);  % set up for 2D initial conditions
U=exp(-X.^2-Y.^2);    % generate a Gaussian matrix
u=reshape(U,N,1);     % reshape into a vector
```

3. Call an ODE solver from the MATLAB suite. The matrix $\mathbf{A}$, the diffusion constant κ and spatial step $\Delta x = \Delta y = dx$ need to be passed into this routine.

   ```
   [t,y]=ode45('rhs',tspan,u0,[],k,dx,A);
   ```

   The function *rhs.m* should be of the following form

   ```
   function rhs=rhs(tspan,u,dummy,k,dx,A)
   rhs=(k/dx^2)*A*u;
   ```

4. 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.

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 17 depicts the process involved in the method of lines for the 1D case. The 2D case was illustrated previously in Fig. 13.

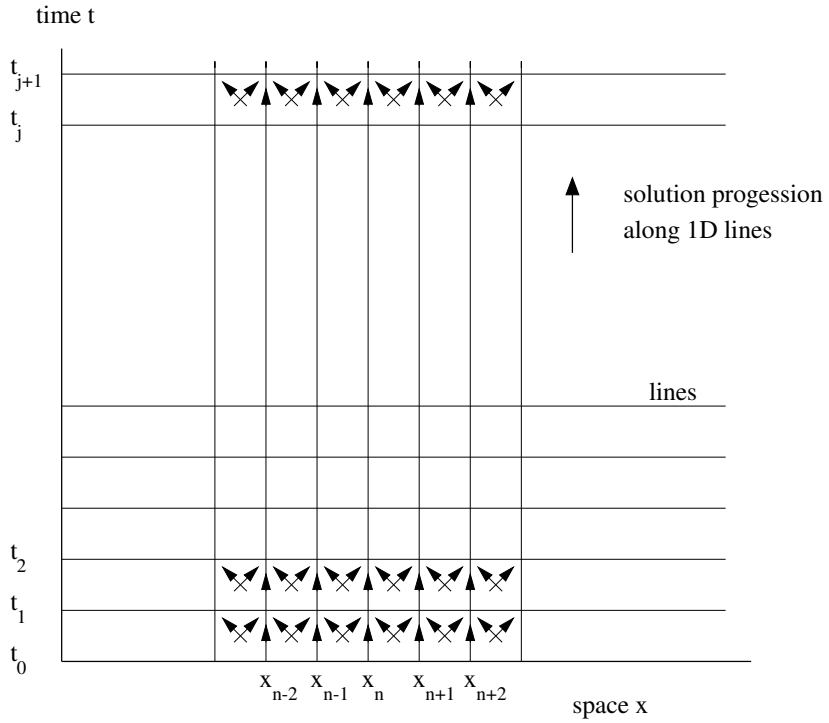

Figure 17: Graphical representation of the progression of the numerical solution using the method of lines. Here a second order discretization scheme is considered which couples each spatial point with its nearest neighbor.

3.2 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 most simple 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.

As an example, we consider the one-way wave equation

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x}\,. \tag{3.2.1}$$

The simplest discretization of this equation is to first central difference in the x

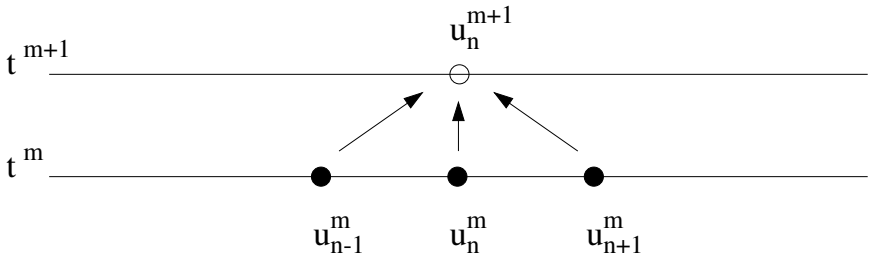

Figure 18: Four-point stencil for second-order spatial discretization and Euler time-stepping of the one-wave wave equation.

direction. This yields

$$\frac{\partial u_n}{\partial t} = \frac{c}{2\Delta x}\left(u_{n+1} - u_{n-1}\right) , \tag{3.2.2}$$

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

$$u_n^{(m+1)} = u_n^{(m)} + \frac{c\Delta t}{2\Delta x}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) . \tag{3.2.3}$$

This simple scheme has the four-point stencil shown in Fig. 18. To illustrate more clearly the iteration procedure, we rewrite the discretized equation in the form

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{3.2.4}$$

where

$$\lambda = \frac{c\Delta t}{\Delta x} \tag{3.2.5}$$

is known as the CFL (Courant, Friedrichs, and Lewy) condition [8]. 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 (3.2.5). 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

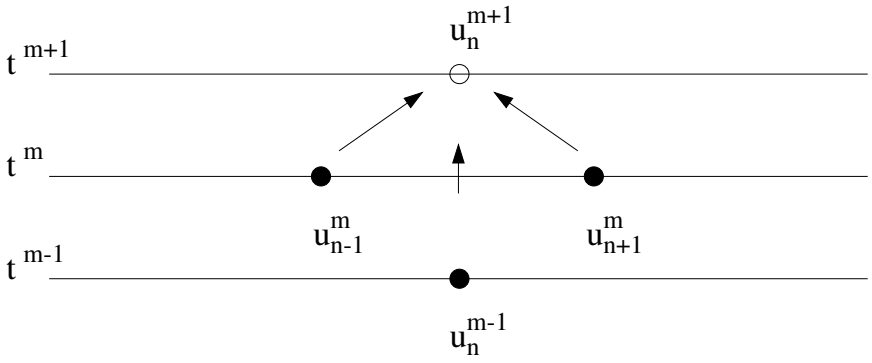

Figure 19: Four-point stencil for second-order spatial discretization and central-difference time-stepping of the one-wave wave equation.

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.

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 2. Thus after spatial discretization we have

$$\frac{\partial u_n}{\partial t} = \frac{c}{2\Delta x}\left(u_{n+1} - u_{n-1}\right) , \tag{3.2.6}$$

as before. And using a central difference scheme in time now yields

$$\frac{u_n^{(m+1)} - u_n^{(m-1)}}{2\Delta t} = \frac{c}{2\Delta x}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) . \tag{3.2.7}$$

This last expression can be rearranged to give

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) . \tag{3.2.8}$$

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. 19. 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.

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 3 gives

$$\frac{\partial u}{\partial x} = \frac{-u(x+2\Delta x) + 8u(x+\Delta x) - 8u(x-\Delta x) + u(x-2\Delta x)}{12\Delta x}\,. \qquad (3.2.9)$$

Combining this fourth-order spatial scheme with second-order central differencing in time gives the iterative scheme

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left[\frac{4}{3}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) - \frac{1}{6}\left(u_{n+2}^{(m)} - u_{n-2}^{(m)}\right)\right]. \qquad (3.2.10)$$

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.

Lax-Wendroff

Another alternative to discretizing in time and space involves a clever use of the Taylor expansion

$$u(x,t+\Delta t) = u(x,t) + \Delta t\frac{\partial u(x,t)}{\partial t} + \frac{\Delta t^2}{2!}\frac{\partial^2 u(x,t)}{\partial t^2} + O(\Delta t^3)\,. \qquad (3.2.11)$$

But we note from the governing one-way wave equation

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x} \approx \frac{c}{2\Delta x}\left(u_{n+1} - u_{n-1}\right). \qquad (3.2.12a)$$

Taking the derivative of the equation results in the relation

$$\frac{\partial^2 u}{\partial t^2} = c\frac{\partial^2 u}{\partial x^2} \approx \frac{c}{\Delta x^2}\left(u_{n+1} - 2u_n + u_{n-1}\right). \qquad (3.2.13a)$$

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

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) + \frac{\lambda^2}{2}\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right). \qquad (3.2.14)$$

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.

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 8: Stability of time-stepping schemes as a function of the CFL number.

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

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m+1)} - u_{n-1}^{(m+1)}\right). \tag{3.2.15}$$

This gives the tridiagonal system

$$u_n^{(m)} = -\frac{\lambda}{2}u_{n+1}^{(m+1)} + u_n^{(m+1)} + \frac{\lambda}{2}u_{n-1}^{(m+1)}, \tag{3.2.16}$$

which can be written in matrix form as

$$\mathbf{A}\mathbf{u}^{(m+1)} = \mathbf{u}^{(m)} \tag{3.2.17}$$

where

$$\mathbf{A} = \frac{1}{2}\begin{pmatrix} 2 & -\lambda & \cdots & 0 \\ \lambda & 2 & -\lambda & \cdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & \lambda & 2 \end{pmatrix}. \tag{3.2.18}$$

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.

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:

$$u_n^{(P)} = u_n^{(m)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{3.2.19a}$$

$$u_n^{(m+1)} = \frac{1}{2}\left[u_n^{(m)} + u_n^{(P)} + \lambda\left(u_{n+1}^{(P)} - u_{n-1}^{(P)}\right)\right]. \qquad (3.2.19b)$$

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 8. The stability of the various schemes hold 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.

3.3 Stability analysis

In the preceeding lecture, 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

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x}. \qquad (3.3.1)$$

Using Euler time-stepping and central differencing in space gives the iteration procedure

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \qquad (3.3.2)$$

where λ is the CFL number.

von Neumann Analysis

To determine the stability of a given scheme, we perform a *von Neumann analysis* [9]. This assumes the solution is of the form

$$u_n^{(m)} = g^m \exp(i\xi_n h) \quad \xi \in [-\pi/h, \pi/h] \qquad (3.3.3)$$

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

$$\lim_{m\to\infty} |g|^m \to \infty \qquad \text{unstable scheme} \qquad (3.3.4a)$$

$$\lim_{m\to\infty} |g|^m \leq 1 \ (< \infty) \qquad \text{stable scheme}. \qquad (3.3.4b)$$

Thus this stability check is very much like that performed for the time-stepping schemes developed for ordinary differential equations.

Forward Euler for One-Way Wave Equation

The Euler discretization of the one-way wave equation produces the iterative scheme (3.3.2). Plugging in the ansatz (3.3.3) gives the following relations:

$$\begin{aligned} g^{m+1}\exp(i\xi_n h) &= g^m \exp(i\xi_n h) + \frac{\lambda}{2}\left(g^m \exp(i\xi_{n+1}h) - g^m \exp(i\xi_{n-1}h)\right) \\ g^{m+1}\exp(i\xi_n h) &= g^m\left(\exp(i\xi_n h) + \frac{\lambda}{2}\left(\exp(i\xi_{n+1}h) - \exp(i\xi_{n-1}h)\right)\right) \\ g &= 1 + \frac{\lambda}{2}\left[\exp(ih(\xi_{n+1}-\xi_n)) - \exp(ih(\xi_{n-1}-\xi_n))\right]. \end{aligned} \tag{3.3.5}$$

Letting $\xi_n = n\zeta$ reduces the equations further to

$$\begin{aligned} g(\zeta) &= 1 + \frac{\lambda}{2}[\exp(i\zeta h) - \exp(-i\zeta h)] \\ g(\zeta) &= 1 + i\lambda \sin(\zeta h). \end{aligned} \tag{3.3.6}$$

From this we can deduce that

$$|g(\zeta)| = \sqrt{g^*(\zeta)g(\zeta)} = \sqrt{1+\lambda^2\sin^2\zeta h}. \tag{3.3.7}$$

Thus

$$|g(\zeta)| \geq 1 \quad \rightarrow \quad \lim_{m\to\infty}|g|^m \to \infty \quad \text{unstable scheme}. \tag{3.3.8}$$

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 a numerical instability.

Backward Euler for One-Way Wave Equation

The Euler discretization of the one-way wave equation produces the iterative scheme

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m+1)} - u_{n-1}^{(m+1)}\right). \tag{3.3.9}$$

Plugging in the ansatz (3.3.3) gives the following relations:

$$\begin{aligned} g^{m+1}\exp(i\xi_n h) &= g^m \exp(i\xi_n h) + \frac{\lambda}{2}\left(g^{m+1} \exp(i\xi_{n+1}h) - g^{m+1} \exp(i\xi_{n-1}h)\right) \\ g &= 1 + \frac{\lambda}{2} g\left[\exp(ih(\xi_{n+1}-\xi_n)) - \exp(ih(\xi_{n-1}-\xi_n))\right]. \end{aligned} \tag{3.3.10}$$

Letting $\xi_n = n\zeta$ reduces the equations further to

$$\begin{aligned} g(\zeta) &= 1 + \frac{\lambda}{2} g[\exp(i\zeta h) - \exp(-i\zeta h)] \\ g(\zeta) &= 1 + i\lambda g \sin(\zeta h) \\ g[1 - i\lambda \sin(\zeta h)] &= 1 \\ g(\zeta) &= \frac{1}{1 - i\lambda \sin\zeta h}. \end{aligned} \tag{3.3.11}$$

From this we can deduce that

$$|g(\zeta)| = \sqrt{g^*(\zeta)g(\zeta)} = \sqrt{\frac{1}{1+\lambda^2 \sin^2 \zeta h}}\,. \tag{3.3.12}$$

Thus

$$|g(\zeta)| \leq 1 \quad \rightarrow \quad \lim_{m\rightarrow\infty} |g|^m \leq 1 \quad \text{unconditionally stable}\,. \tag{3.3.13}$$

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.

Lax-Wendroff for One-Way Wave Equation

The Lax-Wendroff scheme is not as transparent as the forward and backward Euler schemes.

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) + \frac{\lambda^2}{2}\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right)\,. \tag{3.3.14}$$

Plugging in the standard ansatz (3.3.3) gives

$$\begin{aligned} g^{m+1}\exp(i\xi_n h) &= g^m \exp(i\xi_n h) + \frac{\lambda}{2} g^m \left(\exp(i\xi_{n+1}h) - \exp(i\xi_{n-1}h)\right) \\ &\quad + \frac{\lambda^2}{2} g^m \left(\exp(i\xi_{n+1}h) - 2\exp(i\xi_n h) + \exp(i\xi_{n-1}h)\right) \\ g &= 1 + \frac{\lambda}{2}\left[\exp(ih(\xi_{n+1}-\xi_n)) - \exp(ih(\xi_{n-1}-\xi_n))\right] \\ &\quad + \frac{\lambda^2}{2}\left[\exp(ih(\xi_{n+1}-\xi_n)) + \exp(ih(\xi_{n-1}-\xi_n)) - 2\right]\,. \end{aligned} \tag{3.3.15}$$

Letting $\xi_n = n\zeta$ reduces the equations further to

$$\begin{aligned} g(\zeta) &= 1 + \frac{\lambda}{2}[\exp(i\zeta h) - \exp(-i\zeta h)] + \frac{\lambda^2}{2}[\exp(i\zeta h) + \exp(-i\zeta h) - 2] \\ g(\zeta) &= 1 + i\lambda \sin\zeta h + \lambda^2(\cos\zeta h - 1) \\ g(\zeta) &= 1 + i\lambda \sin\zeta h - 2\lambda^2 \sin^2(\zeta h/2)\,. \end{aligned} \tag{3.3.16}$$

This results in the relation

$$|g(\zeta)|^2 = \lambda^4\left[4\sin^4(\zeta h/2)\right] + \lambda^2\left[\sin^2\zeta h - 4\sin^2(\zeta h/2)\right] + 1\,. \tag{3.3.17}$$

This expression determines the range of values for the CFL number λ for which $|g| \leq 1$. Ultimately, the stability of the Lax-Wendroff scheme for the one-way wave equation is determined.

Leap-Frog (2,2) for One-Way Wave Equation

The leap-frog discretization for the one-way wave equation yields the iteration scheme

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda \left(u_{n+1}^{(m)} - u_{n-1}^{(m)} \right) . \tag{3.3.18}$$

Plugging in the ansatz (3.3.3) gives the following relations:

$$\begin{aligned} g^{m+1} \exp(i\xi_n h) &= g^{m-1} \exp(i\xi_n h) + \lambda g^m \left(\exp(i\xi_{n+1} h) - \exp(i\xi_{n-1} h) \right) \\ g^2 &= 1 + \lambda g \left[\exp(ih(\xi_{n+1} - \xi_n)) - \exp(ih(\xi_{n-1} - \xi_n)) \right] . \end{aligned} \tag{3.3.19}$$

Letting $\xi_n = n\zeta$ reduces the equations further to

$$\begin{aligned} g^2 &= 1 + \lambda g [\exp(i\zeta h) - \exp(-i\zeta h)] \\ g - \frac{1}{g} &= 2i\lambda \sin(\zeta h) . \end{aligned} \tag{3.3.20}$$

From this we can deduce that

$$|g(\zeta)| = \sqrt{g^*(\zeta) g(\zeta)} = \sqrt{\frac{1}{1 + \lambda^2 \sin^2 \zeta h}} . \tag{3.3.21}$$

Thus to assure stability, it can be shown that we require

$$\lambda \leq 1 . \tag{3.3.22}$$

Thus the leap-frog (2,2) time-stepping scheme is stable for values of $\lambda \leq 1$. 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.

3.4 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

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x}\,. \tag{3.4.1}$$

The leap-frog discretization for the one-way wave equation yielded the iteration scheme

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right)\,, \tag{3.4.2}$$

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

$$g - \frac{1}{g} = 2i\lambda\sin(\zeta h)\,, \tag{3.4.3}$$

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.

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 than 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

$$\frac{\partial u}{\partial t} = c\frac{\partial^2 u}{\partial x^2}\,. \tag{3.4.4}$$

Discretizing the spatial second derivative yields

$$\frac{\partial u_n^{(m)}}{\partial t} = \frac{c}{\Delta x^2}\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right)\,. \tag{3.4.5}$$

Second order center differencing in time then yields

$$u_n^{(m+1)} = u_n^{(m-1)} + 2\lambda\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right)\,, \tag{3.4.6}$$

where now the CFL number is given by

$$\lambda = \frac{c\Delta t}{\Delta x^2}\,. \tag{3.4.7}$$

In this case, to double the accuracy and hold the CFL number constant requires cutting the time-step by a factor of four. 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

$$\begin{aligned} g^{m+1}\exp(i\xi_n h) &= g^{m-1}\exp(i\xi_n h) \\ &\quad +2\lambda g^m\left(\exp(i\xi_{n+1}h) - 2\exp(i\xi_n h) + \exp(i\xi_{n-1}h)\right) \\ g - \frac{1}{g} &= 2\lambda(\exp(i\zeta h) + \exp(-i\zeta h) - 2) \\ g - \frac{1}{g} &= 2\lambda(i\sin(\zeta h) - 1)\,, \end{aligned} \tag{3.4.8}$$

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

$$u_n^{(m+1)} = u_n^{(m)} + \lambda\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right)\,. \tag{3.4.9}$$

The von Neumann stability analysis based upon the ansatz $u_n^{(m)} = g^m \exp(i\xi_n h)$ results in the expression

$$g = 1 + \lambda(i\sin(\zeta h) - 1)\,, \tag{3.4.10}$$

This gives

$$|g| = 1 - 2\lambda + \lambda^2(1 + \sin^2\zeta h)\,, \tag{3.4.11}$$

so that

$$|g| \leq 1 \quad \text{for} \quad \lambda \leq \frac{1}{2}\,. \tag{3.4.12}$$

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 four.

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

$$\frac{\partial u}{\partial t} = -c\frac{\partial^4 u}{\partial x^4}\,. \tag{3.4.13}$$

Using a forward Euler method in time and a central-difference scheme in space gives the iteration scheme

$$u_n^{(m+1)} = u_n^{(m)} - \lambda \left(u_{n+2}^{(m)} - 4u_{n+1}^{(m)} + 6u_n^{(m)} - 4u_{n-1}^{(m)} + u_{n-2}^{(m)} \right) , \qquad (3.4.14)$$

where the CFL number λ is now given by

$$\lambda = \frac{c\Delta t}{\Delta x^4} . \qquad (3.4.15)$$

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 (3.4.13) with Fast Fourier Transforms. Transforming the equation in the x direction gives

$$\frac{\partial \widehat{u}}{\partial t} = -c(ik)^4 \widehat{u} = -ck^4 \widehat{u} . \qquad (3.4.16)$$

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

$$(k_{max})^4 = (64)^4 = 16.8 \times 10^6 , \qquad (3.4.17)$$

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 MATLAB 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.

3.5 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:

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x} \qquad \text{one-way wave equation} \tag{3.5.1a}$$

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \qquad \text{diffusion equation} \tag{3.5.1b}$$

$$i\frac{\partial u}{\partial t} = \frac{1}{2}\frac{\partial^2 u}{\partial x^2} + |u|^2 u \qquad \text{nonlinear Schrödinger equation}. \tag{3.5.1c}$$

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

$$\frac{\partial u}{\partial x} \to \frac{1}{2\Delta x}\begin{bmatrix} 0 & 1 & 0 & \cdots & 0 & -1 \\ -1 & 0 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & 0 & -1 & 0 & 1 \\ 1 & 0 & \cdots & 0 & -1 & 0 \end{bmatrix}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} \tag{3.5.2}$$

and

$$\frac{\partial^2 u}{\partial x^2} \to \frac{1}{\Delta x^2}\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 & 1 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & 0 & 1 & -2 & 1 \\ 1 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}. \tag{3.5.3}$$

From the previous lectures, we have the following discretization schemes for the one-way wave equation (3.5.1):

$$\text{Euler (unstable):} \qquad u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{3.5.4a}$$

$$\text{leap-frog (2,2) (stable for } \lambda \le 1\text{):} \quad u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{3.5.4b}$$

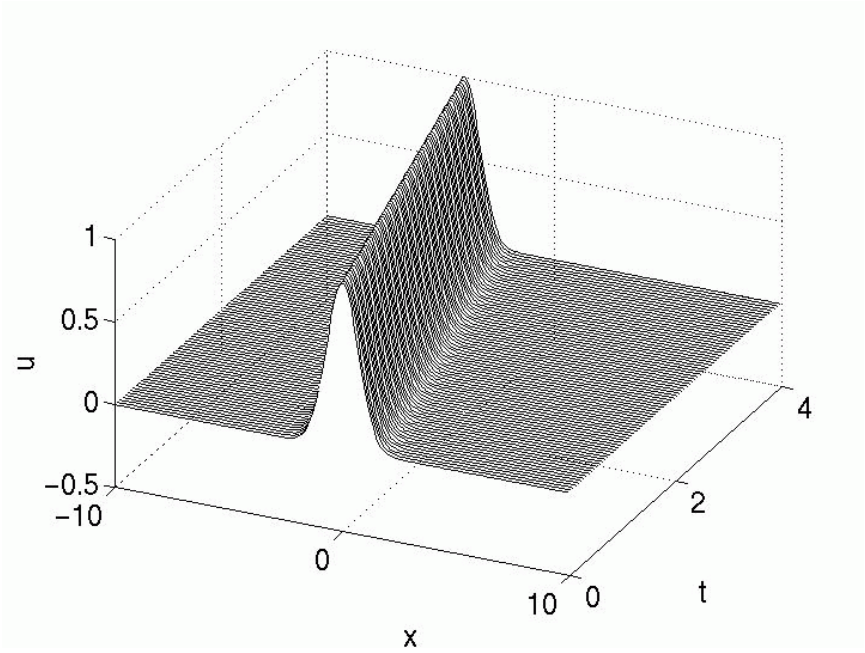

Figure 20: Evolution of the one-way wave equation with the leap-frog (2,2) scheme and with CFL=0.5. The stable traveling wave solution is propagated in this case to the left.

where the CFL number is given by $\lambda = \Delta t/\Delta x$. Similarly for the diffusion equation (3.5.1)

$$\text{Euler (stable for } \lambda \le 1/2\text{):} \quad u_n^{(m+1)} = u_n^{(m)} + \lambda\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right) \tag{3.5.5a}$$

$$\text{leap-frog (2,2) (unstable):} \quad u_n^{(m+1)} = u_n^{(m-1)} + 2\lambda\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right) \tag{3.5.5b}$$

where now the CFL number is given by $\lambda = \Delta t/\Delta x^2$. The nonlinear Schrödinger equation discretizes to the following form:

$$\frac{\partial u_n^{(m)}}{\partial t} = -\frac{i}{2\Delta x^2}\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right) - i|u_n^{(m)}|^2 u_n^{(m)}\,. \tag{3.5.6}$$

We will explore Euler and leap-frog (2,2) time-stepping with this equation.

One-Way Wave Equation

We first consider the leap-frog (2,2) scheme applied to the one-way wave equation. Figure 20 depicts the evolution of an initial Gaussian pulse. For this case,

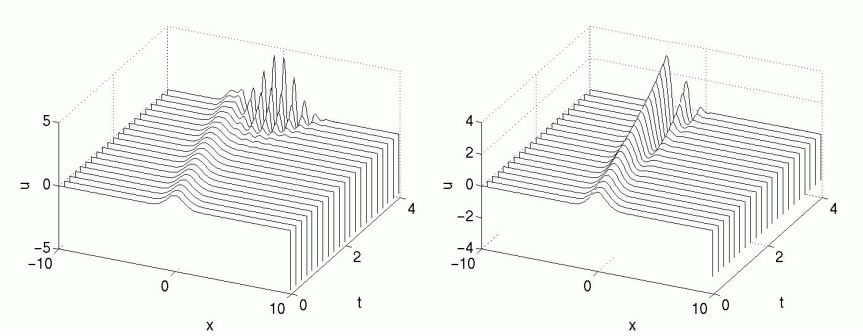

Figure 21: Evolution of the one-way wave equation using the leap-frog (2,2) scheme with CFL=2 (left) along with the Euler time-stepping scheme (right). The analysis predicts stable evolution of leap-frog provided the CFL≤ 1. Thus the onset of numerical instability near $t \approx 3$ for the CFL=2 case is not surprising. Likewise, the Euler scheme is expected to be unstable for all CFL.

the CFL=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 $\lambda \geq 1$ and the system is always unstable for the Euler time-stepping scheme. Figure 21 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. 20. 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 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 MATLAB code used to generate the leap-frog and Euler iterative solutions is given by

```
clear all; close all;  % clear previous figures and values

% initialize grid size, time, and CFL number
Time=4;
L=20;
n=200;
x2=linspace(-L/2,L/2,n+1);
x=x2(1:n);
dx=x(2)-x(1);
dt=0.2;
CFL=dt/dx
time_steps=Time/dt;
t=0:dt:Time;
```

```
% initial conditions
u0=exp(-x.^2)';
u1=exp(-(x+dt).^2)';
usol(:,1)=u0;
usol(:,2)=u1;

% sparse matrix for derivative term
e1=ones(n,1);
A=spdiags([-e1  e1],[-1 1],n,n);
A(1,n)=-1;  A(n,1)=1;

% leap frog (2,2) or euler iteration scheme
for j=1:time_steps-1
%  u2 = u0 + CFL*A*u1;  % leap frog (2,2)
%  u0 = u1; u1 = u2;    % leap frog (2,2)
  u2 = u1 + 0.5*CFL*A*u1;  % euler
  u1 = u2;                 % euler
  usol(:,j+2)=u2;
end

% plot the data
waterfall(x,t,usol');
map=[0 0 0];
colormap(map);

% set x and y limits and fontsize
set(gca,'Xlim',[-L/2 L/2],'Xtick',[-L/2 0 L/2],'FontSize',[20]);
set(gca,'Ylim',[0 Time],'ytick',[0 Time/2 Time],'FontSize',[20]);
view(25,40)

% set axis labels and fonts
xl=xlabel('x'); yl=ylabel('t'); zl=zlabel('u');
set(xl,'FontSize',[20]);set(yl,'FontSize',[20]);set(zl,'FontSize',[20]);

print -djpeg -r0 fig.jpg % print jpeg at screen resolution
```

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 22 shows the expected diffusion behavior for the stable Euler scheme ($\lambda \leq 0.5$). In contrast, Fig. 23 shows the numerical instabilities which are

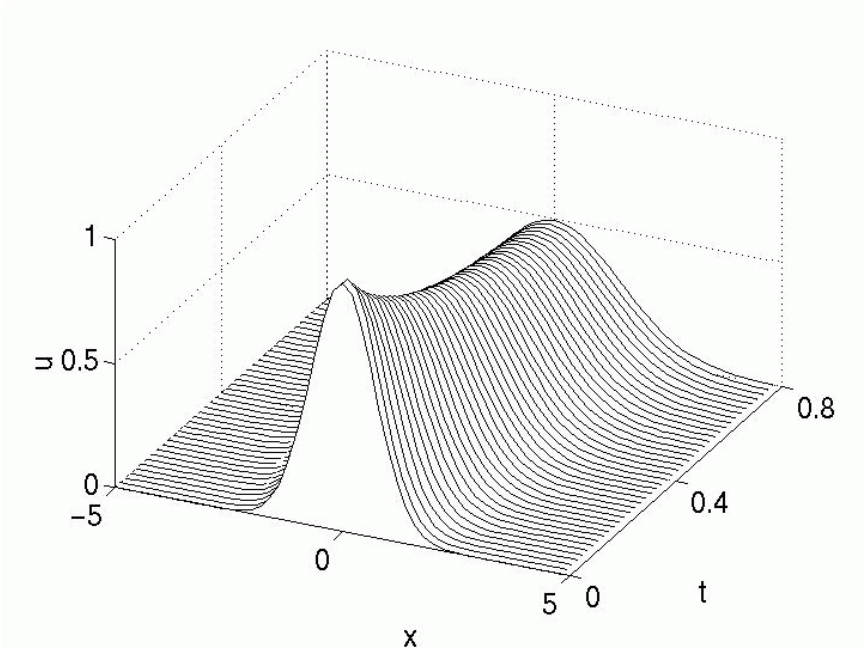

Figure 22: Stable evolution of the heat equation with the Euler scheme with CFL=0.5. The initial Gaussian is diffused in this case.

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

```
% sparse matrix for second derivative term
e1=ones(n,1);
A=spdiags([e1 -2*e1 e1],[-1 0 1],n,n);
A(1,n)=1;  A(n,1)=1;
```

Further, the iterative process is now

```
% leap frog (2,2) or euler iteration scheme
for j=1:time_steps-1
  u2 = u0 + 2*CFL*A*u1;  % leap frog (2,2)
  u0 = u1; u1 = u2;      % leap frog (2,2)
%  u2 = u1 + CFL*A*u1;  % euler
%  u1 = u2;             % euler
  usol(:,j+2)=u2;
end
```

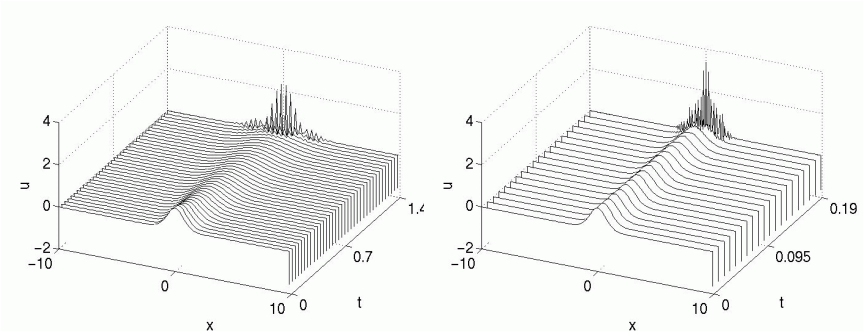

Figure 23: Evolution of the heat equation with the Euler time-stepping scheme (left) and leap-frog (2,2) scheme (right) with CFL=1. The analysis predicts that both these schemes are unstable. Thus the onset of numerical instability is observed.

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.

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 24 shows the evolution of the exact one-soliton solution of the nonlinear Schrödinger equation ($u(x,0) = \mathrm{sech}(x)$) 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 MATLAB 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 + -i*CFL*A*u1- i*2*dt*(conj(u1).*u1).*u1;
u0 = u1; u1 = u2;
```

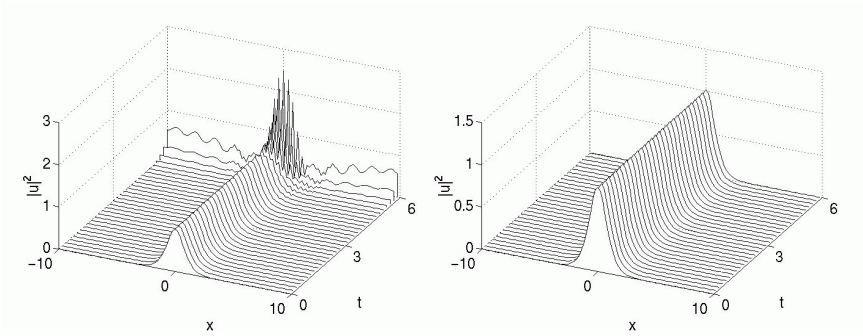

Figure 24: Evolution of the nonlinear Schrödinger equation with the Euler time-stepping scheme (left) and leap-frog (2,2) scheme (right) with CFL=0.05.

Note that i is automatically defined in MATLAB 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 not careful with this definition.

4 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.

4.1 Fast-Fourier Transforms and Cosine/Sine transform

From our previous lectures, 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 lecture explores the underlying mathematical reasons for such performance.

One of the abstract definitions of a Fourier transform pair is given by

$$F(k) = \int_{-\infty}^{\infty} e^{-ikx} f(x) dx \tag{4.1.1a}$$

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{ikx} F(k) dk \,. \tag{4.1.1b}$$

On a practical level, the value of the Fourier transform revolves squarely around the derivative relationship

$$\widehat{f^{(n)}} = (ik)^n \widehat{f} \tag{4.1.2}$$

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

$$F(k) = \sum_{n=1}^{N} f(n) \exp\left[-i\frac{2\pi(k-1)}{N}(n-1)\right] \quad 1 \le k \le N \tag{4.1.3a}$$

$$f(n) = \frac{1}{N}\sum_{k=1}^{N} F(k) \exp\left[i\frac{2\pi(k-1)}{N}(n-1)\right] \quad 1 \le n \le N\,. \tag{4.1.3b}$$

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

$$w^{nk} = \exp\left(\frac{2i\pi(k\text{–}1)(n\text{–}1)}{N}\right) = \cos\left(\frac{2\pi(k\text{–}1)(n\text{–}1)}{N}\right) + i\sin\left(\frac{2\pi(k\text{–}1)(n\text{–}1)}{N}\right), \tag{4.1.4}$$

then the Fourier transform results in the expression

$$F_n = \sum_{k=0}^{N-1} w^{nk} f_k \qquad 0 \le n \le N-1\,. \tag{4.1.5}$$

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. 25. 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 9 illustrates the three different expansions and their associated boundary conditions. The appropriate MATLAB command is also given.

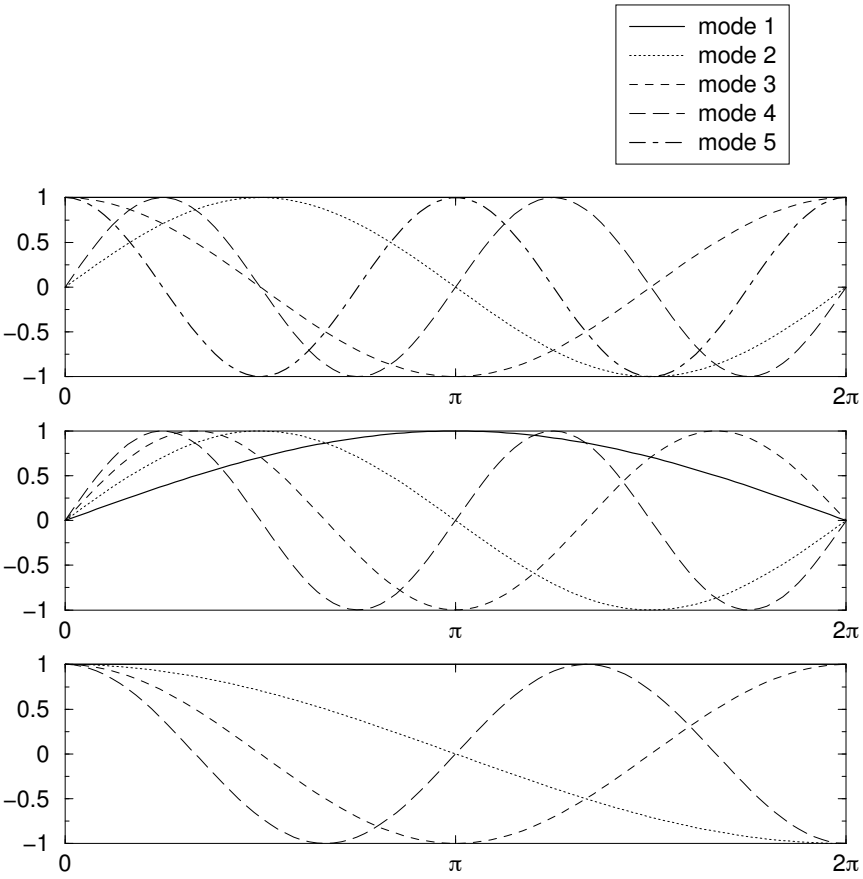

Figure 25: Basis functions used for a Fourier mode expansion (top), a sine expansion (middle), and a cosine expansion (bottom).

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

$$(F_N)_{jk} = w_n^{jk} = \exp(i2\pi jk/N) \tag{4.1.6}$$

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

$$w_{2n}^2 = w_n \,, \tag{4.1.7}$$

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 9: 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.

which is easy to show since $w_n = \exp(i2\pi/n)$ and

$$w_{2n}^2 = \exp(i2\pi/(2n))\exp(i2\pi/(2n)) = \exp(i2\pi/n) = w_n\,. \tag{4.1.8}$$

The consequences to this simple relationship are enormous and at the core of the success of the FFT algorithm. Essentially, the FFT is a matrix operation

$$\mathbf{y} = \mathbf{F}_N\mathbf{x} \tag{4.1.9}$$

which can now be split into two separate operations. Thus defining

$$\mathbf{x}^e = \begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ \vdots \\ x_{N-2} \end{pmatrix} \quad \text{and} \quad \mathbf{x}^o = \begin{pmatrix} x_1 \\ x_3 \\ x_5 \\ \vdots \\ x_{N-1} \end{pmatrix} \tag{4.1.10}$$

which are both vectors of length $M = N/2$, we can form the two $M \times M$ systems

$$\mathbf{y}^e = \mathbf{F}_M\mathbf{x}^e \quad \text{and} \quad \mathbf{y}^o = \mathbf{F}_M\mathbf{x}^o \tag{4.1.11}$$

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

$$y_n = y_n^e + w_N^n y_n^o \quad n = 0, 1, 2, ..., M-1 \tag{4.1.12a}$$
$$y_{n+M} = y_n^e - w_N^n y_n^o \quad n = 0, 1, 2, ..., M-1\,. \tag{4.1.12b}$$

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 [7, 11].

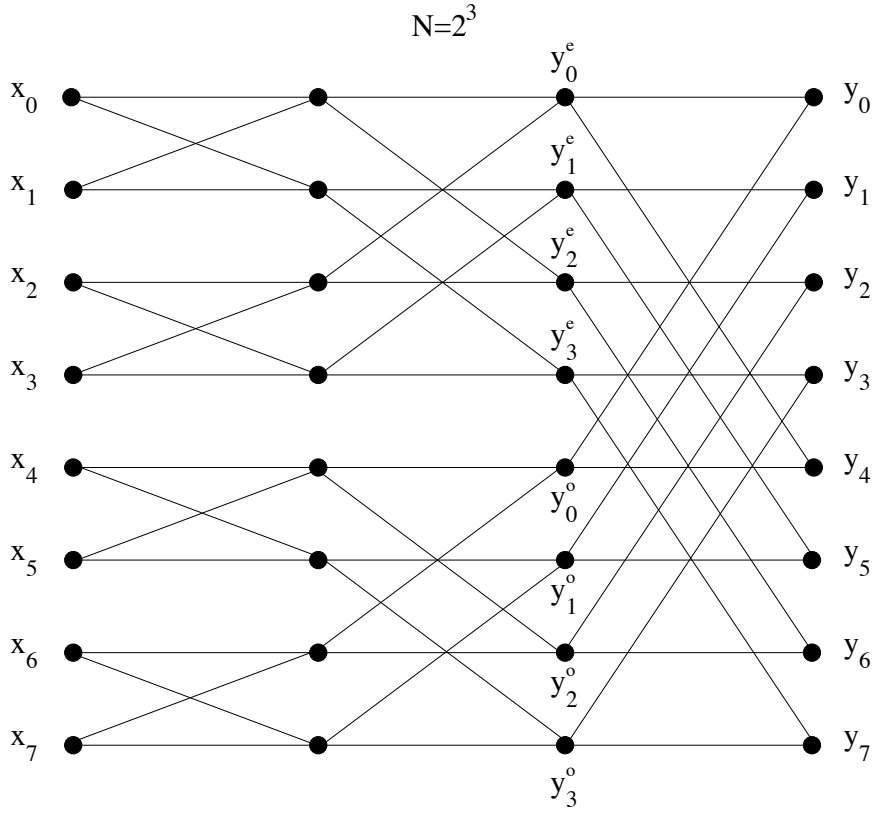

Figure 26: Graphical description of the Fast Fourier Transform process which systematically continues to factor the problem in two. This process allows the FFT routine to drop to $O(N \log N)$ operations.

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 two, 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. 26 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 $O(N \log N)$.

4.2 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.

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

$$\sqrt{1-x^2}\frac{d}{dx}\left(\sqrt{1-x^2}\frac{dT_n}{dx}\right) + n^2 T_n = 0 \quad x \in [-1,1]\,. \tag{4.2.1}$$

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:

$$\int_{-1}^{1} (1-x^2)^{-1/2} T_n(x) T_m(x) dx = \frac{\pi}{2} c_n \delta_{nm} \tag{4.2.2}$$

   where $c_0 = 2, c_n = 1 (n > 0)$ and δ_{nm} is the Delta function
4. eigenfunctions form a complete basis

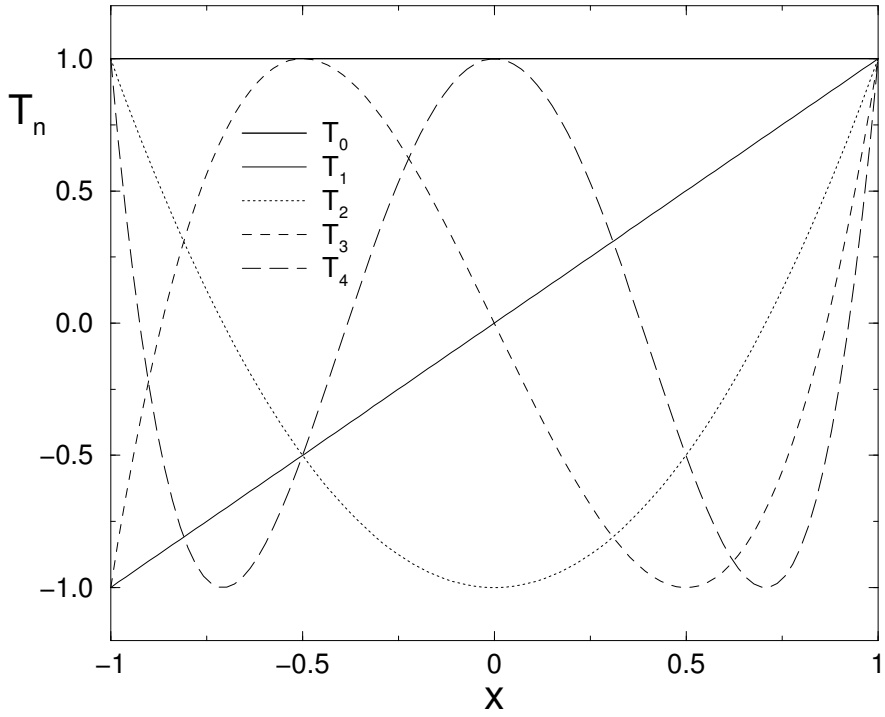

Figure 27: The first five Chebychev polynomials over the the interval of definition $x \in [-1, 1]$.

Each Chebychev polynomial (of degree n) is defined by

$$T_n(\cos\theta) = \cos n\theta\,. \tag{4.2.3}$$

Thus we find

$$T_0(x) = 1 \tag{4.2.4a}$$
$$T_1(x) = x \tag{4.2.4b}$$
$$T_2(x) = 2x^2 - 1 \tag{4.2.4c}$$
$$T_3(x) = 4x^3 - 3x \tag{4.2.4d}$$
$$T_4(x) = 8x^4 - 8x^2 + 1\,. \tag{4.2.4e}$$

The behavior of the first five Chebychev polynomials is illustrated in Fig. 27.

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. 27, 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 yields the Chebychev expansion scheme an $O(N \log N)$ transformation. Specifically, we transform from the interval $x \in [-1, 1]$ by letting

$$x = \cos\theta \qquad \theta \in [0, \pi]\,. \tag{4.2.5}$$

Thus when considering a function $f(x)$, we have $f(\cos\theta) = g(\theta)$. Under differentiation we find

$$\frac{dg}{d\theta} = -f' \cdot \sin\theta \tag{4.2.6}$$

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

$$f(x) = \sum_{k=0}^{\infty} a_k T_k(x) \tag{4.2.7}$$

where the coefficients a_k are determined from orthogonality and inner products to be

$$a_k = \int_{-1}^{1} \frac{1}{\sqrt{1-x^2}} f(x) T_k(x) dx\,. \tag{4.2.8}$$

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

$$x_m = \cos\left(\frac{(2m-1)\pi}{2n}\right) \qquad m = 1, 2, ..., n\,. \tag{4.2.9}$$

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

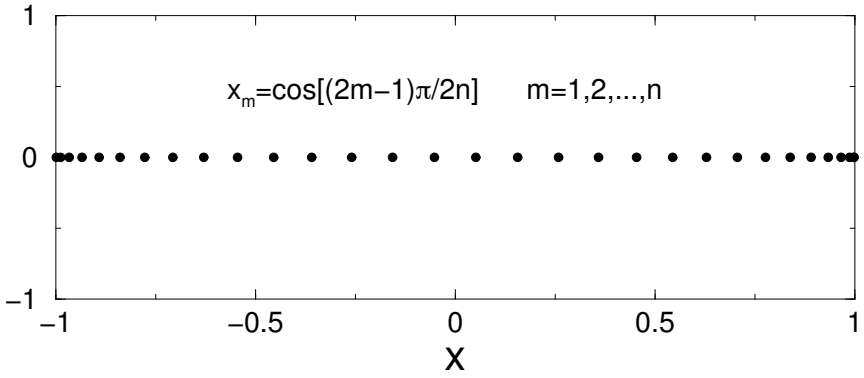

Figure 28: Clustered grid generation for $n = 30$ points using the Chebychev polynomials. Note that although the points are uniformaly spaced in θ, they are clustered due to the fact that $x_m = \cos[(2m-1)\pi/2n]$ where $m = 1, 2, ..., n$.

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. 28. So as the resolution is increased, it is important to be aware that the resolution increase is not uniform across the computational domain.

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 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

$$Lf(x) = \sum_{n=0}^{\infty} b_n T_n(x)\,, \tag{4.2.10}$$

then with $f(x) = \sum_{n=0}^{\infty} a_n T_n(x)$ we find

- $Lf = f'(x) : c_n b_n = 2\sum_{p=n+1(p+nodd)}^{\infty} p a_p$
- $Lf = f''(x) : c_n b_n = \sum_{p=n+2(p+neven)}^{\infty}$
- $Lf = xf(x) : b_n = (c_{n-1}a_{n-1} + a_{n+1})/2$

- $Lf = x^2 f(x) : b_n = (c_{n-2}a_{n-2} + (c_n + c_{n-1})a_n + a_{n+2})/4$

where $c_0 = 2, c_n = 0(n < 0), c_n = 1(n > 0), d_n = 1(n \geq 0)$, and $d_n = 0(n < 0)$.

4.3 Spectral method implementation

In this lecture, we develop an algorithm which implements a spectral method solution technique. We begin by considering the general partial differential equation

$$\frac{\partial u}{\partial t} = Lu + N(u) \tag{4.3.1}$$

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 non-constant 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

$$\frac{d\widehat{u}}{dt} = \alpha(k)\widehat{u} + \widehat{N(u)}\,. \tag{4.3.2}$$

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:

$$Lu = a\frac{d^2u}{dx^2} + b\frac{du}{dx} + cu\,. \tag{4.3.3}$$

then upon transforming this becomes

$$\begin{aligned}(ik)^2 a\widehat{u} + b(ik)\widehat{u} + c\widehat{u} \\ &= (-k^2 a + ibk + c)\widehat{u} \\ &= \alpha(k)\widehat{u}\,.\end{aligned} \tag{4.3.4}$$

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 still they are relatively easy. Consider the following examples

1. $f(x)du/dx$
   - determine $du/dx \rightarrow \widehat{du/dx} = ik\widehat{u}, du/dx = FFT^{-1}(ik\widehat{u})$
   - multiply by $f(x) \rightarrow f(x)du/dx$
   - Fourier transform $FFT(f(x)du/dx)$
2. u^3

- Fourier transform $FFT(u^3)$

3. $u^3 d^2u/dx^2$
   - determine $d^2u/dx^2 \rightarrow \widehat{d^2u/dx^2} = (ik)^2\widehat{u}, d^2u/dx^2 = FFT^{-1}(-k^2\widehat{u})$
   - multiply by $u^3 \rightarrow u^3 d^2u/dx^2$
   - Fourier transform $FFT(u^3 d^2u/dx^2)$

These examples give an outline of how the nonlinear, non-constant 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 MATLAB codes presented here solve the equations using the FFT for both the streamfunction and vorticity evolution. We begin by initializing the appropriate numerical parameters

```
clear all; close all;

nu=0.001;
Lx=20; Ly=20; nx=64; ny=64; N=nx*ny;

x2=linspace(-Lx/2,Lx/2,nx+1); x=x2(1:nx);
y2=linspace(-Ly/2,Ly/2,ny+1); y=y2(1:ny);
```

Thus the computational domain is for $x \in [-10, 10]$ 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]=meshgrid(x,y);
w=1*exp(-0.25*X.^2-Y.^2);
figure(1), pcolor(abs(w)); shading interp; colorbar; drawnow
```

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 (2.6.8) spectrally. Once the streamfunction is determined, a call is made to a time-stepping scheme which steps the vorticity solution forward in time. In this example, the streamfunction is updated every $\Delta t = 0.5$. The loop is run through ten times for a total time of $t = 5$.

```
% SPECTRAL K VALUES
kx=(2*pi/Lx)*[0:(nx/2-1) (-nx/2):-1]'; kx(1)=10^(-6);
ky=(2*pi/Ly)*[0:(ny/2-1) (-ny/2):-1]'; ky(1)=10^(-6);
```

```
%  Streamfunction
for stream_loop=1:10

wt=fft2(w)/N;
for j=1:ny
  psit(:,j)=-wt(:,j)./(kx.^2+ky(j)^2);
end
for j=1:nx
  psitx(j,:)=i*(psit(j,:).*kx');
end
psix=real(ifft2(psitx*N));
for j=1:ny
  psity(:,j)=i*(psit(:,j).*ky);
end
psiy=real(ifft2(psity*N));

wt2=reshape(wt,N,1);
[t,wsol]=ode23('spc_rhs',[0 0.5],wt2,[],psix,psiy,kx,ky,nu,nx,ny,N);

wt2=wsol(end,:)';
wt=reshape(wt2,nx,ny);
w=ifft2(wt*N);

figure(1)
pcolor(abs(w)); shading interp; colorbar; drawnow

end
```

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. Aside from the first function call line, the rest of the routine first finds the derivatives of the vorticity with respect to x and y respectively. The derivatives of the streamfunction are sent into the program. Once these terms are determined, the matrix components are reshaped and a large system of differential equations is solved.

```
function rhs=spc_rhs(tspan,wt2,dummy,psix,psiy,kx,ky,nu,nx,ny,N);
wt=reshape(wt2,nx,ny);

% w_x
for j=1:nx
  wtx(j,:)=i*(wt(j,:).*kx');
end
wx=ifft2(wtx*N);
```

```
% w_y
for j=1:ny
  wty(:,j)=i*(wt(:,j).*ky);
end
wy=ifft2(wty*N);

% transform w_x*psi_y and w_y*psi_x and reshape
wxpyt=fft2(wx.*psiy)/N;
wypxt=fft2(wy.*psix)/N;

wxpyt2=reshape(wxpyt,N,1);
wypxt2=reshape(wypxt,N,1);

% Laplacian Terms
for j=1:nx
  wtxx(j,:)=-wt(j,:).*(kx.^2)';
end
for j=1:ny
  wtyy(:,j)=-wt(:,j).*(ky.^2);
end
wtxx2=reshape(wtxx,N,1);
wtyy2=reshape(wtyy,N,1);

rhs=(nu*(wtxx2+wtyy2)-wypxt2+wxpyt2);
```

The code will quickly and efficiently solve the advection-diffusion equations in two dimensions. Figure 14 demonstrates the evolution of the initial stretched Gaussian vortex over $t \in [0, 8]$.

4.4 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

$$\frac{\partial u}{\partial t} = -\frac{\partial^4 u}{\partial x^4}\,. \tag{4.4.1}$$

In the Fourier domain, this becomes

$$\frac{d\widehat{u}}{dt} = -(ik)^4\widehat{u} = -k^4\widehat{u} \tag{4.4.2}$$

which has the solution

$$\widehat{u} = \widehat{u}_0 \exp(-k^4 t)\,. \tag{4.4.3}$$

However, a time-stepping scheme would obviously solve (4.4.2) 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

$$k^4_{\max} = (64)^4 = 16,777,216\,, \tag{4.4.4}$$

or roughly $k^4_{\max} = 1.7 \times 10^7$. For $n = 1024$ Fourier modes, this is

$$k^4_{\max} = 6.8 \times 10^{10}\,. \tag{4.4.5}$$

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

$$\frac{d\widehat{u}}{dt} = -(10^{10})(10^{-6}) = -10^4\,. \tag{4.4.6}$$

Such large numbers on the right hand side of the equations forces 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.

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

$$\frac{\partial u}{\partial t} = Lu + N(u) \tag{4.4.7}$$

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 non-constant 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

$$\frac{dy}{dt} + p(t)y = g(t)\,. \tag{4.4.8}$$

We can multiply by the integrating factor $\mu(t)$ so that

$$\mu\frac{dy}{dt} + \mu p(t)y = \mu g(t)\,. \tag{4.4.9}$$

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

$$\frac{d}{dt}(\mu y) = \mu g(t)\,, \tag{4.4.10}$$

provided $d\mu/dt = \mu p(t)$. The solution then becomes

$$y = \frac{1}{\mu}\left[\int \mu(t) g(t) dt + c\right] \quad \mu(t) = \exp\left(\int p(t) dt\right) \tag{4.4.11}$$

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 (4.4.7) results in the spectral system of differential equations

$$\frac{d\widehat{u}}{dt} = \alpha(k)\widehat{u} + \widehat{N(u)}\,. \tag{4.4.12}$$

This can be rewritten

$$\frac{d\widehat{u}}{dt} - \alpha(k)\widehat{u} = \widehat{N(u)}\,. \tag{4.4.13}$$

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

$$\frac{d\widehat{v}}{dt} = \exp(-\alpha(k)t)\widehat{N(u)} \tag{4.4.14a}$$

$$\widehat{u} = \widehat{v}\exp(\alpha(k)t)\,. \tag{4.4.14b}$$

Thus the linear, constant coefficient terms are solved for explicitly, and the numerical stiffness associated with the Lu term is effectively eliminated.

Example: Fisher-Kolmogorov Equation

As an example of the implementation of the the filtered pseudo-spectral scheme, we consider the Fisher-Kolmogorov equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + u^3 + cu\,. \tag{4.4.15}$$

Fourier transforming the equation yields

$$\frac{d\widehat{u}}{dt} = (ik)^2\widehat{u} + \widehat{u^3} + c\widehat{u} \tag{4.4.16}$$

which can be rewritten

$$\frac{d\widehat{u}}{dt} + (k^2 - c)\widehat{u} = \widehat{u^3}\,. \tag{4.4.17}$$

Thus $\alpha(k) = c - k^2$ and

$$\frac{d\widehat{v}}{dt} = \exp[-(c-k^2)t]\widehat{u^3} \tag{4.4.18a}$$

$$\widehat{u} = \widehat{v}\exp[(c-k^2)t]\,. \tag{4.4.18b}$$

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 couple of practical issues that should be considered when implementing this technique

- When solving the general equation

$$\frac{\partial u}{\partial t} = Lu + N(u) \tag{4.4.19}$$

  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 savings 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 MATLAB since it uses adaptive time-stepping.

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

$$\alpha u(L) + \beta \frac{du(L)}{dx} = \gamma \tag{4.4.20}$$

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.

4.5 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 non-periodic boundary conditions.

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 phenomena: strong oscillations and errors are accumulated at the jump locations. This will greatly effect the accuracy of the scheme. So although spectral accuracy and speed is retained away from discontinuities, the errors and the jumps will begin to propagate out to the rest of the computational domain.

To see this phenomena, Fig. 29a considers a function which is periodically extended as shown in Fig. 29b. The FFT approximation to this function is shown in Fig. 29c where the Gibb's oscillations are clearly seen at the jump locations. The oscillations clearly impact the usefulness of this periodic extension technique.

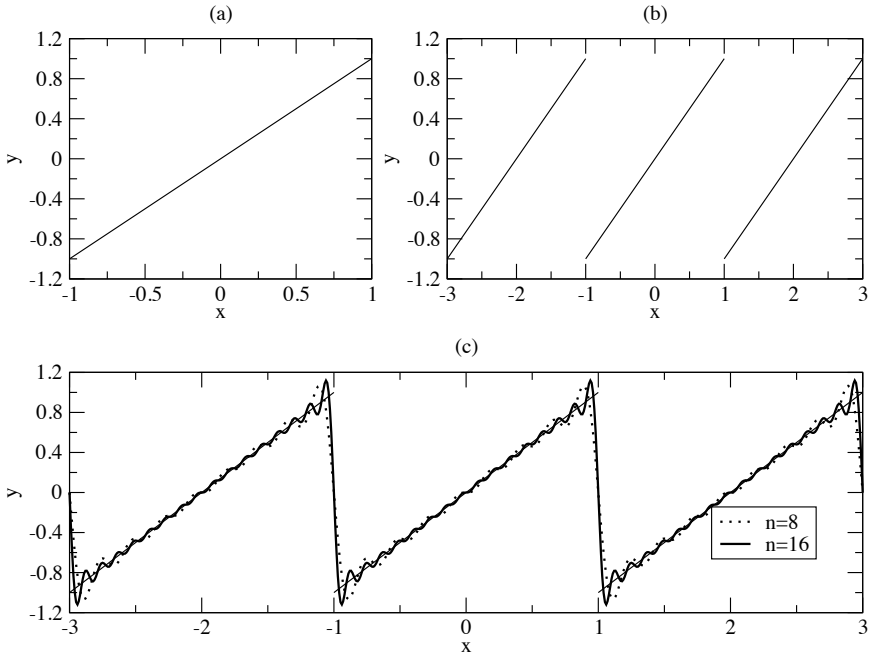

Figure 29: The function $y(x) = x$ for $x \in [-1, 1]$ (a) and its periodic extension (b). The FFT approximation is shown in (c) for $n = 8$ Fourier modes and $n = 16$ Fourier modes. Note the Gibb's oscillations.

Method 2: polynomial approximation with equi-spaced points

In moving away from an FFT basis expansion method, we can consider the most straight forward method available: polynomial approximation. Thus we simply discretize the given function $f(x)$ with $N + 1$ equally spaced points and fit an N^{th} degree polynomial through it. This amounts to letting

$$f(x) \approx a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^N \tag{4.5.1}$$

where the coefficients a_n are determined by an $(N+1) \times (N+1)$ system of equations. This method easily satisfies the prescribed non-periodic boundary conditions. Further, differentiation of such an approximation is trivial. In this case however, Runge phenomena (polynomial oscillations) generally occurs. This is because a polynomial of degree n generally has $N - 1$ combined maximums and minimums.

The Runge phenomena can easily be illustrated with the simple example function $f(x) = (1 + 16x^2)^{-1}$. Figure 30(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 oscil-

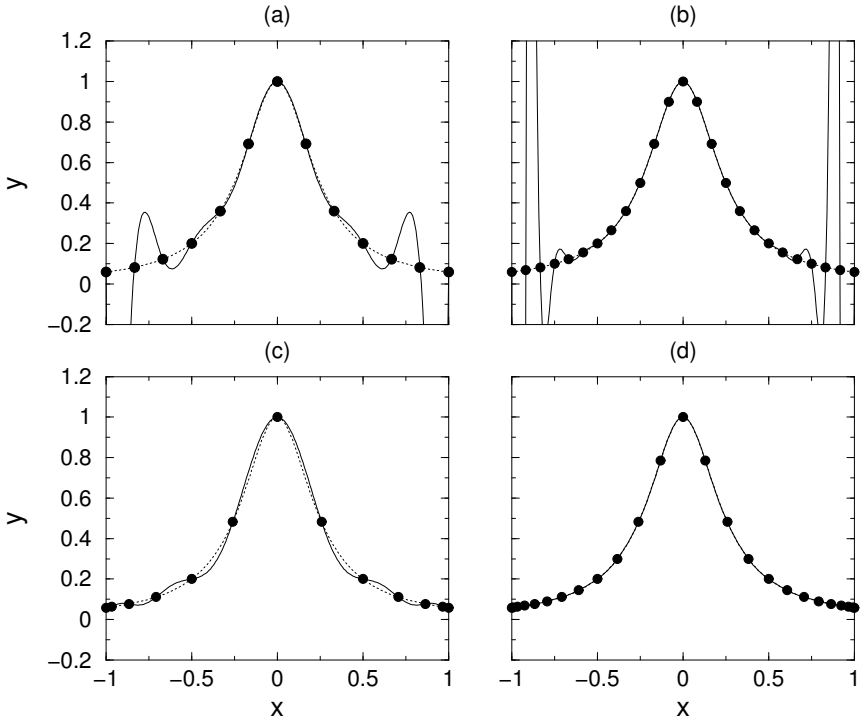

Figure 30: The function $y(x) = (1 + 16x^2)^{-1}$ for $x \in [-1, 1]$ (dotted line) with the bold points indicating the grid points for equi-spaced spaced points (a) $n = 12$ and (b) $n = 24$ and Chebychev clustered points (c) $n = 12$ and (d) $n = 24$. The solid lines are the polynomial approximations generated using the equi-spaced points (a)-(b) and clustered points (c)-(d) respectively.

lations render a simple polynomial approximation based upon equally spaced points useless.

Method 3: polynomial approximation with clustered points

There exists a modification to the straight forward polynomial approximation given above. Specifically, this modification constructs a polynomial approximation on a clustered grid as opposed to the equal spacing 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

$$x_n = \cos(n\pi/N) \tag{4.5.2}$$

where $n = 0, 1, 2, ..., N$. This helps to greatly reduce the effects of the Runge phenomena.

The clustered grid approximation results in a polynomial approximation shown in Fig. 30(c)-(d). There are reasons for this great improvement using a clustered grid [10]. However, we will not discuss them since they are beyond the scope of this course. This clustered grid suggests an accurate and easy way to represent a function which does not have periodic boundaries.

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

$$\mathbf{w} = \mathbf{D}_N \mathbf{v} \tag{4.5.3}$$

where $\mathbf{D}_N$ represents the action of differentiation. By using interpolation of the Lagrange form [5], 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

$$(D_N)_{00} = \frac{2N^2+1}{6} \tag{4.5.4a}$$

$$(D_N)_{NN} = -\frac{2N^2+1}{6} \tag{4.5.4b}$$

$$(D_N)_{jj} = -\frac{x_j}{2(1-x_j^2)} \quad j = 1, 2, ..., N-1 \tag{4.5.4c}$$

$$(D_N)_{ij} = \frac{c_i(-1)^{i+j}}{c_j(x_i - x_j)} \quad i, j = 0, 1, ..., N \ (i \neq j) \tag{4.5.4d}$$

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 $\mathbf{D}_N$

$$\mathbf{D}_N = \left(\begin{array}{c|ccc|c} \frac{2N^2+1}{6} & & \frac{2(-1)^j}{1-x_j} & & \frac{(-1)^N}{2} \\ \hline & \ddots & & \frac{(-1)^{i+j}}{x_i-x_j} & \\ \frac{-(-1)^i}{2(1-x_i)} & & \frac{-x_j}{2(1-x_j^2)} & & \frac{(-1)^{N+i}}{2(1+x_j)} \\ & \frac{(-1)^{i+j}}{x_i-x_j} & & \ddots & \\ \hline \frac{-(-1)^N}{2} & & \frac{-2(-1)^{N+j}}{1+x_j} & & -\frac{2N^2+1}{6} \end{array} \right) \tag{4.5.5}$$

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
- $\mathbf{D}_N^m$ - m^{th} derivative

Boundaries

The construction of the differentiation matrix $\mathbf{D}_N$ does not explicitly include the boundary conditions. Thus the general differentiation given by (4.5.3) must be modified to include the given boundary conditions. Consider, for example, the simple boundary conditions

$$v(-1) = v(1) = 0\,. \tag{4.5.6}$$

The given differentiation matrix is then written as

$$\begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{N-1} \\ w_N \end{pmatrix} = \left(\begin{array}{c|c|c} & \text{top row} & \\ \hline \text{first column} & (D_N) & \text{last column} \\ \hline & \text{bottom row} & \end{array}\right) \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{N-1} \\ v_N \end{pmatrix}. \tag{4.5.7}$$

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

$$\tilde{\mathbf{w}} = \tilde{\mathbf{D}}_N \tilde{\mathbf{v}} \tag{4.5.8}$$

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 MATLAB command:

```
tildeD=D(2:N,2:N)
```

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.

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 (4.5.2), 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

$$\frac{\partial T_n(\pm 1)}{\partial x} = 0 \tag{4.5.9}$$

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.

4.6 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

$$x_j = \cos(j\pi/N) \quad j = 0, 1, 2, ..., N\,. \tag{4.6.1}$$

This clustering will give higher resolution near the boundaries than the interior of the computational domain. The Chebychev differentiation matrix

$$\mathbf{D}_N \tag{4.6.2}$$

can then be constructed. Recall that the elements of this matrix are given by

$$(D_N)_{00} = \frac{2N^2+1}{6} \tag{4.6.3a}$$

$$(D_N)_{NN} = -\frac{2N^2+1}{6} \tag{4.6.3b}$$

$$(D_N)_{jj} = -\frac{x_j}{2(1-x_j^2)} \quad j = 1, 2, ..., N-1 \tag{4.6.3c}$$

$$(D_N)_{ij} = \frac{c_i(-1)^{i+j}}{c_j(x_i - x_j)} \quad i, j = 0, 1, ..., N \ (i \neq j) \tag{4.6.3d}$$

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 MATLAB 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 $x \in [-1, 1]$.

```
% cheb.m - compute the matrix D_N
function [D,x]=cheb(N)
```

```
if N==0, D=0; x=1; return; end
x=cos(pi*(0:N)/N)';
c=[2; ones(N-1,1); 2].*(-1).^(0:N)';
X=repmat(x,1,N+1);
dX=X-X';
D=(c*(1./c)')./(dX+eye(N+1));  % off diagonals
D=D-diag(sum(D'));              % diagonals
```

To test the differentiation, we consider two functions for which we know the exact values for the derivative and second derivative. Consider then

```
x=[-1:0.01:1]
u=exp(x).*sin(5*x);
v=sech(x);
```

The first and second derivative of each of these functions is given by

```
ux=exp(x).*sin(5*x) + 5*exp(x).*cos(5*x);
uxx=-4*exp(x).*sin(5*x) + 10*exp(x).*cos(5*x);

vx=-sech(x).*tanh(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 derivative. All that is required is the number of discretation points N and the routine *cheb.m*.

```
N=20
[D,x2]=cheb(N) % x2-clustered grid, D-differentiation matrix
D2=D^2;      % D2 - second derivative matrix
```

Given the differentiation matrix $\mathbf{D}_N$ and clustered grid, the given function and its derivatives can be constructed numerically.

```
u2=exp(x2).*sin(5*x2);
v2=sech(x2);

u2x=D*u2;   % first derivatives
v2x=D*v2;

u2xx=D2*u2;   % second derivatives
v2xx=D2*v2;
```

A comparison between the exact values for the differentiated functions and their approximations is shown in Fig. 31. As is expected the agreement is best for higher values of N. The MATLAB code for generating these graphical figures is given by

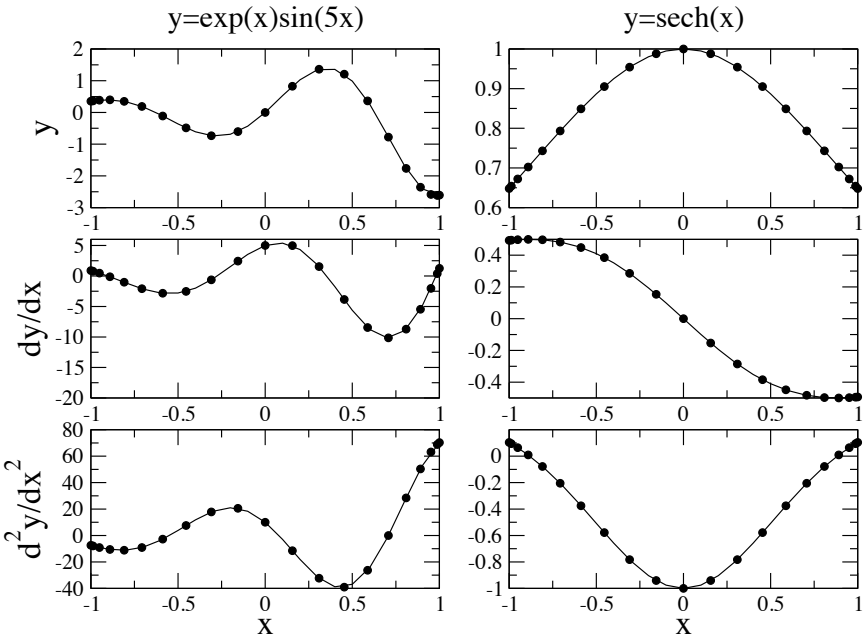

Figure 31: The function $y(x) = \exp(x)\sin 5x$ (left) and $y(x) = \text{sech}x$ (right) for $x \in [-1, 1]$ and their first and second derivatives. The dots indicate the numerical values while the solid line is the exact solution. For these calculations, $N = 20$ in the differentiation matrix $\mathbf{D}_N$.

```
figure 1; plot(x,u,'r',x2,u2,'.',x,v,'g',x2,v2,'.');
figure 2; plot(x,ux,'r',x2,u2x,'.',x,vx,'g',x2,v2x,'.'):
figure 3; plot(x,uxx,'r',x2,u2xx,'.',x,vxx,'g',x2,v2xx,'.');
```

Differentiation matrix in 2-D

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* command is MATLAB makes this calculation trivial. In particular, the 2D Laplacian operator L given by

$$Lu = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{4.6.4}$$

can be constructed with

```
L = kron(I,D2) + kron(D2,I)
```

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.

Solving PDEs with the Chebychev differentiation matrix

To illustrate the use of the Chebychev differentiation matrix, the two dimensional heat equation is considered:

$$\frac{\partial u}{\partial t} = \nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{4.6.5}$$

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.

```
clear all; close all;
N=30;
[D,x]=cheb(N); D2=D^2; y=x;
```

The Dirichlet boundary conditions are then imposed by modifying the first and last rows of the differentiation matrix. Specifically, these rows are set to zero.

```
D22=D2;
D22(N+1,:)=zeros(1,N+1);
D22(1,:)=zeros(1,N+1);
```

The two dimensional Laplacian is then constructed from the one dimensional differentiation matrix by using the *kron* command

```
I=eye(length(D22));
L=kron(I,D22)+kron(D22,I);  % 2D Laplacian
```

The initial conditions for this simulation will be a Gaussian centered at the origin.

```
[X,Y]=meshgrid(x,y);
U=exp(-(X.^2+Y.^2)/0.1);
surfl(x,y,U), shading interp, colormap(gray), drawnow
```

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*.

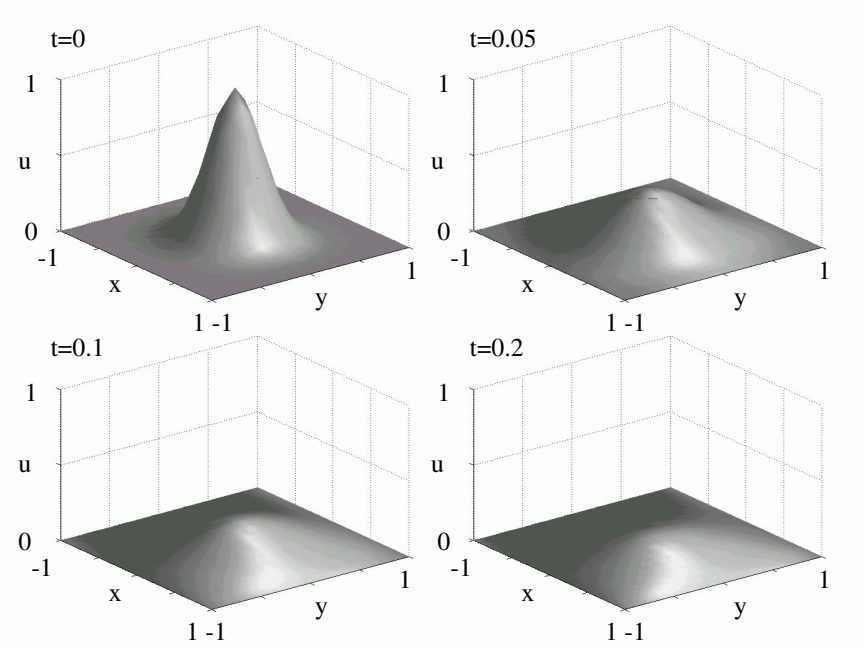

Figure 32: Evolution of an initial two dimensional Gaussian governed by the heat equation on the domain $x, y \in [-1, 1]$ with Dirichlet boundary conditions $u = 0$. The Chebychev differentiation matrix is used to calculate the Laplacian with $N = 30$.

```
u=reshape(U,(N+1)^2,1);
for j=1:4
  [t,ysol]=ode23('heatrhs2D',[0 0.05],u,[],L);
  u=ysol(end,:);
  U=reshape(u,N+1,N+1);
  surfl(x,y,U), shading interp, colormap(gray), drawnow
end
```

This code will advance the solution $\Delta t = 0.05$ in the future and plot the solution. Figure 32 depicts the two dimensional diffusive behavior given the Dirichlet boundary conditions using the Chebychev differentiation matrix.

4.7 Operator splitting techniques

With either the finite difference or spectral method, 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 Neu-

mann 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

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2}\,. \tag{4.7.1}$$

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.

Operator Splitting

The key idea behind operator splitting is to decouple the various physical effects of the problem from each other. Over very small time-steps, for instance, one can imagine that diffusion would essentially act independently of the wave propagation and vice-versa in (4.7.1). Just as in (4.7.1), the advection-diffusion can also be thought of as decoupling. So we can consider

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega \tag{4.7.2}$$

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:

$$\Delta t: \quad \frac{\partial \omega}{\partial t} + [\psi, \omega] = 0 \quad \text{advection only} \tag{4.7.3a}$$

$$\Delta t: \quad \frac{\partial \omega}{\partial t} = \nu \nabla^2 \omega \quad \text{diffusion only}\,. \tag{4.7.3b}$$

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.

Additional Advantages of Splitting

There can be additional advantages to the splitting scheme. To see this, we consider the nonlinear Schrödinger equation

$$i\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial^2 u}{\partial x^2} + |u|^2 u = 0 \tag{4.7.4}$$

where we split the operations so that

$$\textbf{I.}\ \ \Delta t: \quad i\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial^2 u}{\partial x^2} = 0 \tag{4.7.5a}$$

$$\textbf{II.}\ \ \Delta t: \quad i\frac{\partial u}{\partial t} + |u|^2 u = 0\,. \tag{4.7.5b}$$

Thus the solution will be decomposed into a linear and nonlinear part. We begin by solving the linear part **I.** Fourier transforming yields

$$i\frac{d\widehat{u}}{dt} - \frac{k^2}{2}\widehat{u} = 0 \tag{4.7.6}$$

which has the solution

$$\widehat{u} = \widehat{u}_0 \exp\left(-i\frac{k^2}{2}t\right)\,. \tag{4.7.7}$$

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

$$u = u_0 \exp(i|u_0|^2 t) \tag{4.7.8}$$

where u_0 is the initical 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.

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

$$\frac{\partial u}{\partial t} = Lu + N(u) \tag{4.7.9}$$

involves the splitting

$$\textbf{I.}\ \ \Delta t: \quad \frac{\partial u}{\partial t} + Lu = 0 \tag{4.7.10a}$$

$$\textbf{II.}\ \ \Delta t: \quad \frac{\partial u}{\partial t} + N(u) = 0\,. \tag{4.7.10b}$$

To drop the error down by another $O(\Delta t)$, we use what is called a Strang splitting technique [11] which is based upon the Trotter product formula. This essentially involves symmetrizing the splitting algorithm so that

$$\textbf{I.}\ \ \frac{\Delta t}{2}: \quad \frac{\partial u}{\partial t} + Lu = 0 \tag{4.7.11a}$$

$$\textbf{II.}\ \ \Delta t: \quad \frac{\partial u}{\partial t} + N(u) = 0 \tag{4.7.11b}$$

$$\textbf{III.}\ \ \frac{\Delta t}{2}: \quad \frac{\partial u}{\partial t} + Lu = 0 \tag{4.7.11c}$$

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.

5 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.

5.1 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

1. Discretization of the computational domain
2. Selection of the interpolating functions
3. Derivation of characteristic matrices and vectors
4. Assembly of characteristic matrices and vectors
5. 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.

Domain Discretization

Finite element domain discretization usually involves the use of a commercial package. Two commonly used packages are the *MATLAB PDE Toolbox* and *FEMLAB* (which is built on the MATLAB platform). Writing your own code to generate an unstructured computational grid is a difficult task and well beyond the scope of this course. 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 33 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.

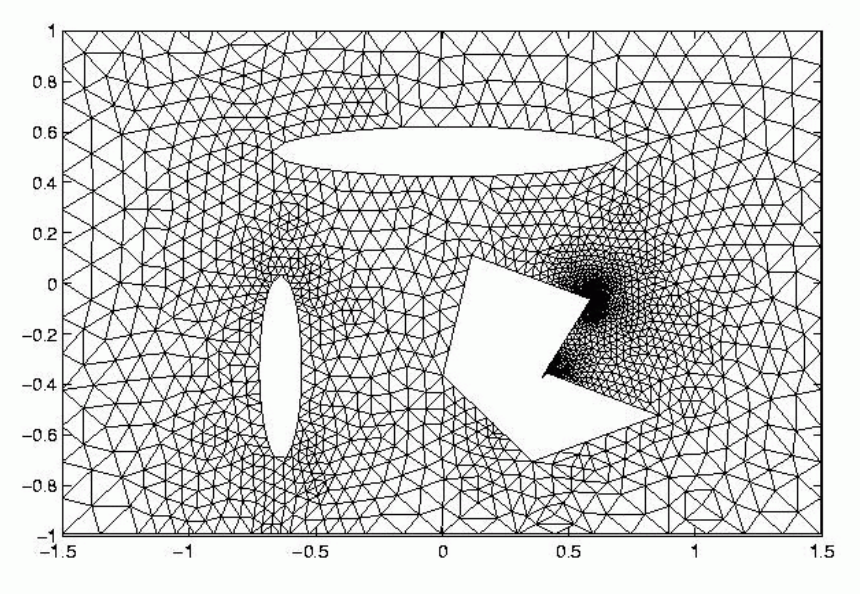

Figure 33: Finite element triangular discretization provided by the MATLAB PDE Toolbox. Note the increased resolution near corners and edges.

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 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. 34 and 35 for one-dimensional and two-dimensional finite elements.

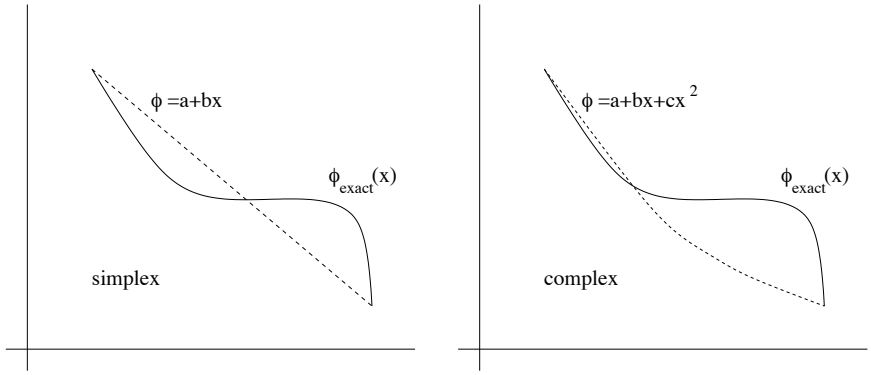

Figure 34: Finite element discretization for simplex and complex finite elements. Essentially the finite elements are approximating polynomials.

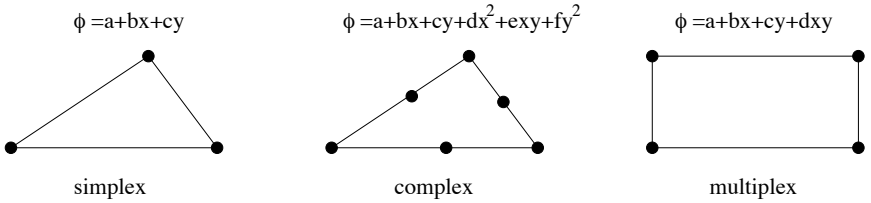

Figure 35: Finite element discretization for simplex, complex, and multiplex finite elements in two-dimensions. The finite elements are approximating surfaces.

1D Simplex

To consider the finite element implementation, we begin by considering the one-dimensional problem. Figure 36 shows the linear polynomial used to approximate the solution between points x_i and x_j. The approximation of the function is thus

$$\phi(x) = a + bx = (1 \;\; x) \begin{pmatrix} a \\ b \end{pmatrix} . \tag{5.1.1}$$

The coefficients a and b are determined by enforcing that the function goes through the end points. This gives

$$\phi_i = a + bx_i \tag{5.1.2a}$$

$$\phi_j = a + bx_j \tag{5.1.2b}$$

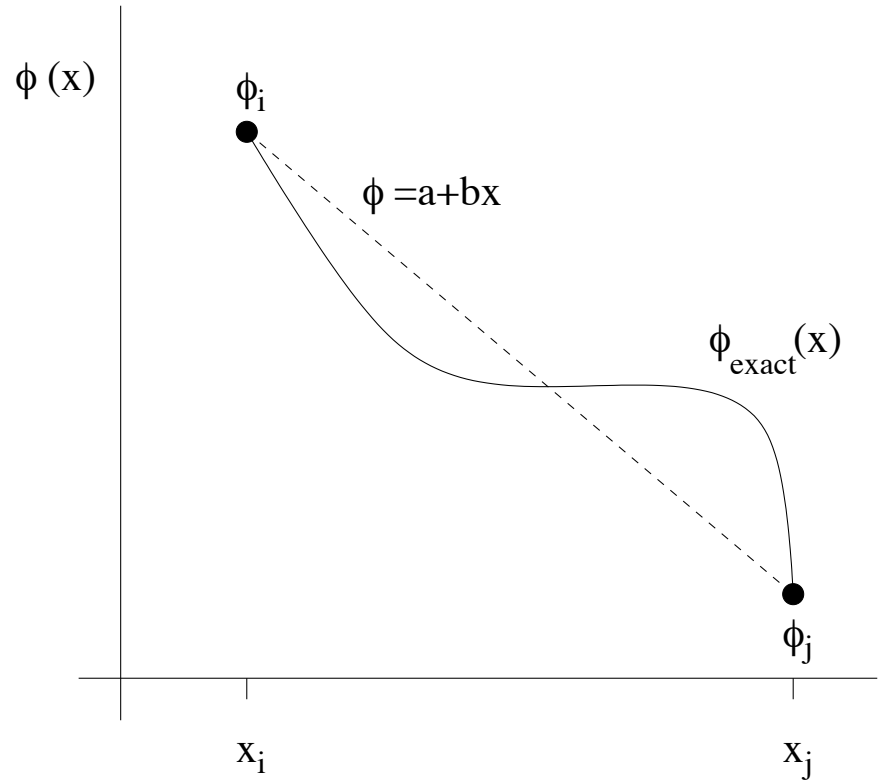

Figure 36: One dimensional approximation using finite elements.

which is a 2×2 system for the unknown coefficients. In matrix form this can be written

$$\phi = \mathbf{A}\mathbf{a} \;\rightarrow\; \phi = \begin{pmatrix} \phi_i \\ \phi_j \end{pmatrix}, \; \mathbf{A} = \begin{pmatrix} 1 & x_i \\ 1 & x_j \end{pmatrix}, \; \mathbf{a} = \begin{pmatrix} a \\ b \end{pmatrix}. \tag{5.1.3}$$

Solving for $\mathbf{a}$ gives

$$\mathbf{a} = \mathbf{A}^{-1}\phi = \frac{1}{x_j - x_i}\begin{pmatrix} x_j & -x_i \\ -1 & 1 \end{pmatrix}\begin{pmatrix} \phi_i \\ \phi_j \end{pmatrix} = \frac{1}{l}\begin{pmatrix} x_j & -x_i \\ -1 & 1 \end{pmatrix}\begin{pmatrix} \phi_i \\ \phi_j \end{pmatrix} \tag{5.1.4}$$

where $l = x_j - x_i$. Recalling that $\phi = (1 \;\; x)\mathbf{a}$ then gives

$$\phi = \frac{1}{l}(1 \;\; x)\begin{pmatrix} x_j & -x_i \\ -1 & 1 \end{pmatrix}\begin{pmatrix} \phi_i \\ \phi_j \end{pmatrix}. \tag{5.1.5}$$

Multiplying this expression out yields the approximate solution

$$\phi(x) = N_i(x)\phi_i + N_j(x)\phi_j \tag{5.1.6}$$

where $N_i(x)$ and $N_j(x)$ are the Lagrange polynomial coefficients [5] (shape functions)

$$N_i(x) = \frac{1}{l}(x_j - x) \tag{5.1.7a}$$

$$N_j(x) = \frac{1}{l}(x - x_i)\,. \tag{5.1.7b}$$

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.

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 Figure 37)

$$\phi(x) = a + bx + cy\,. \tag{5.1.8}$$

The coefficients a, b, and c are determined by enforcing that the function goes through the end points. This gives

$$\phi_i = a + bx_i + cy_i \tag{5.1.9a}$$

$$\phi_j = a + bx_j + cy_j \tag{5.1.9b}$$

$$\phi_k = a + bx_k + cy_k \tag{5.1.9c}$$

which is now a 3×3 system for the unknown coefficients. In matrix form this can be written

$$\phi = \mathbf{Aa} \;\rightarrow\; \phi = \begin{pmatrix} \phi_i \\ \phi_j \\ \phi_k \end{pmatrix},\; \mathbf{A} = \begin{pmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_k & y_k \end{pmatrix},\; \mathbf{a} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}. \tag{5.1.10}$$

The geometry of this problem is reflected in Fig. 37 where each of the points of the discretization triangle is illustrated.

Solving for $\mathbf{a}$ gives

$$\mathbf{a} = \mathbf{A}^{-1}\phi = \frac{1}{2S}\begin{pmatrix} x_jy_k - x_ky_j & x_ky_i - x_iy_k & x_iy_j - x_jy_i \\ y_j - y_k & y_k - y_i & y_i - y_j \\ x_k - x_j & x_i - x_k & x_j - x_i \end{pmatrix}\begin{pmatrix} \phi_i \\ \phi_j \\ \phi_k \end{pmatrix} \tag{5.1.11}$$

where S is the area of the triangle projected onto the $x - y$ plane (see Fig. 37. Recalling that $\phi = \mathbf{Aa}$ gives

$$\phi(x) = N_i(x,y)\phi_i + N_j(x,y)\phi_j + N_k(x,y)\phi_k \tag{5.1.12}$$

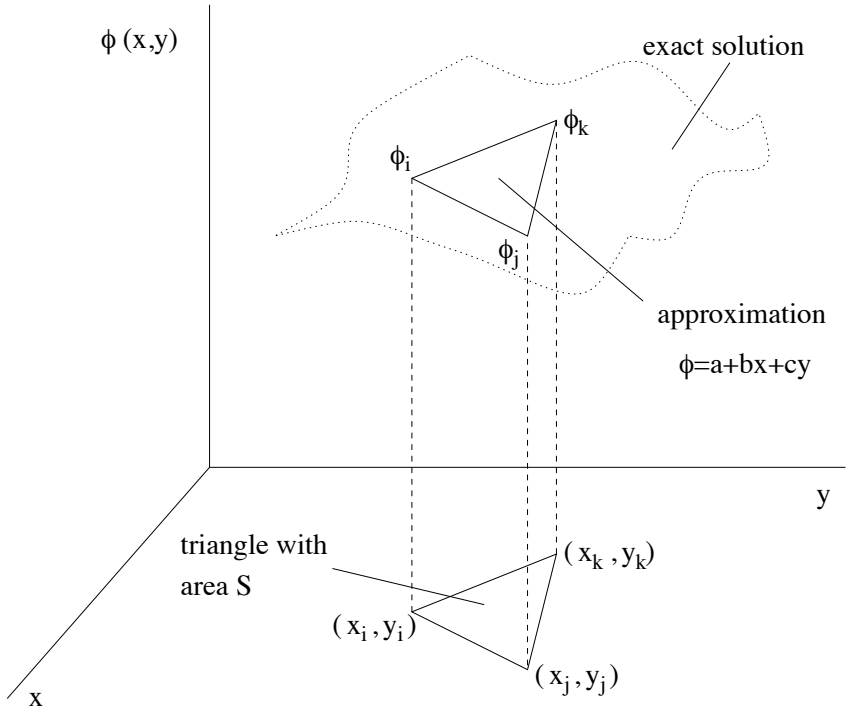

Figure 37: Two dimensional triangular approximation of the solution using the finite element method.

where $N_i(x,y)$, $N_j(x,y)$, $N_k(x,y)$ are the Lagrange polynomial coefficients [5] (shape functions)

$$N_i(x,y) = \frac{1}{2S}[(x_j y_k - x_k y_j) + (y_j - y_k)x + (x_k - x_j)y] \quad (5.1.13a)$$

$$N_j(x,y) = \frac{1}{2S}[(x_k y_i - x_i y_k) + (y_k - y_i)x + (x_i - x_k)y] \quad (5.1.13b)$$

$$N_k(x,y) = \frac{1}{2S}[(x_i y_j - x_j y_i) + (y_i - y_j)x + (x_j - x_i)y]\,. \quad (5.1.13c)$$

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. This completes the generic construction of the surface which approximates the solution in two dimensions.

5.2 Discretizing with finite elements and boundaries

In the previous lecture, 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

$$\frac{\partial}{\partial x}\left(p(x,y)\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right)+r(x,y)u=f(x,y) \tag{5.2.1}$$

where over part of the boundary

$$u(x,y)=g(x,y) \quad \text{on } S_1 \tag{5.2.2}$$

and

$$p(x,y)\frac{\partial u}{\partial x}\cos\theta_1+q(x,y)\frac{\partial u}{\partial y}\cos\theta_2+g_1(x,y)u=g_2(x,y) \quad \text{on } S_2\,. \tag{5.2.3}$$

The boundaries and domain are illustrated in Fig. 38. 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.

The Variational Principle

To formulate the problem correctly for the finite element method, the governing equations and its 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 [12], 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

$$I(\phi)=\int\!\!\int\!\!\int_V F\left(\phi,\frac{\partial\phi}{\partial x},\frac{\partial\phi}{\partial y},\frac{\partial\phi}{\partial z}\right)dV+\int\!\!\int_S g\left(\phi,\frac{\partial\phi}{\partial x},\frac{\partial\phi}{\partial y},\frac{\partial\phi}{\partial z}\right)dS\,, \tag{5.2.4}$$

where a three dimensional problem is being considered. The derivation of the functions F, which captures the governing equation, and g, which captures

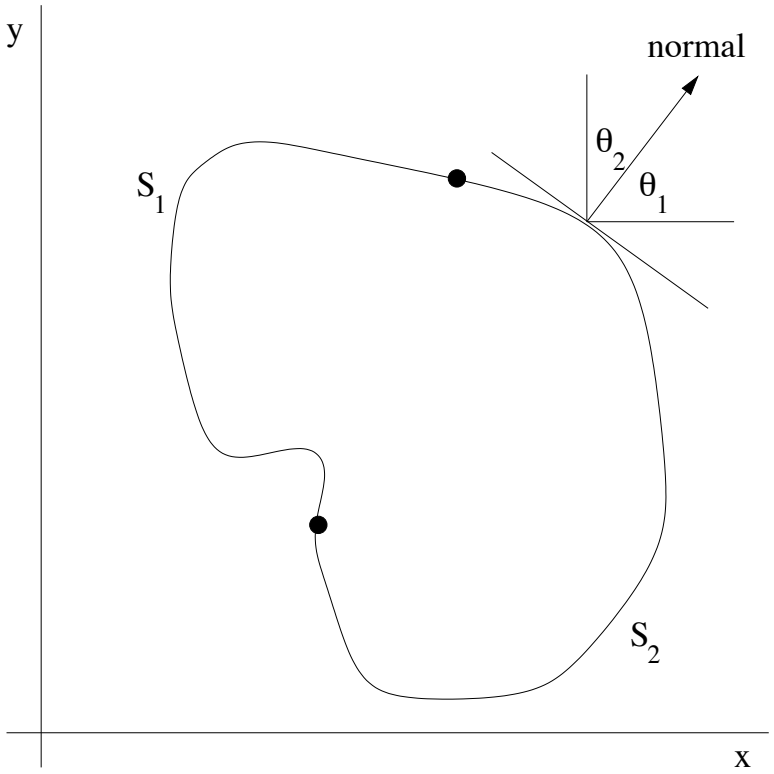

Figure 38: Computational domain of interest which has the two domain regions S_1 and S_2.

the boundary conditions, follow the Euler-Lagrange equations for minimizing variations:

$$\frac{\delta F}{\delta \phi} = \frac{\partial}{\partial x}\left(\frac{\partial F}{\partial(\phi_x)}\right) + \frac{\partial}{\partial y}\left(\frac{\partial F}{\partial(\phi_y)}\right) + \frac{\partial}{\partial z}\left(\frac{\partial F}{\partial(\phi_z)}\right) - \frac{\partial F}{\partial \phi} = 0\,. \qquad (5.2.5)$$

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 (5.2.1) with boundary conditions given by those of S_2 (5.2.3), we find the functional $I(u)$ to be given by

$$I(u) = \frac{1}{2}\iint_D dxdy\left[p(x,y)\left(\frac{\partial u}{\partial x}\right)^2 + q(x,y)\left(\frac{\partial u}{\partial y}\right)^2 - r(x,y)u^2 + 2f(x,y)u\right] + \int_{S_2} dS\left[-g_2(x,y)u + \frac{1}{2}g_1(x,y)u^2\right]. \qquad (5.2.6)$$

Thus the interior domain over which we integrate D has the integrand

$$I_D = \frac{1}{2}\left[p(x,y)\left(\frac{\partial u}{\partial x}\right)^2 + q(x,y)\left(\frac{\partial u}{\partial y}\right)^2 - r(x,y)u^2 + 2f(x,y)u\right]. \qquad (5.2.7)$$

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 (5.2.5) and find

$$\begin{aligned}\frac{\delta I_D}{\delta u} &= \frac{\partial}{\partial x}\left(\frac{\partial I_D}{\partial(u_x)}\right) + \frac{\partial}{\partial y}\left(\frac{\partial I_D}{\partial(u_y)}\right) - \frac{\partial I_D}{\partial u} = 0 \\ &= \frac{\partial}{\partial x}\left(\frac{1}{2}p(x,y)\cdot 2\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{1}{2}q(x,y)\cdot 2\frac{\partial u}{\partial y}\right) - \left(-\frac{1}{2}r(x,y)\cdot 2u + f(x,y)\right) \\ &= \frac{\partial}{\partial x}\left(p(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + r(x,y)u - f(x,y) = 0\,. \qquad (5.2.8)\end{aligned}$$

Upon rearranging, this gives back the governing equation (5.2.1)

$$\frac{\partial}{\partial x}\left(p(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + r(x,y)u = f(x,y)\,. \qquad (5.2.9)$$

A similar procedure can be followed on the boundary terms to derive the appropriate boundary conditions (5.2.2) or (5.2.3). Once the integral formulation has been achieved, the finite element method can be implemented with the following key ideas:

1. 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 (5.2.1) would have no contribution from a finite element of the form $\phi = a_1 + a_2 x + a_3 y$. However, in the integral formulation, the second derivative terms are proportional to $(\partial u/\partial x)^2 + (\partial u/\partial y)^2$ which gives $\phi = a_2^2 + a_3^2$.

2. A solution is sought which takes the form

$$u(x,y) = \sum_{i=1}^{m} \gamma_i \phi_i(x,y) \qquad (5.2.10)$$

   where $\phi_i(x,y)$ are the linearly independent piecewise-linear polynomials and $\gamma_1, \gamma_2, ..., \gamma_m$ are constants.

   - $\gamma_{n+1}, \gamma_{n+2}, ..., \gamma_m$ ensure that the boundary conditions are satisfied, i.e. those elements which touch the boundary must meet certain restrictions.
   - $\gamma_1, \gamma_2, ..., \gamma_n$ ensure that the integrand $I(u)$ in the interior of the computational domain is minimized, i.e. $\partial I/\partial\gamma_i = 0$ for $i = 1, 2, 3, ..., n$.

Solution Method

We begin with the governing equation in integral form (5.2.6) and assume an expansion of the form (5.2.10). This gives

$$\begin{aligned} I(u) &= I\left(\sum_{i=1}^{m}\gamma_i\phi_i(x,y)\right) \\ &= \frac{1}{2}\iint_D dxdy\left[p(x,y)\left(\sum_{i=1}^{m}\gamma_i\frac{\partial\phi_i}{\partial x}\right)^2+q(x,y)\left(\sum_{i=1}^{m}\gamma_i\frac{\partial\phi_i}{\partial y}\right)^2\right. \\ &\qquad \left.-r(x,y)\left(\sum_{i=1}^{m}\gamma_i\phi_i(x,y)\right)^2+2f(x,y)\left(\sum_{i=1}^{m}\gamma_i\phi_i(x,y)\right)\right] \qquad (5.2.11) \\ &\quad +\int_{S_2} dS\left[-g_2(x,y)\sum_{i=1}^{m}\gamma_i\phi_i(x,y)+\frac{1}{2}g_1(x,y)\left(\sum_{i=1}^{m}\gamma_i\phi_i(x,y)\right)^2\right]. \end{aligned}$$

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 $i = 1, 2, 3, ..., n$. This results in the rather complicated expression

$$\begin{aligned} \frac{\partial I}{\partial\gamma_j} &= \iint_D dxdy\left[p(x,y)\sum_{i=1}^{m}\gamma_i\frac{\partial\phi_i}{\partial x}\frac{\partial\phi_j}{\partial x}+q(x,y)\sum_{i=1}^{m}\gamma_i\frac{\partial\phi_i}{\partial y}\frac{\partial\phi_j}{\partial y}\right. \\ &\qquad \left.-r(x,y)\sum_{i=1}^{m}\gamma_i\phi_i\phi_j+f(x,y)\phi_j\right] \qquad (5.2.12) \\ &\quad +\int_{S_2} dS\left[-g_2(x,y)\phi_j+g_1(x,y)\sum_{i=1}^{m}\gamma_i\phi_i\phi_j\right]=0\,. \end{aligned}$$

Term by term, this results in an expression for the γ_i given by

$$\begin{aligned} &\sum_{i=1}^{m}\left\{\iint_D dxdy\left[p\frac{\partial\phi_i}{\partial x}\frac{\partial\phi_j}{\partial x}+q\frac{\partial\phi_i}{\partial y}\frac{\partial\phi_j}{\partial y}-r\phi_i\phi_j\right]+\int_{S_2} dSg_1\phi_i\phi_j\right\}\gamma_i \\ &\qquad +\iint_D dxdyf\phi_j-\int_{S_2} dSg_2(x,y)\phi_j=0\,. \qquad (5.2.13) \end{aligned}$$

But this expression is simply a matrix solve $\mathbf{Ax} = \mathbf{b}$ for the unknowns γ_i. In particular, we have

$$\mathbf{x}=\begin{pmatrix}\gamma_1\\ \gamma_2\\ \vdots\\ \gamma_n\end{pmatrix} \quad \mathbf{b}=\begin{pmatrix}\beta_1\\ \beta_2\\ \vdots\\ \beta_n\end{pmatrix} \quad \mathbf{A}=(\alpha_{ij}) \qquad (5.2.14)$$

where

$$\beta_i = -\int\int_D dxdy f\phi_i + \int_{S_2} dS g_2\phi_i - \sum_{k=n+1}^{m} \alpha_{ik}\gamma_k \tag{5.2.15a}$$

$$\alpha_{ij} = \int\int_D dxdy \left[p\frac{\partial\phi_i}{\partial x}\frac{\partial\phi_j}{\partial x} + q\frac{\partial\phi_i}{\partial y}\frac{\partial\phi_j}{\partial y} - r\phi_i\phi_j \right] + \int_{S_2} dS g_1\phi_i\phi_j. \tag{5.2.15b}$$

Recall that each element ϕ_i is given by the simplex approximation

$$\phi_i = \sum_{i=1}^{3} N_j^{(i)}(x,y)\phi_j^{(i)} = \sum_{i=1}^{3} \left(a_j^{(i)} + b_j^{(i)}x + c_j^{(i)}y \right) \phi_j^{(i)} . \tag{5.2.16}$$

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.

1. Discretize the computational domain into triangles. $T_1, T_2, ..., T_k$ are the interior triangles and $T_{k+1}, T_{k+2}, ..., T_m$ are triangles that have at least one edge which touches the boundary.

2. For $l = k+1, k+2, ..., m$, determine the values of the vertices on the triangles which touch the boundary.

3. Generate the shape functions

$$N_j^{(i)} = a_j^{(i)} + b_j^{(i)}x + c_j^{(i)}y \tag{5.2.17}$$

   where $i = 1, 2, 3, ..., m$ and $j = 1, 2, 3, ..., m$.

4. Compute the integrals for matrix elements α_{ij} and vector elements β_j in the interior and boundary.

5. Construct the matrix $\mathbf{A}$ and vector $\mathbf{b}$.

6. Solve $\mathbf{Ax} = \mathbf{b}$.

7. 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 (5.2.1), (5.2.2), and (5.2.3) are known, the solution procedure is straightforward.

A Application Problems

The purpose of this class: to be able to solve realistic and difficult problems which arise in a variety of physical settings. This appendix outlines a few physical problems which give rise to phenomena of great general interest. The techniques developed in the lectures on finite difference, spectral, and finite element methods provide all the necessary tools for solving these problems. Thus each problem isn't simply a mathematical problem, but rather a way to describe phenomena which may be observed in various physically interesting systems.

A.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 the shallow-water modeling is illustrated in Fig. 39. 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

$$\delta = \frac{D}{L} \ll 1 \,. \tag{A.1.1}$$

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

$$\mathbf{v} = \begin{pmatrix} u \\ v \\ w \end{pmatrix} \tag{A.1.2}$$

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. $\mathbf{\Omega} = (\omega_\mathbf{x} \ \omega_\mathbf{y} \ \omega_\mathbf{z})^\mathbf{T}$. 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

$$\omega_z = \omega = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \,. \tag{A.1.3}$$

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

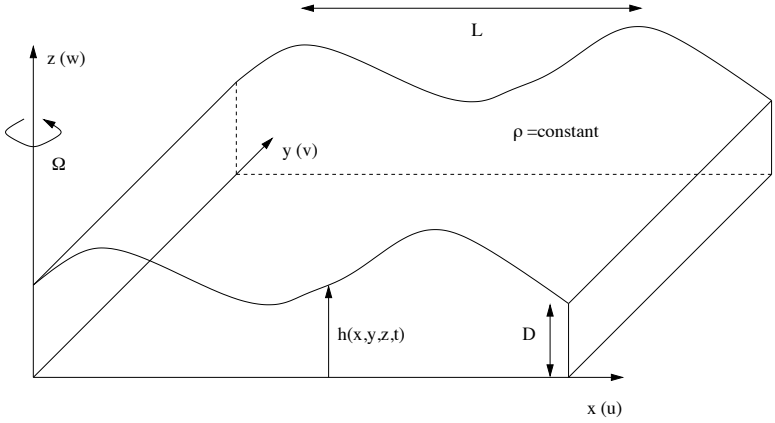

Figure 39: Physical setting of a shallow atmosphere or shallow water equations with constant density ρ and scale separation $D/L \ll 1$. The vorticity is measured by the parameter Ω.

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: 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. 39 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

$$\text{mass} = \int_{x_1}^{x_2} \int_{y_1}^{y_2} \rho(x,y) h(x,y,t) dx dy \tag{A.1.4}$$

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

$$\begin{aligned} &\frac{\partial}{\partial t} \int_{x_1}^{x_2} \int_{y_1}^{y_2} h(x,y,t) dx dy \\ &\quad + \int_{y_1}^{y_2} \left[u(x_2,y,t) h(x_2,y,t) - u(x_1,y,t) h(x_1,y,t) \right] dy \\ &\quad + \int_{x_1}^{x_2} \left[v(x,y_2,t) h(x,y_2,t) - v(x,y_1,t) h(x,y_1,t) \right] dx = 0, \end{aligned} \tag{A.1.5}$$

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 respectively. Note that from the

fundamental theorem of calculus, we can rewrite the second and third terms:

$$\int_{y_1}^{y_2}[u(x_2,y,t)h(x_2,y,t)-u(x_1,y,t)h(x_1,y,t)]\,dy=\int_{x_1}^{x_2}\int_{y_1}^{y_2}\frac{\partial}{\partial x}(uh)dxdy \quad \text{(A.1.6a)}$$

$$\int_{x_1}^{x_2}[v(x,y_2,t)h(x,y_2,t)-v(x,y_1,t)h(x,y_1,t)]\,dx=\int_{x_1}^{x_2}\int_{y_1}^{y_2}\frac{\partial}{\partial y}(vh)dxdy. \quad \text{(A.1.6b)}$$

Replacing these new expressions in (A.1.5) 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

$$\frac{\partial h}{\partial t}+\frac{\partial}{\partial x}(hu)+\frac{\partial}{\partial y}(hv)=0\,. \quad \text{(A.1.7)}$$

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

$$\frac{\partial h}{\partial t}+\frac{\partial}{\partial x}(hu)+\frac{\partial}{\partial y}(hv)=0 \quad \text{(A.1.8a)}$$

$$\frac{\partial}{\partial t}(hu)+\frac{\partial}{\partial x}\left(hu^2+\frac{1}{2}gh^2\right)+\frac{\partial}{\partial y}\left(huv\right)=fhv \quad \text{(A.1.8b)}$$

$$\frac{\partial}{\partial t}(hv)+\frac{\partial}{\partial y}\left(hv^2+\frac{1}{2}gh^2\right)+\frac{\partial}{\partial x}(huv)=-fhu\,. \quad \text{(A.1.8c)}$$

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 (A.1.8a) then reduces to

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0\,, \quad \text{(A.1.9)}$$

which is referred to as the *imcompressible 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

$$\frac{\partial u}{\partial t}+2u\frac{\partial u}{\partial x}+\frac{\partial}{\partial y}\left(uv\right)=fv \quad \text{(A.1.10a)}$$

$$\frac{\partial v}{\partial t}+2v\frac{\partial v}{\partial y}+\frac{\partial}{\partial x}(uv)=-fu\,. \quad \text{(A.1.10b)}$$

To simplify further, take the y derivative of the first equation and the x derivative of the second equation. The new equations are

$$\frac{\partial^2 u}{\partial t \partial y} + 2\frac{\partial u}{\partial y}\frac{\partial u}{\partial x} + 2u\frac{\partial^2 u}{\partial x \partial y} + \frac{\partial^2}{\partial y^2}(uv) = f\frac{\partial v}{\partial y} \tag{A.1.11a}$$

$$\frac{\partial^2 v}{\partial t \partial x} + 2\frac{\partial v}{\partial x}\frac{\partial v}{\partial y} + 2v\frac{\partial^2 v}{\partial x \partial y} + \frac{\partial^2}{\partial x^2}(uv) = -f\frac{\partial u}{\partial x}\,. \tag{A.1.11b}$$

Subtracting the first equation from the second gives the following reductions

$$\begin{aligned}
&\frac{\partial}{\partial t}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) - 2\frac{\partial u}{\partial y}\frac{\partial u}{\partial x} - 2u\frac{\partial^2 u}{\partial x \partial y} - \frac{\partial^2}{\partial y^2}(uv) \\
&\quad +2\frac{\partial v}{\partial x}\frac{\partial v}{\partial y} + 2v\frac{\partial^2 v}{\partial x \partial y} + \frac{\partial^2}{\partial x^2}(uv) = -f\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)
\end{aligned}$$

$$\begin{aligned}
&\frac{\partial}{\partial t}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) + 2\frac{\partial u}{\partial x}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) + 2\frac{\partial v}{\partial y}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) \\
&\quad +u\left(\frac{\partial^2 v}{\partial x^2} - \frac{\partial^2 v}{\partial y^2} - 2\frac{\partial^2 u}{\partial x \partial y}\right) + v\left(\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} + 2\frac{\partial^2 v}{\partial x \partial y}\right) = -f\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)
\end{aligned} \tag{A.1.12}$$

$$\begin{aligned}
&\frac{\partial}{\partial t}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) + u\frac{\partial}{\partial x}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) + v\frac{\partial}{\partial y}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right) - u\frac{\partial}{\partial y}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) \\
&\quad +v\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) + 2\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right)\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) = -f\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\,.
\end{aligned}$$

The final equation is reduced greatly by recalling the definition of the vorticity (A.1.3) and using the imcompressibility condition (A.1.9). The governing equations then reduce to

$$\frac{\partial \omega}{\partial t} + u\frac{\partial \omega}{\partial x} + v\frac{\partial \omega}{\partial y} = 0\,. \tag{A.1.13}$$

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:

$$u = -\frac{\partial \psi}{\partial y} \quad v = \frac{\partial \psi}{\partial x}\,. \tag{A.1.14}$$

Thus the streamfunction is specified up to an arbitrary constant. Note that the streamfunction automatically satisfies the imcompressibility condition since

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = -\frac{\partial^2 \psi}{\partial x \partial y} + \frac{\partial^2 \psi}{\partial x \partial y} = 0\,. \tag{A.1.15}$$

In terms of the vorticity, the streamfunction is related as follows:

$$\omega = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = \nabla^2 \psi \,. \tag{A.1.16}$$

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 (A.1.13). In the presence of frictional forces, modification of this governing equation occurs. Specifically, the motion in the shallow-water limit is given by

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega \tag{A.1.17a}$$

$$\nabla^2 \psi = \omega \tag{A.1.17b}$$

where

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} \tag{A.1.18}$$

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 equations classifications: parabolic, elliptic, and hyperbolic:

$$\text{parabolic:} \quad \frac{\partial \omega}{\partial t} = \nu \nabla^2 \omega \tag{A.1.19a}$$

$$\text{elliptic:} \quad \nabla^2 \psi = \omega \tag{A.1.19b}$$

$$\text{hyperbolic:} \quad \frac{\partial \omega}{\partial t} + [\psi, \omega] = 0 \,. \tag{A.1.19c}$$

Two things need to be solved for as a function of time:

$$\psi(x, y, t) \quad \text{streamfunction} \tag{A.1.20a}$$

$$\omega(x, y, t) \quad \text{vorticity} \,. \tag{A.1.20b}$$

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 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.

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 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.

A.2 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

$$\frac{dx}{dt} = ax \tag{A.2.1}$$

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

$$\frac{dy}{dt} = -cy \tag{A.2.2}$$

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:

$$\frac{dx}{dt} = ax - \alpha xy \tag{A.2.3a}$$

$$\frac{dy}{dt} = -cx + \alpha xy\,, \tag{A.2.3b}$$

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

$$\frac{\partial x}{\partial t} = a\left(x - \frac{x^2}{k}\right) - \alpha xy + D_1\nabla^2 x \tag{A.2.4a}$$

$$\frac{\partial y}{\partial t} = -cx + \alpha xy + D_2\nabla^2 y\,, \tag{A.2.4b}$$

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:

$$\frac{\partial u}{\partial t} = u(a-u)(1-u) - v + D\nabla^2 u \tag{A.2.5a}$$

$$\frac{\partial v}{\partial t} = bu - \gamma v\,, \tag{A.2.5b}$$

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

$$\frac{\partial u}{\partial t} = u(a-u)(1-u) - v \tag{A.2.6a}$$

$$\frac{\partial v}{\partial t} = bu - \gamma v\,. \tag{A.2.6b}$$

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

$$(u, v) = (0, 0) \tag{A.2.7a}$$

$$(u, v) = (u_\pm, (a-u_\pm)(1-u_\pm)u_\pm) \tag{A.2.7b}$$

where $u_\pm = [(a+1) \pm ((a+1)^2 - 4(a - b/\gamma))^{1/2}]/2$.

The stability of these three fixed points can be found by linearization [1]. In particular, consider the behavior near the steady-state solution $u = v = 0$. Thus let

$$u = 0 + \tilde{u} \tag{A.2.8a}$$

$$v = 0 + \tilde{v} \tag{A.2.8b}$$

where $\tilde{u}, \tilde{v} \ll 1$. Plugging in and discarding higher order terms gives the linearized equations

$$\frac{\partial \tilde{u}}{\partial t} = a\tilde{u} - \tilde{v} \tag{A.2.9a}$$

$$\frac{\partial \tilde{v}}{\partial t} = b\tilde{u} - \gamma\tilde{v}\,. \tag{A.2.9b}$$

This can be written as the linear system

$$\frac{d\mathbf{x}}{dt} = \begin{pmatrix} a & -1 \\ b & -\gamma \end{pmatrix} \mathbf{x}\,. \tag{A.2.10}$$

Assuming a solution of the form $\mathbf{x} = \mathbf{v}\exp(\lambda t)$ results in the eigenvalue problem

$$\begin{pmatrix} a-\lambda & -1 \\ b & -\gamma-\lambda \end{pmatrix} \mathbf{v} = 0 \tag{A.2.11}$$

which has the eigenvalues

$$\lambda_\pm = \frac{1}{2}\left[(a-\gamma) \pm \sqrt{(a-\gamma)^2 + 4(b - a\gamma)}\right]\,. \tag{A.2.12}$$

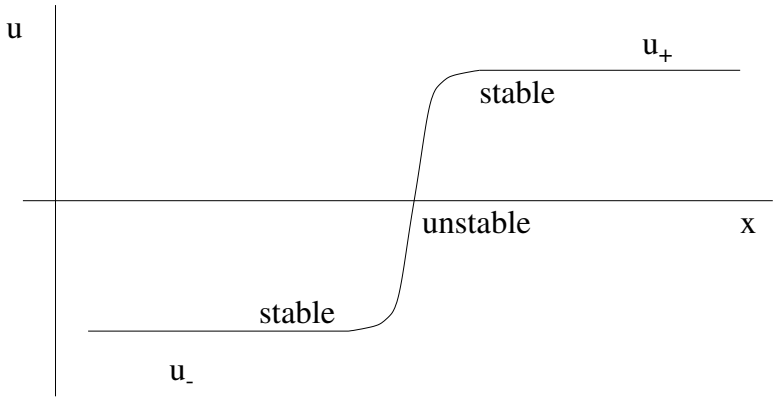

Figure 40: Front solution which connects the two stable branch of solutions $u_\pm$ through the unstable solution $u = 0$.

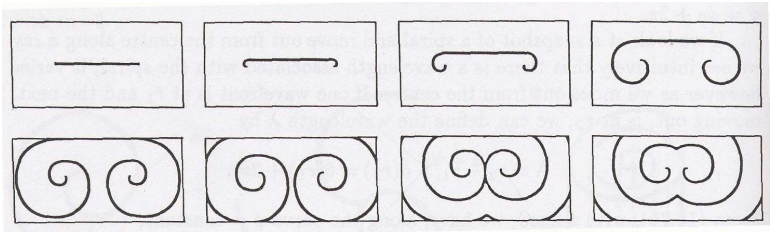

Figure 41: Spatial-temporal evolution of the solution of the Fitzhugh-Nagumo model for the parameter $D = 2 \times 10^{-6}$, $a = 0.25$, $b = 10^{-3}$, and $\gamma = 3 \times 10^{-3}$. The dark regions are where $u = u_+$ whereas the white regions are where $u = u_-$.

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. 40 where the two stable $u_\pm$ branches are connected through the unstable $u = 0$ solution. Figure 41 depicts the spatial-temporal evolution of the solution of the Fitzhugh-Nagumo model for the parameter $D = 2\times10^{-6}$, $a = 0.25$, $b = 10^{-3}$, and $\gamma = 3\times10^{-3}$. The dark regions are where $u = u_+$ whereas the white regions are where $u = u_-$. Note

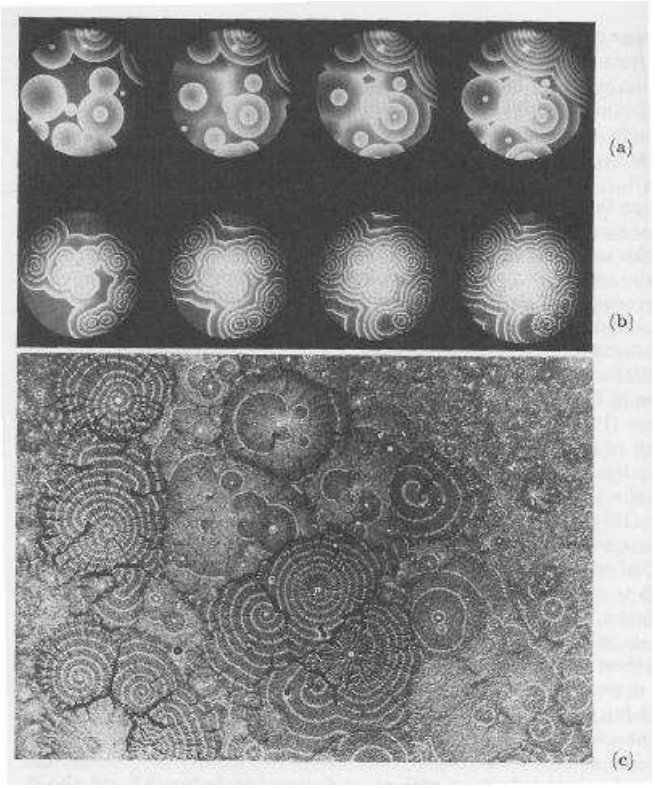

Figure 42: Experimental observations in which target patterns and spiral waves are both exhibited. These behaviors are generated in the Belousov-Zhabotinskii reaction by pacemaker nuclei (a) and (b), and by the slime mold Dictyostelium (c) which emit periodic signals of the chemical cyclic AMP, which is a chemoattractant for the cells.

the formation of spiral waves. The boundary conditions used here are no flux, i.e. $\partial u/\partial x = 0$ or $\partial u/\partial y = 0$ across the left/right and top/bottom boundaries respectively.

Not just an interesting mathematical phenomena, spiral waves are also exhibited in nature. Figure 42 illustrates a series of experimental observations in which target patterns and spiral waves are both exhibited. These behaviors are generated in the Belousov-Zhabotinskii reaction by pacemaker nuclei and by the slime mold Dictyostelium which emit periodic signals of the chemical cyclic AMP, which is a chemoattractant for the cells. Spiral waves are seen to be naturally generated in these systems and are of natural interest for analytic study.

The $\lambda - \omega$ Reaction-Diffusion System

The general reaction-diffusion system can be written in the vector form

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{f}(\mathbf{u}) + D\nabla^2 \mathbf{u} \tag{A.2.13}$$

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

$$\frac{\partial}{\partial t}\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \lambda(A) & -\omega(A) \\ \omega(A) & \lambda(A) \end{pmatrix}\begin{pmatrix} u \\ v \end{pmatrix} + D\nabla^2 \begin{pmatrix} u \\ v \end{pmatrix} \tag{A.2.14}$$

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

$$u_0(x, y) = \tanh|\mathbf{r}| \cos(m\theta - |\mathbf{r}|) \tag{A.2.15a}$$

$$v_0(x, y) = \tanh|\mathbf{r}| \sin(m\theta - |\mathbf{r}|) \tag{A.2.15b}$$

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.

A.3 Steady State Flow Over an Airfoil

The derivation for the steady-state flow over an airfoil is similar to that of the vorticity and streamfunction dynamics in shallow-water. The physical setting of interest now is geared towards the behavior of fluids around airfoils. In this case, a separation of scales is achieved by assuming the airfoil (wing) is much longer in length than its thickness or width. The scale separation once again allows for a two-dimensional study of the problem. Additionally, since we are considering the steady-state flow, the time-dependence of the problem will be eliminated. The resulting problem will be of the elliptic type. This will give the steady-state streamfunction and flow over the wing.

The physical setting for the airfoil modeling is illustrated in Fig. 43. In this figure, the characteristic wing width is given by the parameter D while the characteristic wing length is given by the parameter L. For a quasi-two-dimensional description to hold, we must have

$$\delta = \frac{D}{L} \ll 1\,. \tag{A.3.1}$$

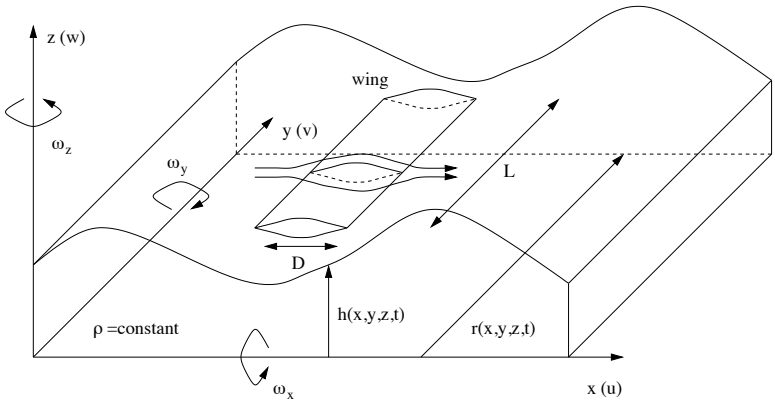

Figure 43: Physical setting of an airfoil in a constant density fluid ρ with scale separation $D/L \ll 1$.

This gives the necessary separation of scales for reducing the Navier-Stokes equations to the shallow-water description. In particular, the shallow-water reduction which led to the vorticity-streamfunction equations can be pursued here. The governing equations which result from mass conservation and momentum conservation in the x and z planes give

$$\frac{\partial r}{\partial t} + \frac{\partial}{\partial x}(ru) + \frac{\partial}{\partial z}(rw) = 0 \tag{A.3.2a}$$

$$\frac{\partial}{\partial t}(ru) + \frac{\partial}{\partial x}\left(ru^2 + \frac{1}{2}gr^2\right) + \frac{\partial}{\partial z}(ruw) = 0 \tag{A.3.2b}$$

$$\frac{\partial}{\partial t}(rw) + \frac{\partial}{\partial z}\left(rw^2 + \frac{1}{2}gr^2\right) + \frac{\partial}{\partial x}(ruw) = 0 \tag{A.3.2c}$$

where we have ignored the coriolis parameter.

At leading order, we assume $r(x, y, t) \approx$constant so that the conservation of mass equation reduces to

$$\frac{\partial u}{\partial x} + \frac{\partial w}{\partial z} = 0\,, \tag{A.3.3}$$

which is again referred to as the *imcompressible flow* condition. Under the assumption of a constant width $r(x, z, t)$, the remaining two momentum equations reduce further. The final reduction comes from taking the z derivative of the first equation and the x derivative of the second equation and performing the algebra associated with (A.1.10) through (A.1.13) which gives

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega \tag{A.3.4a}$$

$$\nabla^2 \psi = \omega \tag{A.3.4b}$$

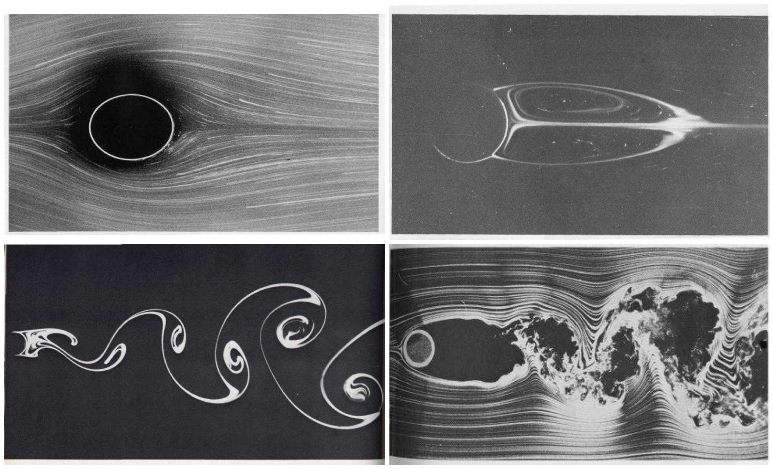

Figure 44: Fluid flow past a cylinder for increasing Reynolds number. The four pictures correspond to $R = 9.6, 41.0, 140$, and 10000. Note the transition from laminar (steady-state) flow to a recirculating steady-state flow, to Kárman vortex shedding and finally to fully turbulent flow.

where

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial z} - \frac{\partial \psi}{\partial z}\frac{\partial \omega}{\partial x} \tag{A.3.5}$$

and $\nabla^2 = \partial_x^2 + \partial_z^2$ is the two dimensional Laplacian. The diffusion component which is proportional to ν measures the frictional forces present in the fluid motion. Note that we have once again introduced the streamfunction $\psi(x, z, t)$ which is defined as

$$u = -\frac{\partial \psi}{\partial z} \quad w = \frac{\partial \psi}{\partial x}\,. \tag{A.3.6}$$

Thus the streamfunction is specified up to an arbitrary constant. Note that the streamfunction automatically satisfies the imcompressibility condition since

$$\frac{\partial u}{\partial x} + \frac{\partial w}{\partial z} = -\frac{\partial^2 \psi}{\partial x \partial z} + \frac{\partial^2 \psi}{\partial x \partial z} = 0\,. \tag{A.3.7}$$

In terms of the vorticity, the streamfunction is related as follows:

$$\omega = \omega_y = \frac{\partial w}{\partial x} - \frac{\partial u}{\partial z} = \nabla^2 \psi\,. \tag{A.3.8}$$

This gives a second equation of motion (A.3.4a) which must be considered in solving the airfoil equations.

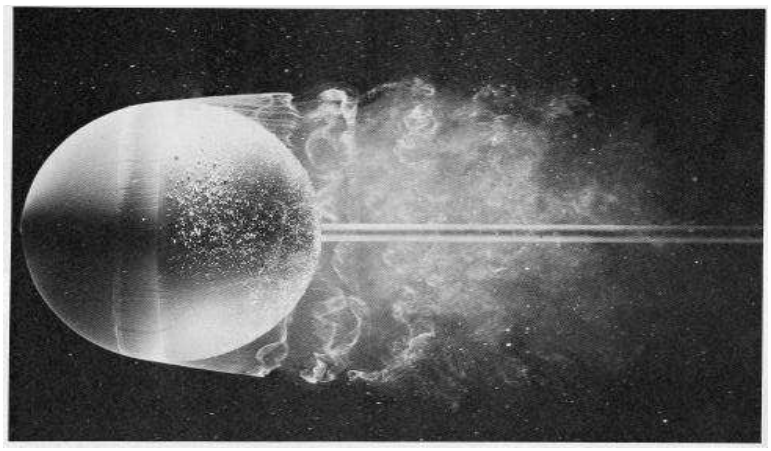

Figure 45: Turbulent flow over a sphere for Reynolds number $R = 15000$.

Steady-State Flow

The behavior of the fluid as it propagates over the airfoil can change drastically depending upon the density and velocity of the fluid. The typical measure of the combined fluid density and velocity is given by a quantity known as the *Reynolds number R*. Essentially, as the the Reynolds number increases, then the fluid is either moving at higher velocities or has a lower density. The experimental fluid flow around a cylinder, which can be thought of as an airfoil, is depicted in Fig. 44. The four pictures correspond to $R = 9.6, 41.0, 140$, and 10000. Note the transition from laminar (steady-state) flow to a recirculating steady-state flow, to Kárman vortex shedding and finally to fully turbulent flow. The same type of phenomena occurs also in three dimensions. A fully developed turbulent flow over a sphere is illustrated in Fig. 45 for $R = 15000$.

The general turbulent behavior is difficult to capture computationally. However, the steady-state behavior can be well understood by considering the equations of motion in the steady-state limit. In particular, at large times in the steady-state flow the diffusion term in the advection-diffusion equation (A.3.4a) causes the vorticity to diffuse to zero, i.e. as $t \to \infty$ then $\omega \to 0$. The stream-function equation (A.3.4b) then becomes Poisson's equation

$$\nabla^2 \psi = 0\,. \tag{A.3.9}$$

Thus an elliptic solve is the only thing required to find the steady-state behavior.

Boundary Conditions

To capture the steady-state flow which is governed by the elliptic problem, we need to develop boundary conditions for both the computational domain and air-

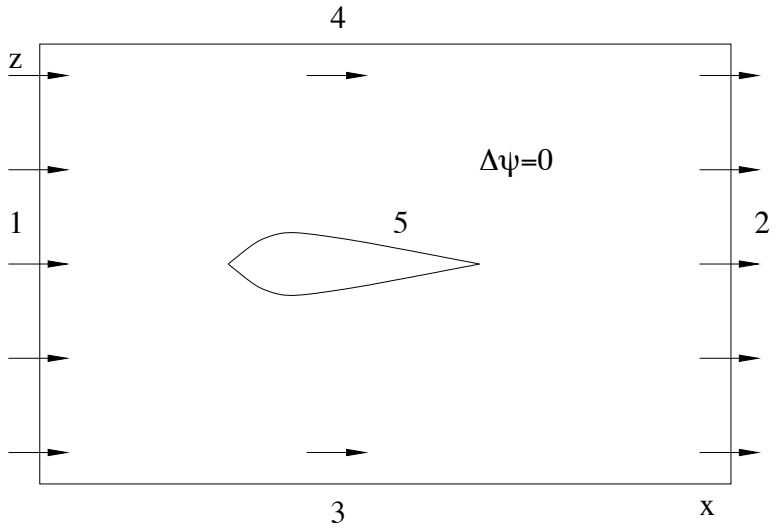

Figure 46: Computational domain for calculating the steady-state flow over an airfoil. Note the five boundaries which must be specified to reach a solution.

foil. Figure 46 illustrates the computational domain which may be implemented to capture the steady-state flow over an airfoil. Five boundary conditions are specified to determine the steady-state flow over the airfoil. These five boundary conditions on the domain $x \in [-L, L]$ and $y \in [-L, L]$ are as follows:

1. $x = -L$: The incoming flux flow it imposed at this location. Assuming the flow enters so that $u(x = -L) = u_0$ and $w(x = -L) = 0$ gives

$$u = u_0 = -\frac{\partial \psi}{\partial z} \quad \rightarrow \quad \frac{\partial \psi}{\partial z} = -u_0 \tag{A.3.10}$$

2. $x = L$: The outgoing flux flow should be indentical to that imposed for 1. Thus (A.3.10) should apply.

3. $z = -L$: Assume that far above or below the airfoil, the fluid flow remains parallel to the x direction. Thus (A.3.10) still applies. However, we also have that

$$w = 0 = \frac{\partial \psi}{\partial x} \quad \rightarrow \quad \frac{\partial \psi}{\partial x} = 0 \tag{A.3.11}$$

4. $z = L$: The boundary conditions at the top of the domain will mirror those at the bottom.

5. airfoil: the steady-state flow cannot peneratrate into the airfoil. Thus a boundary condition is required on the wing itself. Physically, a *no slip*

boundary condition is applied. Thus the fluid is stationary on the domain of the wing:

$$u = w = 0 \quad \rightarrow \quad \frac{\partial \psi}{\partial x} = \frac{\partial \psi}{\partial z} = 0 \tag{A.3.12}$$

Although the rectangular computational domain is easily handled with finite difference techniques, the airfoil itself presents a significant challenge. In particular, the shape is not regular and the finite difference technique is only equipped to handle rectangular type domains. Thus an alternative method should be used. Specifically, finite elements can be considered since it allows for an unstructured domain and can handle any complicated wing shape. Realistically, any attempt with finite difference or spectral techniques on a complicated domain is more trouble than its worth.

References

[1] W. E. Boyce and R. C. DiPrima, *Elementary Differential Equations and Boundary Value Problems*, 7th Ed. (Wiley, 2001).

[2] See, for instance, R. Finney, F. Giordano, G. Thomas, and M. Weir, *Calculus and Analytic Geometry*, 10th Ed. (Prentice Hall, 2000).

[3] C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, (Prentice Hall, 1971).

[4] J. D. Lambert, *Computational Methods in Ordinary Differential Equations*, (Wiley, 1973)

[5] R. L. Burden and J. D. Faires, *Numerical Analysis*, (Brooks/Cole, 1997).

[6] A. Greenbaum, *Iterative Methods for Solving Linear Systems*, (SIAM, 1997).

[7] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes*, 2nd Ed. (Cambridge, 1992).

[8] R. Courant, K. O. Friedrichs, and H. Lewy, "Über die partiellen differenzengleichungen der mathematischen physik," *Mathematische Annalen* **100**, 32-74 (1928).

[9] J. C. Strikwerda, *Finite Difference Schemes and Partial Differential Equations*, (Chapman & Hall, 1989).

[10] N. L. Trefethen, *Spectral methods in MATLAB*, (SIAM, 2000).

[11] G. Strang, *Introduction to Applied Mathematics*, (Wellesley-Cambridge Press, 1986).

[12] I. M. Gelfand and S. V. Fomin, *Calculus of Variations*, (Prentice-Hall, 1963).