Parallel Spectral Numerical Methods

Gong Chen, Brandon Cloutier, Ning Li, Benson K. Muite and Paul Rigge
with contributions from
Sudarshan Balakrishnan, Andre Souza and Jeremy West

September 5, 2012

Contents

List of Figures

Listings

License

Acknowledgements

The example programs have used a similar structure to those in Trefethen [56], to which the reader is referred to for further code examples. The codes for the nonlinear Schrödinger equation were developed in collaboration with Christian Klein and Kristelle Roidot. The codes for the Navier-Stokes equation were developed in collaboration with Hans Johnston. We thank Peter Miller, Brock Palen, David O'Neal, Mark Vanmoer, Divakar Viswanath, Jared Whitehead and Tim Williams for helpful comments, discussions and suggestions. A class project by Sou-Chi Chang and Sophie Zhang, influenced the presentation of the nonlinear Schrödinger equation and the exercises on the Gross-Pitaevskii equation, while a class project by Fuzhou Hu and Yutao Qin influenced the presentation and exercises for the two dimensional Navier-Stokes equations. We also thank Daniel Brahan, Emily Cizmas, Kohei Harada, Seth Jordan, Joshua Kirschenheiter, Brian Leu, Albert Liu, Thomas Olsen, Henry Rensch, Parth Sheth, Jeffrey Smolik and Matthew Warnez for constructive criticism and testing the materials. We thank Danny Ellis for providing his notes taken during a lecture on some of this material.

This tutorial was produced using resources of:

- The Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.
- Hopper at the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.
- Intrepid at the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.
- Jaguar at the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- SCREMS NSF DMS-1026317

Partial financial support was also provided by:

- A curriculum material development grant from The Blue Waters Undergraduate Petascale Education Program administered by the Shodor foundation.

- A Faculty Grant for Innovations in Teaching with Technology from the division of Literature, Sciences and Arts at the University of Michigan.
- CNRS (Centre National de la Recherche Scientifique)

This tutorial utilizes the following packages:

- Mcode created by Florian Knorn which can be downloaded at `http://www.mathworks.com/matlabcentral/fileexchange/8015-m-code-latex-package`

Chapter 1

Overview

1.1 Summary

We start by taking a quick look at finite-precision arithmetic. We then discuss how to solve ordinary differential equations (ODE) and partial differential equations (PDE) using the technique of separation of variables. We then introduce numerical time-stepping schemes that can be used to solve ODEs and PDEs. Next we introduce pseudo spectral methods by giving an overview of the discrete Fourier Transform (DFT) and the Fast Fourier Transform (FFT) algorithm that is used to quickly calculate the DFT. Finally we will combine all of this to solve a couple of different PDEs first in a serial setting and then in a parallel setting. The programs will use Matlab[1] and Fortran. A Python[2] implementation of some of the Matlab programs is also provided.

1.2 Prerequisites

We assume that the reader has introductory programming experience, for example using C, C++, Fortran, Matlab, Octave, Python or equivalent. Since detailed programming examples have been provided, we do not expect a significant programming background, but hope the required knowledge will be acquired as one works through the examples. We also assume the level of mathematical maturity obtained in a demanding calculus course, for example at the level of Courant and Johns "Introduction to Calculus and Analysis". A course in differential equations would also be helpful, but for many scientists or engineers, their fields of interest will provide numerous examples of these. More programming experience or mathematical background will make the material easier to understand. Checking whether the simulations are correct may also be easier for those with knowledge of the behavior of solutions of the partial differential equations that are being approximated, however we have tried to choose

[1]`http://www.mathworks.com/products/matlab/index.html` – if this is not available, we suggest modifying the Matlab programs to use Octave which can be freely downloaded at `http://www.gnu.org/software/octave/`.

[2]`http://python.org/`

representative differential equations that will make it easy for one to use the programs and then adapt them to the use being considered.

1.3 Using the Programs

The programs have been tested on several different computers. The programs are located in program directories which correspond to the chapter in which the programs first appear. While they are not explicitly hyperlinked, one can find their locations either by reading the LaTeX source code or by searching the appropriate directory.

The Matlab programs are guaranteed to work with Matlab R2011b, but should also work with other recent versions of Matlab. They should also be easy to modify so that they work with Octave. The Fortran programs have been tested primarily with the GCC 4.6.2 compiler suite, although they should work with most other recent compilers. If using an implementation of MPI that depends on a particular compiler, we suggest also using the GCC compiler. We expect that the programs should work with minor modifications with other compilers, but cannot guarantee this. For simplicity and to allow checking of program correctness, we have chosen to use a low compiler optimization level. We encourage users to increase the compiler optimization level and compiler flags once they have checked that the programs are working correctly on their systems. FFTW, a free Fast Fourier transform library, is also required to run the programs. This can be downloaded from `http://fftw.org/`. The MPI programs make use of the library 2DECOMP&FFT which can be downloaded from `http://www.2decomp.org`. Finally, the last part of the tutorial requires the use of the free and open source VisIt parallel visualization program, which can be obtained from `https://wci.llnl.gov/codes/visit/home.html`. If you expect to do large parallel simulations (A guide for large at present is 20% of the system for systems larger than 10,000 cores), it may be worth learning the most efficient system settings for performing output and for parallelization. We do not address this in this tutorial, but suggest that you contact your computing center for suggestions.

1.4 Course Outlines / Assessment Rubric

The material in these notes can form the basis of a short course. The most important portions are chapters 1 to 11. A selection can then be made from chapters 12, 13 and 14. A selection of the problems can be used to assess student learning. Note that problems in chapters 8, 12, 13 and 14 can develop into extensive research projects, so only a sample of these should be given to students if they only have a limited time to solve them. A student will have successfully understood the material if they can run the example Matlab/Python, serial Fortran, OpenMP Fortran and MPI Fortran programs, and can also modify them to solve related problems. Successful completion of problems which test these abilities will be enough to indicate that students have understood the fundamental concepts.

Chapter 2

Finite Precision Arithmetic

[1] Because computers have a fixed amount of memory, floating point numbers can only be stored with a finite number of digits of precision. This limits the accuracy to which the solution to a numerical problem can be obtained in finite time. Most computers use binary IEEE 754 arithmetic to perform numerical calculations. There are other formats, but this will be the one of most relevance to us.

2.1 Exercises

1) Download the most recent IEEE 754 standard. `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=2355`, see also `http://grouper.ieee.org/groups/754/` – unfortunately the links to the official standard requires either IEEE membership or a subscription. If you do not have this please see the wikipedia page (`http://en.wikipedia.org/wiki/IEEE_754-2008`) for the information you will need to answer the questions below[2].
    a) In this standard what is the range and precision of numbers in:
        i) Single precision
        ii) Double precision
    b) What does the standard specify for quadruple precision?
    c) What does the standard specify about how elementary functions should be computed? How does this affect the portability of programs?
2) Suppose we discretize a function for $x \in [-1, 1]$. For what values of ϵ is

$$\epsilon \log\left(\cosh\left(\frac{x}{\epsilon}\right)\right) = |x|$$

in

[1]For more on this see a text book on numerical methods such as Bradie [4].

[2]These links are correct as of 1 April 2012, should they not be active, we expect that the information should be obtained by a search engine or by referring to a numerical analysis textbook such as Bradie [4].

i) Single precision?

ii) Double precision?

3) Suppose we discretize a function for $x \in [-1, 1]$. For what values of ϵ is

$$\tanh\left(\frac{x}{\epsilon}\right) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

in

i) Single precision?

ii) Double precision?

4) a) What is the magnitude of the largest 4 byte integer in the IEEE 754 specification that can be stored?

b) Suppose you are doing a simulation with N^3 grid points and need to calculate N^3. If N is stored as a 4 byte integer, what is the largest value of N for which N^3 can also be stored as a 4 byte integer?

Chapter 3

Separation of Variables

Separation of variables is a technique which can be used to solve both ODEs and PDEs. The basic idea for an equation in two variables is to rewrite the equation so that each of the two variables is located on different sides of an equality sign, and since both sides of the equation depend on different variables, the two sides must be equal to a constant. We introduce this idea with the simple first order linear ODE

$$\frac{dy}{dt} = y. \tag{3.1}$$

As long as $y(t) \neq 0$ for any value of t, we can formally separate variables and rewrite eq. (3.1) as

$$\frac{dy}{y} = dt. \tag{3.2}$$

Now we can solve for $y(t)$ by integrating both sides

$$\begin{aligned}
\int \frac{dy}{y} &= \int dt && (3.3)\\
\ln y + a &= t + b && (3.4)\\
e^{\ln y + a} &= e^{t+b} && (3.5)\\
e^{\ln y} e^{a} &= e^{t} e^{b} && (3.6)\\
y &= \frac{e^{b}}{e^{a}} e^{t} && (3.7)\\
y(t) &= ce^{t}. && (3.8)
\end{aligned}$$

Where a, b, and c are arbitrary constants of integration.

We now perform a similar example for a linear partial differential equation. The heat equation is

$$u_t = -u_{xx}. \tag{3.9}$$

We suppose that $u = X(x)T(t)$, so that we obtain

$$X(x)\frac{\mathrm{d}T}{\mathrm{d}t}(t) = -\frac{\mathrm{d}^2X}{\mathrm{d}x^2}(x)T(t). \tag{3.10}$$

We can rewrite this as

$$\frac{\frac{\mathrm{d}T}{\mathrm{d}t}(t)}{T(t)} = \frac{\frac{\mathrm{d}^2X}{\mathrm{d}x^2}(x)}{X(x)} = -C, \tag{3.11}$$

where C is a constant independent of x and t. The two sides can be integrated separately to get $T(t) = \exp(-Ct)$ and either $X(x) = \sin(\sqrt{C}x)$ or $X(x) = \cos(\sqrt{C}x)$. Since the heat equation is linear, one can then add different solutions to the heat equation and still obtain a solution of the heat equation. Hence solutions of the heat equation can be found by

$$\sum_n \alpha_n \exp(-C_n t)\sin(\sqrt{C_n}x) + \beta_n \exp(-C_n t)\cos(\sqrt{C_n}x) \tag{3.12}$$

where the constants α_n, β_n and C_n are appropriately chosen. Convergence of such series to an actual solution is studied in mathematics courses on analysis (see for example Evans [17] or Renardy and Rogers [50]), however the main ideas necessary to choose the constants, α_n, β_n and C_n and hence construct such solutions are typically encountered towards the end of a calculus course or at the beginning of a differential equations course, see for example Courant and John [13] or Boyce and DiPrima [6]. Here, we consider the case where $x \in [0, 2\pi]$, and for which we have periodic boundary conditions. In this case $\sqrt{C_n}$ must be integers, which we choose to be non-negative to avoid redundancies. At time $t = 0$, we shall suppose that the initial condition is given by

$$u(x, t = 0) = f(x). \tag{3.13}$$

Now,

$$\int_0^{2\pi} \sin(nx)\sin(mx) = \begin{cases} \pi & m = n \\ 0 & m \neq n \end{cases}, \tag{3.14}$$

$$\int_0^{2\pi} \cos(nx)\cos(mx) = \begin{cases} \pi & m = n \\ 0 & m \neq n \end{cases}, \tag{3.15}$$

and

$$\int_0^{2\pi} \cos(nx)\sin(mx) = 0. \tag{3.16}$$

Thus we can consider the trigonometric polynomials as being orthogonal vectors. It can be shown that a sum of these trigonometric polynomials can be used to approximate a wide class of periodic functions on the interval $[0, 2\pi]$; for well behaved functions, only the first few terms in such a sum are required to obtain highly-accurate approximations. Thus, we can suppose that

$$f(x) = \sum_n \alpha_n \sin(\sqrt{C_n}x) + \beta_n \cos(\sqrt{C_n}x). \tag{3.17}$$

Multiplying the above equation by either $\sin(\sqrt{C_n}x)$ or $\cos(\sqrt{C_n}x)$ and using the orthogonality of the functions, we deduce that

$$\alpha_n = \frac{\int_0^{2\pi} f(x)\sin(\sqrt{C_n}x)\mathrm{d}x}{\int_0^{2\pi}\sin^2(\sqrt{C_n}x)\mathrm{d}x} \tag{3.18}$$

and

$$\beta_n = \frac{\int_0^{2\pi} f(x)\cos(\sqrt{C_n}x)\mathrm{d}x}{\int_0^{2\pi}\cos^2(\sqrt{C_n}x)\mathrm{d}x}. \tag{3.19}$$

Most ODEs and PDEs of practical interest will not be separable. However, the ideas behind separation of variables can be used to allow one to find series solutions to a wide class of PDEs. These series solutions can also be found numerically and are what we will use to find approximate solutions to PDEs, and so the ideas behind this simple examples are quite useful.

3.1 Exercises

1) Solve the ordinary differential equation

$$u_t = u(u-1) \quad u(t=0) = 0.8$$

using separation of variables.

2) a) Use separation of variables to solve the partial differential equation

$$u_{tt} = u_{xx}$$

with

$$u(x=0,t) = u(x=2\pi,t),$$
$$u(x,t=0) = \sin(6x) + \cos(4x)$$

and

$$u_t(x,t=0) = 0.$$

b) Create plots of your solution at several different times and/or create an animation of the solution you have found.[1]

c) The procedure required to find the coefficients in the Fourier series expansion for the initial condition can become quite tedious/intractable. Consider the initial condition $u(x,t=0) = \exp(\sin(x))$. Explain why it would be difficult to compute the Fourier coefficients for this by hand. Also explain why it would be nice to have an algorithm or computer program that does this for you.

[1]Your solution should involve only a few modes and so you should be able to use a wide variety of software to create plots, for example a graphing calculator, a spreadsheet program such as Excel, Mathematica, Wolfram Alpha, Matlab, Maple, Python, Sage etc. You can use Wolfram Alpha and Sage online.

Chapter 4

Motivation for Numerical Methods

Many partial differential equations do not have exact closed-form solutions for all choices of initial conditions[1]. Irregular boundary conditions can also make finding an analytic solution difficult for many partial differentail equation. In these cases, finding an approximate solution with a numerical method can be helpful either for physical purposes, engineering purposes or for mathematical investigations of the behavior of solutions to these partial differential equations. There are also cases where the partial differential equations have explicitly known exact solutions, but the formulae used to express the exact solutions require a large number of computations to evaluate them[2]. In this case we are interested in making numerical approximations that result in accurate and cost-efficient solutions.

Numerical methods allows us to use a computer to calculate approximate solutions to partial differential equations. The accuracy of the solution will depend on which numerical method is used and usually more accurate numerical methods tend to be more complicated than less accurate methods. We will therefore start with some simple numerical methods to familiarize ourselves with how numerical methods work. We encourage the reader to take a full course on the numerical solution of partial differential equations as well as reading the references to find out about numerical techniques not discussed here.

[1]An example is the Navier-Stokes equation which is thought to describe the motion of an incompressible viscous fluid.

[2]An example is the sine-Gordon equation.

Chapter 5

Timestepping

We now briefly discuss how to solve initial value problems. For more on this see Bradie [4, Chap. 7]. A slightly longer but still quick introduction to these ideas can also be found in Boyce and DiPrima [6].

5.1 Forward Euler

In order to compute solutions to differential equations on computers efficiently, it is convenient to do our calculations at a finite number of specified points and then interpolate between these points. For many calculations it is convenient to use a grid whose points are equally distant from each other.

For the rest of the section h will be our step size, which is assumed to be constant. When solving an ODE or PDE, the choice of h isn't selected at random, but rather requires some intuition and/or theoretical analysis. We are going to start with the forward Euler method which is the most basic numerical method. Let us first denote the time at the nth time-step by t^n and the computed solution at the n^{th} time-step by y^n, where $y^n \equiv y(t = t^n)$. The step size h in terms of t is defined as $h = t^{n+1} - t^n$. Lets first start with a basic ODE with initial conditions, in which $f(t, y)$ is some arbitrary function and $y(t)$ is our solution,

$$\frac{dy}{dt} = f(t, y) \qquad y(t^0) = y^0. \tag{5.1}$$

The differential equation can be approximated by finite differences,

$$\frac{y^{n+1} - y^n}{h} = f(t^n, y^n). \tag{5.2}$$

Now all we have to do is solve for y^{n+1} algebraically,

$$y^{n+1} = y^n + hf(t^n, y^n) \qquad \text{(Forward Euler/Explicit method)} \tag{5.3}$$

If we wanted to calculate $\frac{dy}{dt}$ at time t^0, then we could generate an approximation for the value at time t^{n+1} using (5.3) by first finding $y(t^0)$ and using it to compute y^{n+1}. We then repeat this process until the final time is reached.

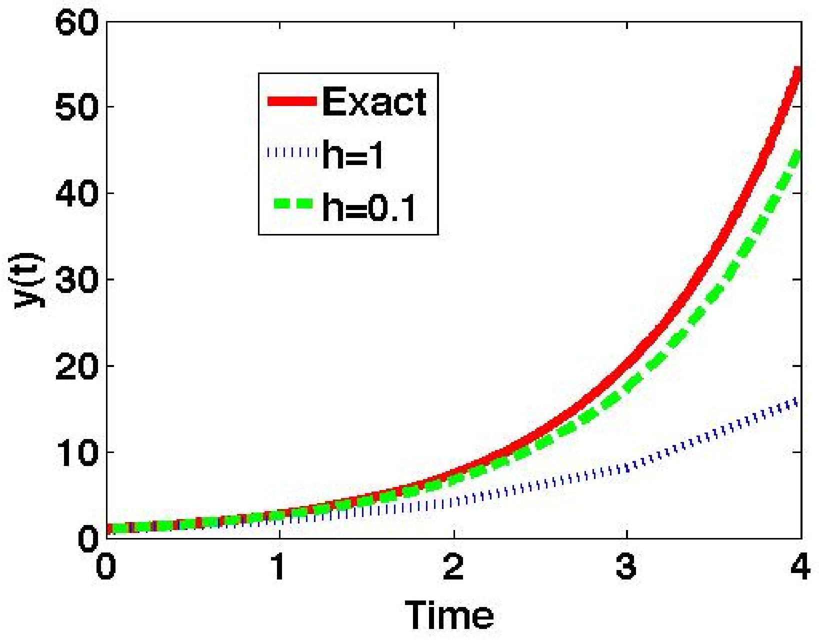

Figure 5.1: A numerical solution to the ODE in eq. (5.1) with $f(t, y) = y$ demonstrating the accuracy of the Forward Euler method for different choices of timestep.

5.1.1 An Example Computation

Let us consider the ODE in eq. (5.1) with $f(t, y) = y$ and initial conditions $y(t^0) = 1$ where $t^0 = 0$. Two numerical solutions are computed using the forward Euler method with $h = 1$ and $h = .1$

It should be no surprise that a smaller step size like $h = .1$ compared to $h = 1$ will be more accurate. Looking at the line for $h = 1$, you can see that $y(t)$ is calculated at only 4 points then straight lines interpolate between each point. This is obviously not very accurate, but gives a rough idea of what the function looks like. The solution for $h = .1$ might require 10 times more steps to be taken, but it is clearly more accurate. Forward Euler is an example of a first-order method and approximates the exact solution using the first two terms in the Taylor expansion[1]

$$y(t^n + h) = y(t^n) + h \left.\frac{dy}{dt}\right|_{t^n} + \mathrm{O}(h^2), \tag{5.4}$$

where terms of higher order than $\mathrm{O}(h^2)$ are omitted in the approximate solution. Substituting this into eq. (5.3) we get that

$$y^n + h \left.\frac{dy}{dt}\right|_{t^n} + \mathrm{O}(h^2) = y^n + hf(t^n, y^n)$$

[1]The derivation of the Taylor expansion can be found in most books on calculus.

after cancelling terms and dividing by h, we get that

$$\left.\frac{dy}{dt}\right|_{t^n} + \mathrm{O}(h) = f(t^n, y^n),$$

from which it is clear that the accuracy of the method changes linearly with the step size, and hence it is first-order accurate.

5.2 Backwards Euler

A variation of forward Euler can be obtained by approximating a derivative by using a backward difference quotient. Using eq. (5.1) and applying

$$\frac{y^n - y^{n-1}}{h} \approx f(t^n, y^n) \tag{5.5}$$

$$y^n = y^{n-1} + hf(t^n, y^n). \tag{5.6}$$

Stepping the index up from n to $n+1$ we obtain,

$$y^{n+1} = y^n + hf(t^{n+1}, y^{n+1}) \qquad \text{(Backwards Euler/Implicit method)} \tag{5.7}$$

Notice how y^{n+1} is not written explicitly like it was in the forward Euler method. This equation instead implicitly defines y^{n+1} and must be solved to determine the value of y^{n+1}. How difficult this is depends entirely on the complexity of the function f. For example, if f is just y^2, then the quadratic formula could be used, but many nonlinear PDEs require other methods. Some of these methods will be introduced later.

5.3 Crank-Nicolson

By taking an average of the forward and backward Euler methods, we can find the Crank-Nicolson method:

$$\frac{y^{n+1} - y^n}{h} = \frac{1}{2}f(t^{n+1}, y^{n+1}) + \frac{1}{2}f(t^n, y^n) \tag{5.8}$$

Rearranging we obtain,

$$y^{n+1} = y^n + \frac{h}{2}\left[f(t^{n+1}, y^{n+1}) + f(t^n, y^n)\right] \qquad \text{(Crank-Nicolson)} \tag{5.9}$$

Notice again how y^{n+1} is not written explicitly like it was in forward Euler. This equation instead implicitly defines y^{n+1} and so the equation must be solved algebraically to obtain y^{n+1}.

5.4 Stability of Forward Euler, Backward Euler and Crank-Nicolson

Let's look at the following ODE

$$\frac{dy}{dt} = -\lambda y(t) \tag{5.10}$$

where λ is a constant and $y(t^0) = 1$ where $t^0 = 0$. Lets numerically solve this ODE using the forward Euler, backward Euler and Crank-Nicolson time-stepping schemes. The results are as follows

$$y^{n+1} = y^n - \lambda h y^n \qquad \text{(Forward Euler)} \tag{5.11}$$

$$y^{n+1} = \frac{y^n}{(1+\lambda h)} \qquad \text{(Backward Euler)} \tag{5.12}$$

$$y^{n+1} = y^n \left(\frac{2-\lambda h}{2+\lambda h}\right) \qquad \text{(Crank-Nicolson)} \tag{5.13}$$

and the exact solution is given by

$$y(t) = e^{-\lambda t} \qquad \text{(Exact solution)} \tag{5.14}$$

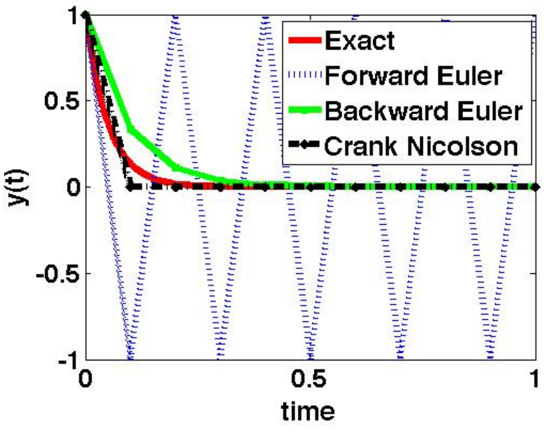

Figure 5.2: A numerical solution to the ODE in eq. (5.10) with $\lambda = 20$ and with a timestep of $h = 0.1$ demonstrating the instability of the Forward Euler method and the stability of the Backward Euler and Crank Nicolson methods.

Figure 5.2 above shows how both methods converge to the solution, but the forward Euler solution is unstable for the chosen timestep. Listing 5.1 is a Matlab program where you can play around with the value of λ to see how, for a fixed timestep, this changes the stability of the method.

Listing 5.1: A Matlab program to demonstrate instability of different timestepping methods.

```matlab
% A program to demonstrate instability of timestepping methods
% when the timestep is inappropriately choosen.

%Differential equation: y'(t)=-y(t) y(t_0)=y_0
%Initial Condition, y(t_0)=1 where t_0=0
clear all; clc; clf;

%Grid
h=.1;
tmax=4;
Npoints = tmax/h;
lambda=.1;

%Initial Data
y0=1;
t_0=0;
t(1)=t_0;
y_be(1)=y0;
y_fe(1)=y0;
y_imr(1)=y0;

for n=1:Npoints
  %Forward Euler
    y_fe(n+1)=y_fe(n)-lambda*h*y_fe(n);
        %Backwards Euler
    y_be(n+1)=y_be(n)/(1+lambda*h);
      %Crank Nicolson
    y_imr(n+1)=y_imr(n)*(2-lambda*h)/(2+lambda*h)
    t(n+1)=t(n)+h;
end

%Exact Solution
tt=[0:.001:tmax];
exact=exp(-lambda*tt);

%Plot
figure(1); clf; plot(tt,exact,'r-',t,y_fe,'b:',t,y_be,'g--',t,y_imr,'k-.')
    ;
xlabel time; ylabel y;
legend('Exact','Forward Euler','Backward Euler',...
    'Crank Nicolson');
```

5.5 Stability and Accuracy of Forward Euler, Backward Euler and Crank-Nicolson Time Stepping Schemes for $y' = -\lambda y$

The examples discussed show that numerical stability is an important consideration when finding approximate solutions to differential equations on computers. Numerical stability requires a careful choice of numerical method and timestep for each numerical solution to a differential equation. We now try to understand these observations so that we have some guidelines to design numerical methods that are stable. The numerical solution to an initial value problem with a bounded solution is **stable** if the numerical solution can be bounded by functions which are independent of the step size. There are two methods which are typically used to understand stability. The first method is linearized stability, which involves calculating eigenvalues of a linear system to see if small perturbations grow or decay. A second method is to calculate an energy like quantity associated with the differential equation and check whether this remains bounded.

We shall assume that $\lambda \geq 0$ so that the exact solution to the ODE does not grow without bound. The forward Euler method gives us

$$\begin{aligned}
&\frac{y^{n+1} - y^n}{h} = -\lambda y^n \\
&y^{n+1} = (1 - \lambda h) y^n \\
&\Rightarrow |y^{n+1}| \geq |(1 - \lambda h)||y^n| \quad \text{if } |(1 - \lambda h)| > 1 \\
&\Rightarrow |y^{n+1}| \leq |(1 - \lambda h)||y^n| \quad \text{if } |(1 - \lambda h)| < 1.
\end{aligned}$$

We can do a similar calculation for backward Euler to get

$$\begin{aligned}
&\frac{y^{n+1} - y^n}{h} = -\lambda y^{n+1} \\
&y^{n+1} = \frac{y^n}{1 + \lambda h} \\
&\Rightarrow |y^{n+1}| \leq \left| \frac{y^n}{1 + \lambda h} \right| \leq |y^n| \quad \text{since } \left| \frac{1}{1 + \lambda h} \right| < 1.
\end{aligned}$$

Thus, the backward Euler method is unconditionally stable, whereas the forward Euler method is not. We leave the analysis of the Crank-Nicolson method as an exercise.

A second method, often used to show stability for partial differential equations is to look for an energy like quantity and show that this bounds the solution and prevents it from becoming too positive or too negative. Usually, the quantity is chosen to be non negative, then all one needs to do is deduce there is an upper bound. We sketch how this is done for an ordinary differential equation so that we can use the same ideas when looking at partial differential equations. Recall that the forward Euler algorithm is given by

$$\frac{y^{n+1} - y^n}{h} = -\lambda y^n.$$

Multiplying this by y^{n+1} we find that

$$(y^{n+1})^2 = (1 - h\lambda)y^n y^{n+1}.$$

Now to obtain a bound on $|y^{n+1}|$ in terms of $|y^n|$, we use the following fact

$$(a-b)^2 \geq 0 \Rightarrow a^2 + b^2 \geq 2ab \Rightarrow \frac{(y^{n+1})^2 + (y^n)^2}{2} \geq y^n y^{n+1}.$$

Hence a sufficient condition for stability if

$$(1 - h\lambda) > 0$$

is that

$$\begin{aligned}
(y^{n+1})^2 &\leq (1 - h\lambda)\frac{(y^{n+1})^2 + (y^n)^2}{2} \\
(y^{n+1})^2\frac{1 + h\lambda}{2} &\leq \frac{1 - h\lambda}{2}(y^n)^2 \\
(y^{n+1})^2 &\leq \frac{1 - h\lambda}{1 + h\lambda}(y^n)^2,
\end{aligned}$$

thus if $1 - h\lambda > 0$, then $0 < \frac{1-h\lambda}{1+h\lambda} < 1$ and so we have stability, we again see that the algorithm is stable provided the timestep is small enough. There are many situations for which λ is large and so the timestep, h needs to be very small. In such a situation, the forward Euler method can be very slow on a computer.

Stability is not the only requirement for a numerical method to approximate the solution to an initial value problem. We also want to show that as the timestep is made smaller, the numerical approximation becomes better. For the forward Euler method we have that

$$\frac{y^{n+h} - y^n}{h} = -\lambda y^n$$

now if

$$\begin{aligned}
y^n &= y(t) \\
y^{n+1} &= y(t+h)
\end{aligned}$$

then[2]

$$y^{n+1} = y(t) + h\frac{\mathrm{d}y}{\mathrm{d}t} + O(h^2)$$

[2]We will use big ‘Oh’ to mean that there exists a constant so that if f $O(h)$, then for $h \to 0$, we have that $\left|\frac{f}{h}\right| < C$, where C is some constant.

so

$$\begin{aligned}\frac{y^{n+1}-y^n}{h}+\lambda y^n &= \frac{y(t+h)-y(t)}{h}+\lambda y(t)\\ &= \frac{\mathrm{d}y}{\mathrm{d}t}+O(h)+\lambda y(t)\\ &= O(h).\end{aligned}$$

We can do a similar calculation to show that the Crank-Nicolson method is second-order. In this case however, we use Taylor expansions around $y(t+h/2)$.

$$\frac{y^{n+1}-y^n}{h} = -\lambda\frac{y^{n+1}+y^n}{2}$$

so

$$\begin{aligned}y^{n+1} &= y(t+h) = y(t+h/2)+(h/2)\frac{\mathrm{d}y}{\mathrm{d}t}+(h/2)^2\frac{1}{2}\frac{\mathrm{d}^2y}{\mathrm{d}t^2}+O(h^3)\\ y^n &= y(t) = y(t+h/2)-(h/2)\frac{\mathrm{d}y}{\mathrm{d}t}+(h/2)^2\frac{1}{2}\frac{\mathrm{d}^2y}{\mathrm{d}t^2}+O(h^3)\end{aligned}$$

hence

$$\begin{aligned}\frac{y^{n+1}-y^n}{h}+\lambda\frac{y^{n+1}+y^n}{2} &= \frac{\mathrm{d}y}{\mathrm{d}t}+O(h^2)+\lambda\left[y(t+h/2)+O(h^2)\right]\\ &= O(h^2).\end{aligned}$$

Thus this is a second-order method.

5.6 Exercises

1) Determine the real values of λ and timestep h for which the implicit midpoint rule is stable for the ODE

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -\lambda y$$

Sketch the stable region in a graph of λ against timestep h.

2) Show that the backward Euler method is a first-order method.

Chapter 6

One-Dimensional Discrete Fourier Transforms

[1] The discrete Fourier transform (DFT) takes a function sampled at a finite number of points and finds the coefficients for the linear combination of trigonometric polynomials that best approximates the function; the number of trigonometric polynomials used is equal to the number of sample points. Suppose we have a function $f(x)$ which is defined on the interval $a \leq x \leq b$. Due to memory limitations, a computer can only store values at a finite number of sample points, i.e. $a \leq x_0 < x_1 < ... < x_n \leq b$. For our purposes these points will be equally spaced, for example $x_1 - x_0 = x_3 - x_2$, and so we can write

$$x_j = a + jh, \qquad j = 0, 1, 2, ..., n \tag{6.1}$$

where x_j are the *sample points*, n is the number of sample points and

$$h = \frac{b-a}{n}. \tag{6.2}$$

It is convenient to use the *standard interval*, for which $0 \leq x \leq 2\pi$. Rewriting x in terms of standard interval yields

$$x_0 = 0, x_1 = \frac{2\pi}{n}, x_2 = \frac{4\pi}{n}, x_j = \frac{2j\pi}{n}, ..., x_{n-1} = \frac{2(n-1)\pi}{n} \tag{6.3}$$

Notice how $x_n = 2\pi$ is omitted; periodicity implies that the value of the function at 2π is the same as the value of the function at 0, so it need not be included. We will introduce the DFT using the language of linear algebra. Much of this formalism carries over to continuous functions that are being approximated. It also makes it easier to understand the computer implementation of the algorithms. Many computer packages and programs are optimized to perform calculations through matrix operations, so the formalism is also useful when actually calculating transforms. We write the approximation to $f(x)$ at the sample points as a finite dimensional vector

$$\boldsymbol{f} = (f_0, f_1, ..., f_{n-1})^T = (f(x_0), f(x_1), ..., f(x_{n-1})) \tag{6.4}$$

[1] For more detail, see Olver and Shakiban [47].

where

$$f_j = f(x_j) = f\left(\frac{2j\pi}{n}\right). \tag{6.5}$$

The DFT decomposes the sampled function $f(x)$ into a linear combination of complex exponentials, $\exp(ikx)$ where k is an index. Since

$$\exp(ikx) = \cos(kx) + i\sin(kx), \tag{6.6}$$

we also obtain an expansion in trigonometric functions, which may be more familiar from courses in calculus and differential equations. Since the function is sampled at n points, the highest frequency of oscillation that can be resolved will have n oscillations. Any frequencies higher than n in the original function are not adequately resolved and cause an *aliasing* error (see, for example, Boyd [7] or Uecker [59] for more on this). This error can be reduced by sampling at a greater number of points so that the number of approximating exponentials functions can also be increased. There is a tradeoff between increasing the accuracy of the simulation and the time required for the simulation to complete. For many cases of scientific and practical interest, simulations with up to thousands of grid points can be computed relatively quickly. Below we explain how a function $f(x)$ can be approximated by an interpolating trigonometric polynomial $p(x)$ so that

$$f(x) \approx p(x) = c_0 + c_1 e^{2ix} + c_2 e^{2ix} + ... + c_{n-1}e^{(n-1)ix} = \sum_{k=0}^{n-1} c_k e^{ikx} \tag{6.7}$$

The $\approx$ symbol means that $f(x)$ and $p(x)$ agree on each sample point, i.e., $f(x_j) = p(x_j)$ for each $j = 0, 1, ...n-1$, but the interpolated polynomial $p(x)$ is only an approximation of the true solution $f(x)$ away from the sample points.. The c_n are called discrete *Fourier coefficients* and are what we will be looking to solve for. $p(x)$ represents the values of interpolating trigonometric polynomial of degree $\leq n-1$, so if we have the values of these coefficients then we have a function we can use as an approximation of $f(x)$. Since we are working in a finite-dimensional vector space, a useful approach is to rewrite the discrete Fourier series as a vector. We let

$$\begin{aligned} \boldsymbol{\omega_k} &= (e^{ikx_0}, e^{ikx_1}, e^{ikx_2}, ..., e^{ikx_n})^T & (6.8)\\ &= (1, e^{2k\pi i/n}, e^{4k\pi i/n}, ..., e^{2(n-1)k\pi i/n})^T, & (6.9) \end{aligned}$$

where $k = 0, 1, ..., n-1$. The interpolation conditions, $f(x_j) = p(x_j)$, can also be rewritten in vectorial form

$$\boldsymbol{f} = c_0\boldsymbol{\omega_0} + c_1\boldsymbol{\omega_1} + ... + c_{n-1}\boldsymbol{\omega_{n-1}}. \tag{6.10}$$

Here $\boldsymbol{f}$ is a vector evaluated at the sample points, which is decomposed into vectors $\boldsymbol{\omega}_k$, much as a vector in three dimensional space can be decomposed into the components in the x, y and z directions. The DFT allows us to compute the coefficients c_i given the value of the function at the sample points. This may at first seem unmotivated, but in many applications, such as solving differential equations, it is easier to manipulate a linear combination of trigonometric polynomials, $\boldsymbol{\omega_0}, ..., \boldsymbol{\omega_{n-1}}$, than it is to work with the original function. In order to solve for c_k, we use the orthonormality of the basis elements $\boldsymbol{\omega_0}, ..., \boldsymbol{\omega_{n-1}}$. We now explain how this is done [2].

Define $\xi_n = e^{2\pi i/n}$. We observe that

$$(\xi_n)^n = \exp\left(\frac{2\pi i n}{n}\right) = \cos(2\pi) + i\sin(2\pi) = 1 \tag{6.11}$$

For this reason ξ_n is known as the primitive n^{th} root of unity. Note also that for $0 \le k < n$, we have that $(\xi_n^k)^n = 1$, so all other roots of unity when taken to the power n can be obtained from the primitive n^{th} root of unity. We will use this to perform the DFT algorithm to calculate the coefficients $c_0, ..., c_{k-1}$ in eq. (6.10). The main idea behind the DFT algorithm is to use orthogonality of the vectors $\boldsymbol{\omega}_k$. To show the orthogonality between the vectors $\boldsymbol{\omega}_k$ and $\boldsymbol{\omega}_l$, we let $\boldsymbol{\omega}_l^*$ denote the complex conjugate of $\boldsymbol{\omega}_l$, and then take the inner product of $\boldsymbol{\omega}_k$ and $\boldsymbol{\omega}_l$ and find that

$$\begin{aligned}
\langle \boldsymbol{\omega}_k, \boldsymbol{\omega}_l \rangle &= \frac{1}{n}\sum_{m=0}^{n-1} \exp\left(\frac{2\pi i k m}{n}\right)\left[\exp(\frac{2\pi i l m}{n})\right]^* \\
&= \frac{1}{n}\sum_{m=0}^{n-1} \exp\left(\frac{2\pi i (k-l) m}{n}\right) \\
&= \frac{1}{n}\sum_{m=0}^{n-1} \cos\left(\frac{\pi (k-l) m}{n}\right) + i\sin\left(\frac{\pi (k-l) m}{n}\right) \\
&= \begin{cases} 1 & \text{if } k = l \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

To deduce the last part, if $k = l$ then $\exp(0) = 1$, and if $k \ne l$, then we are sampling the sine and cosine functions at equally spaced points on over an integral number of wavelengths. Since these functions have equal magnitude positive and negative parts, they sum to zero, much as the integral of a sine or cosine over an integral number of wavelengths is zero. This implies that we can compute the Fourier coefficients in the discrete Fourier sum by taking inner products

$$c_k =< \boldsymbol{f}, \boldsymbol{\omega_k} >= \frac{1}{n}\sum_{m=0}^{n-1} \xi_n^{-mk} f_j. \tag{6.12}$$

We note the close connection between the continuous and discrete settings, where an integral is replaced by a sum.

[2] For a more detailed explanation see Olver and Shakiban [47].

6.1 Fast Fourier Transform

Computing the Fourier coefficients, $c_0, ..., c_{n-1}$ using the DFT from the definition can be very slow for large values of n. Computing the Fourier coefficients $c_0, ...c_{n-1}$ requires $n^2 - n$ complex multiplications and $n^2 - n$ complex additions. In 1960, Cooley and Tukey [12] rediscovered a much more efficient way of computing DFT by developing an algorithm known as the Fast Fourier Transforms (FFT) – the method was known to Gauss, but received little attention since he did not publish it [24]. The FFT cuts the number of arithmetic operations down to $O(n \log n)$. For large values of n, this can make a huge difference in computation time compared to the standard DFT. The reason why the FFT is so important is that it is heavily used in spectral methods. The basic FFT algorithm used by Cooley and Tukey [12] is well documented in many places, however, there are other implementations of the algorithm and the best version of the algorithm to use depends heavily on computer architecture. We therefore do not give further descriptions here.

Chapter 7

Finding Derivatives using Fourier Spectral Methods

Spectral methods are a class of numerical techniques that often utilize the FFT. Spectral methods can be implemented easily in Matlab, but there are some conventions to note. First note that Matlab's "fft" and "ifft" functions store wave numbers in a different order than has been used so far. The wave numbers in Matlab and in most other FFT packages are ordered, $0, 1, ..., \frac{n}{2}, -\frac{n}{2}+1, -\frac{n}{2}+2, ..., -1$. Secondly, Matlab does not take full advantage of real input data. The DFT of real data satisfies the symmetry property $\hat{v}(-k) = \hat{v}(k)$, so it is only necessary to compute half of the wave numbers. Matlab's "fft" command does not take full advantage of this property and wastes memory storing both the positive and negative wave numbers. Third, spectral accuracy (exponential decay of the magnitude of the Fourier coefficients) is better for smooth functions, so where possible be sure your initial conditions are smooth – **When using a Fourier spectral method this requires that your initial conditions are periodic**.

7.1 Taking a Derivative in Fourier Space

Let $u(x)$ be a function which is sampled at the n discrete points $x_i \in h, 2h, ..., ih, .., 2\pi - h, 2\pi$ and $h = 2\pi/n$ in real space. Now take the FFT

$$\mathrm{FFT}(u_j) \equiv \hat{u}_k \qquad \text{where} \quad k \in \frac{-n}{2}+1, ...\frac{n}{2}. \tag{7.1}$$

The Fourier transform of $\frac{\partial^2 u_j}{\partial x^2}$ can be easily computed from $\hat{u}_k$[1]:

$$\mathrm{FFT}(\frac{\partial^\nu u_j}{\partial x^\nu}) \equiv (ik)^\nu \hat{u}_k \qquad \text{where} \quad \hat{u}_{n/2} = 0 \quad \text{, if} \quad \nu \quad \text{is odd.} \tag{7.2}$$

Thus, differentiation in real space becomes multiplication in Fourier space. We can then take the inverse fast Fourier Transform (IFFT) to yield a solution in real space. In the

[1]More details can be found in Trefethen [56, Chap. 3]

next section we will use this technique to implement forward Euler and backward Euler timestepping schemes to compute solutions for several PDEs.

7.1.1 Exercises

1) Let $u(x) = \sum_k \hat{u}_k \exp(ikx)$ be the Fourier series representation of a function $u(x)$. Explain why

$$\frac{\mathrm{d}^\nu u}{\mathrm{d}x^\nu} = \sum (ik)^\nu \hat{u}_k,$$

provided the series converges.

2) [2] Consider the linear KdV equation

$$u_t + u_{xxx} = 0$$

with periodic boundary conditions for $x \in (0, 2\pi]$ and the initial data

$$u(x,0) = \begin{cases} 0 & \text{if } 0 < x \leq \pi \\ 1 & \text{if } \pi < x \leq 2\pi \end{cases}$$

a) Using separation of variables, show that the "solution" is

$$u(t,x) = \frac{1}{2} - \frac{2}{\pi} \sum_{j=0}^{\infty} \frac{\sin((2j+1)x - (2j+1)^3 t)}{2j+1}.$$

Quotation marks are used because the expression for the solution that is given does not converge when differentiated either once in time or twice in space.

b) As explained by Olver [46], this solution has a fractal structure at times that are an irrational multiple of π and a quantized structure at times that are rational multiples of π. The Matlab program in listing 7.1 uses the Fast Fourier transform to find a solution to the linearized KdV equation. Explain how this program finds a solution to the linearized KdV equation.

c) Compare the numerical solution produced by the Matlab program with the analytical solution. Try to determine which is more accurate and see if you can find evidence or an explanation to support your suggestions.

Listing 7.1: A Matlab program which solves the linearized KdV equation using the Fast Fourier transform.

```
% This program computes the solution to the linearly dispersive
% wave equation using the Fast Fourier Transform
```

[2]This question was prompted by an REU and UROP project due to Sudarshan Balakrishan which is available at `http://www.math.lsa.umich.edu/undergrad/REU/projects.html`.

```

N = 512;                                % Number of grid points.
h = 2*pi/N;                             % Size of each grid.
x = h*(1:N);                            % Variable x as an array.
t = .05*pi;                             % Time to plot solution at
dt = .001;                              % Appropriate time step.
u0 = zeros(1,N);                        % Array to hold initial data
u0(N/2+1:N)= ones(1,N/2);               % Defining the initial data
k=(1i*[0:N/2-1 0 -N/2+1:-1]);           % Fourier wavenumbers
k3=k.^3;
u=ifft(exp(k3*t).*fft(u0));             % Calculate the solution
plot(x,u,'r-');                         % Plot the solution
xlabel x; ylabel u;                     % Label the axes of the graphs
title(['Time ',num2str(t/(2*pi)),' \pi']);
```

Chapter 8

Examples in Matlab

We now want to find approximate numerical solutions using Fourier spectral methods. In this section we focus primarily on the heat equation with periodic boundary conditions for $x \in [0, 2\pi)$. Many of the techniques used here will also work for more complicated partial differential equations for which separation of variables cannot be used directly.

8.1 1D Heat Equation

The 1D heat equation

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \tag{8.1}$$

is a well known second order PDE for which exact series solutions can be found using separation of variables. It arises in several contexts such as in predicting the temperature in a thin uniform cross section rod. The equation and its derivation can be found in introductory books on partial differential equations and calculus, for example [6], [13] and [26], The constant α is the thermal diffusivity and $u(x, t)$ is temperature. We have already described how to solve the heat equation using separation of variables. Let us first discretize x such that x_j where $j = 0, 1, 2, ..., n$. x_j are uniformly spteaced in $[0, 2\pi)$. Let's now take the FFT of both sides of the 1D heat equation to obtain

$$\widehat{\frac{\partial u}{\partial t}} = \alpha \widehat{\frac{\partial^2 u}{\partial x^2}}. \tag{8.2}$$

We then rewrite the spatial derivative using eq. (7.2) [1]

$$\frac{\partial \hat{u}_k}{\partial t} = \alpha (ik)^2 \hat{u}_k, \tag{8.3}$$

so that the partial differential equation now becomes a collection of independent ODEs. While we can solve these ODEs in time exactly, we will use techniques that will also allow

[1]The k subscript denotes the coefficient of the kth Fourier mode.

us to obtain approximate solutions to PDEs we cannot solve exactly. We will discuss two methods for solving these ODEs, forward Euler and backward Euler.

8.1.1 Forward Euler

Using the forward Euler method in time, we obtain

$$\frac{\hat{u}_k^{n+1} - \hat{u}_k^n}{h} = \alpha(ik)^2\hat{u}_k^n \tag{8.4}$$

$$\hat{u}_k^{n+1} = \hat{u}_k^n + \alpha h(ik)^2\hat{u}_k^n \tag{8.5}$$

All that is left is to take the IFFT of the computed solution after all timesteps are taken to transfer it back to real space. This is a linear PDE, so only one IFFT is needed at the end. We will later see that this is different for a nonlinear PDE. A Matlab implementation of this is in listing 8.1.

Listing 8.1: A Matlab program to solve the heat equation using forward Euler timestepping.

```
%Solving Heat Equation using pseudo-spectral and Forward Euler
%u_t= \alpha*u_xx
%BC= u(0)=0, u(2*pi)=0
%IC=sin(x)
clear all; clc;

%Grid
N = 64;          %Number of steps
h = 2*pi/N;      %step size
x = h*(1:N);     %discretize x-direction

alpha = .5;       %Thermal Diffusivity constant
t = 0;
dt = .001;

%Initial conditions
v = sin(x);
k=(1i*[0:N/2-1 0 -N/2+1:-1]);
k2=k.^2;

%Setting up Plot
tmax = 5; tplot = .1;
plotgap= round(tplot/dt);
nplots = round(tmax/tplot);
data = [v; zeros(nplots,N)]; tdata = t;


for i = 1:nplots
    v_hat = fft(v);  %Fourier Space
    for n = 1:plotgap
        v_hat = v_hat+dt*alpha*k2.*v_hat; %FE timestepping
```

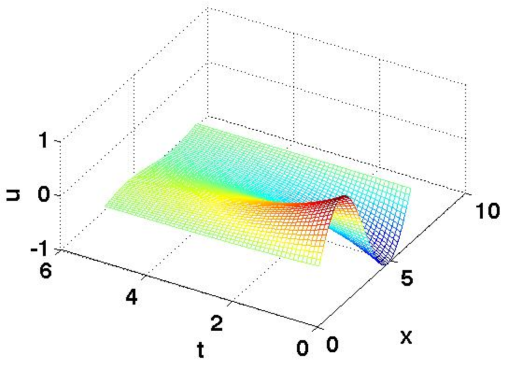

Figure 8.1: A numerical solution to the heat equation, eq. (8.1) computed using the backward Euler method.

```
    end
    v = real(ifft(v_hat)); %Back to real space
    data(i+1,:) = v;
    t=t+plotgap*dt;
    tdata = [tdata; t]; %Time vector
end

%Plot using mesh
mesh(x,tdata,data), grid on,
view(-60,55), xlabel x, ylabel t, zlabel u, zlabel u
```

8.1.2 Backward Euler

To derive this method, we start by applying the FFT and then perform timestepping using backward Euler. We then rewrite the implicit form into a form that gives the next iterate,

$$\begin{aligned}
\frac{\partial \hat{u}_k}{\partial t} &= \alpha(ik)^2\hat{u}_k && (8.6)\\
\frac{\hat{u}_k^{n+1}-\hat{u}_k^n}{h} &= \alpha(ik)^2\hat{u}_k^{n+1} && (8.7)\\
\hat{u}_k^{n+1}(1-\alpha h(ik)^2) &= \hat{u}_k^n && (8.8)\\
\hat{u}_k^{n+1} &= \frac{\hat{u}_k^n}{(1-\alpha h(ik)^2)}. && (8.9)
\end{aligned}$$

Below is a graph of the numerical solution to the heat equation[2] where $n = 64$ obtained using the Matlab program in listing 8.2.

[2]Methods to obtain the exact solution can be found in, among other places, Boyce and DiPrima [6].

Listing 8.2: A Matlab program to solve the heat equation using backward Euler timestepping.

```
%Solving Heat Equation using pseudospectral methods with Backwards Euler:
%u_t= \alpha*u_xx
%BC = u(0)=0 and u(2*pi)=0 (Periodic)
%IC=sin(x)
clear all; clc;

%Grid
N = 64; h = 2*pi/N; x = h*(1:N);

% Initial conditions
v = sin(x);
alpha = .5;
t = 0;
dt = .001; %Timestep size

%(ik)^2 Vector
k=(1i*[0:N/2-1 0 -N/2+1:-1]);
k2=k.^2;

%Setting up Plot
tmax = 5; tplot = .1;
plotgap= round(tplot/dt);
nplots = round(tmax/tplot);
data = [v; zeros(nplots,N)]; tdata = t;


for i = 1:nplots
    v_hat = fft(v); %Converts to fourier space
    for n = 1:plotgap
        v_hat = v_hat./(1-dt*alpha*k2); %Backwards Euler timestepping
    end
    v = ifft(v_hat); %Converts back to real Space
    data(i+1,:) = real(v); %Records data
    t=t+plotgap*dt; %Records time
    tdata = [tdata; t];
end

%Plot using mesh
mesh(x,tdata,data), grid on, %axis([-1 2*pi 0 tmax -1 1]),
view(-60,55), xlabel x, ylabel t, zlabel u, zlabel u,
```

8.1.3 Exercises

1) Write a program to solve the heat equation using the Crank-Nicolson method.

2) Solve the advection equation $u_t = u_x$ for $x \in [0, 2\pi)$ with the initial data

   a) $u(t = 0, x) = \cos(x)$

b) $u(t=0,x) = \begin{cases} 0 & x<\pi \\ 1 & x\geq\pi \end{cases}$

up to a time $T = 1$. You can do this either by using separation of variables or by assuming that the solution is of the form $u(x,t) = f(x+t)$ and deducing what f is in order to satisfy the initial conditions. In both cases please use the forward Euler, backward Euler and Crank-Nicolson timestepping schemes. After calculating the exact solution in each of these cases, examine how the maximum error at the final time depends on the timestep for each of these three methods.

8.2 Nonlinear Equations

8.2.1 The 1D Allen-Cahn Equation

So far we have dealt only with linear equations. Now it's time for a nonlinear PDE. The *Allen-Cahn equation* models the separation of phases in a material. It was introduced by Sam Allen and J. W. Cahn [1] and is

$$\frac{\partial u}{\partial t} = \epsilon\frac{\partial^2 u}{\partial x^2} + u - u^3, \tag{8.10}$$

where ϵ is a small but positive constant. The way to numerically solve this is similar to the method used for the heat equation, but there are some notable differences. The biggest difference is that FFT(u^3)≠FFT(u)^3, so the u^3 must be computed before taking the FFT. The FFT is a linear operation but cubing is non-linear operation, so the order matters

$$\frac{\partial \hat{u}_k}{\partial t} = \epsilon\frac{\partial^2 \hat{u}_k}{\partial x^2} + \hat{u}_k - \widehat{u^3}_k. \tag{8.11}$$

Next rewrite the first term on the right hand side, just like we did in the heat equation

$$\frac{\partial \hat{u}_k}{\partial t} = \epsilon(ik)^2\hat{u}_k + \hat{u}_k - \widehat{u^3}_k. \tag{8.12}$$

In order to solve this numerically we are going to use a combination of implicit (backward Euler) and explicit (forward Euler) methods. We are going to skip forward Euler because it is unstable.

Implicit-Explicit Method

You might have already noticed that backward Euler is not going to work for the Allen-Cahn in its present state because of the nonlinear term. If you go to implement backward Euler you can see that you can't factor out all of the $\hat{u}_k^{n+1}$. Luckily there is a simple intuitive way around this that isn't detrimental to the accuracy of the solution. Write all the terms implicitly (backwards Euler) except for the nonlinear term which is expressed explicitly. Applying this to Allen-Cahn we find that [3]

$$\frac{\hat{u}_k^{n+1} - \hat{u}_k^n}{h} = \epsilon(ik)^2\hat{u}_k^{n+1} + \hat{u}_k^n - \widehat{(u^n)^3}_k \tag{8.13}$$

$$\hat{u}_k^{n+1}\left(-\epsilon(ik)^2 + \frac{1}{h}\right) = \frac{1}{h}\hat{u}_k^n + \hat{u}_k^n - \widehat{(u^n)^3}_k \tag{8.14}$$

$$\hat{u}_k^{n+1} = \frac{\hat{u}_k^n(\frac{1}{h}+1) - \widehat{(u^n)^3}_k}{\left(-\epsilon(ik)^2 + \frac{1}{h}\right)}. \tag{8.15}$$

Now we have a form that we can work with. We can set the initial conditions to be $u(x,0) = \frac{1}{4}\sin(x)$ and plot the computed space-time evolution calculated by the Matlab code in listing 8.3. The computed result is in Fig. 8.2.

Listing 8.3: A Matlab program to solve the 1D Allen-Cahn equation using implicit explicit timestepping.

```
%Solving 1D Allen-Cahn Eq using pseudo-spectral and Implicit/Explicit
   method
%u_t=u_{xx} + u - u^3
%where u-u^3 is treated explicitly and u_{xx} is treated implicitly
%BC = u(0)=0, u(2*pi)=0 (Periodic)
%IC=.25*sin(x);
clear all; clc;

%Grid and Initial Data
N = 8000; h = 2*pi/N; x = h*(1:N); t = 0;

dt = .001; %timestep size
epsilon= .001;

%initial conditions
v = .25*sin(x);

%(ik) and (ik)^2 vectors
k=(1i*[0:N/2-1 0 -N/2+1:-1]);
k2=k.^2;

%setting up plot
tmax = 5; tplot = .2;
plotgap= round(tplot/dt);
nplots = round(tmax/tplot);
data = [v; zeros(nplots,N)]; tdata = t;
```

[3]Notice that when programming you are going to have to update the nonlinear term (u^3) each time you want to calculate the next timestep $n+1$. The reason this is worth mentioning is because for each timestep you are going to have to go from real space to Fourier space to real space, then repeat. For, the heat equation you can perform any number of timesteps in Fourier space and only convert back when you record your data.

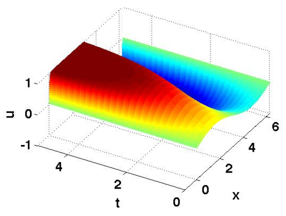

Figure 8.2: A numerical solution to the 1D Allen-Cahn equation, eq. (8.10), with $\epsilon = 0.001$ and $u(x, t = 0) = 0.25 \sin(x)$ computed using an implicit explicit method.

```

for i = 1:nplots
    for n = 1:plotgap
        v_hat = fft(v); %converts to Fourier space
        vv = v.^3;      %computes nonlinear term in real space
        vv = fft(vv);   %converts nonlinear term to Fourier space
        v_hat = (v_hat*(1/dt+1) - vv)./(1/dt-k2*epsilon); %Implicit/
           Explicit
        v = ifft(v_hat); %Solution back to real space
    end
    data(i+1,:) = real(v); %Records data each "plotgap"
    t=t+plotgap*dt; %Real time
    tdata = [tdata; t];
end

mesh(x,tdata,data), grid on, axis([-1 2*pi 0 tmax -1 1]),
view(-60,55), xlabel x, ylabel t, zlabel u
```

8.2.2 The 2D Allen-Cahn Equation

Now we will look at the 2D form of the Allen-Cahn Equation, where $u(x, y, t)$ satisfies

$$\frac{\partial u}{\partial t} = \epsilon \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) + u - u^3. \tag{8.16}$$

The convert it into Fourier space by taking the FFT of both sides

$$\frac{\partial \hat{u}_k}{\partial t} = \epsilon \left(\frac{\partial^2 \hat{u}_k}{\partial x^2} + \frac{\partial^2 \hat{u}_k}{\partial y^2} \right) + \hat{u}_k - \widehat{u^3}_k \tag{8.17}$$

$$\frac{\partial \hat{u}_k}{\partial t} = \epsilon \left((ik_x)^2 \hat{u}_k + (ik_y)^2 \hat{u}_k \right) + \hat{u}_k - \widehat{(u^3)}_k \tag{8.18}$$

where k_x and k_y is to remind us that we take the FFT in respected directions. We will also define

$$f(u) \equiv u - u^3 \tag{8.19}$$

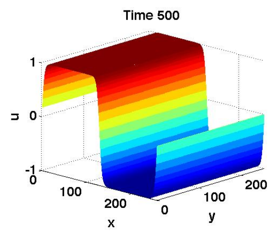

Figure 8.3: A numerical solution to the 2D Allen-Cahn equation, eq. (8.16) at time $t = 500$ with $\epsilon = 0.1$ and $u(x, y, t = 0) = \sin(2\pi x) + 0.001\cos(16\pi x)$ computed using an implicit explicit method.

The way to deal with the first two terms on the right hand side is to take the FFT in the x-direction and then take it in the y-direction. The order in which the FFT is done, x first or y first is not important. Some software libraries offer a two dimensional FFT. It usually depends on the equation being solved whether it is more efficient to use a multidimensional FFT or many one dimensional FFTs. Typically, it is easier to write a program which uses a multidimensional FFT, but in some situations this is not very efficient since one can immediately reuse data that has just been Fourier transformed.

Implicit-Explicit Method

In this method, the nonlinear term in eq. (8.19) is calculated explicitly, while the rest of the terms will be written implicitly such that

$$\frac{\hat{u}_k^{n+1} - \hat{u}_k^n}{h} = \epsilon\left((ik_x)^2\hat{u}_k^{n+1} + (ik_y)^2\hat{u}_k^{n+1}\right) + \widehat{f(u^n)}_k \tag{8.20}$$

$$\hat{u}_k^{n+1}\left(-\epsilon(ik_x)^2 - \epsilon(ik_y)^2 + \frac{1}{h}\right) = \frac{\hat{u}_k^n}{h} + \widehat{f(u^n)}_k \tag{8.21}$$

$$\hat{u}_k^{n+1} = \frac{\frac{\hat{u}_k^n}{h} + \widehat{f(u^n)}_k}{\left(-\epsilon(ik_x)^2 - \epsilon(ik_y)^2 + \frac{1}{h}\right)} \tag{8.22}$$

we can then substitute in for $f(u)$

$$\hat{u}_k^{n+1} = \frac{\hat{u}_k^n\left(\frac{1}{h} + 1\right) - \widehat{(u^n)^3}_k}{\left(-\epsilon(ik_x)^2 - \epsilon(ik_y)^2 + \frac{1}{h}\right)}. \tag{8.23}$$

The Matlab code used to generate Fig. 8.3 is in listing 8.4.

Listing 8.4: A Matlab program to solve the 2D Allen-Cahn equation using implicit explicit timestepping.

```
%Solving 2D Allen-Cahn Eq using pseudo-spectral with Implicit/Explicit
```

```
%u_t= epsilon(u_{xx}+u_{yy}) + u - u^3
%where u-u^3 is treated explicitly and epsilon(u_{xx} + u_{yy}) is treated
    implicitly
%BC = Periodic
%IC=v=sin(2*pi*x)+0.001*cos(16*pi*x;
clear all; clc;

%Grid
N = 256; h = 1/N; x = h*(1:N);
dt = .01;

%x and y meshgrid
y=x';
[xx,yy]=meshgrid(x,y);

%initial conditions
v=sin(2*pi*xx)+0.001*cos(16*pi*xx);
epsilon=.01;

%(ik) and (ik)^2 vectors in x and y direction
kx=(1i*[0:N/2-1 0 -N/2+1:-1]);
ky=(1i*[0:N/2-1 0 -N/2+1:-1]');
k2x=kx.^2;
k2y=ky.^2;

[kxx,kyy]=meshgrid(k2x,k2y);

for n = 1:500
    v_nl=v.^3;  %calculates nonlinear term in real space
    %FFT for linear and nonlinear term
    v_nl = fft2(v_nl);
    v_hat=fft2(v);
    vnew=(v_hat*(1+1/dt)-v_nl)./ ...
       (-(kxx+kyy)*epsilon+1/dt); %Implicit/Explicit timestepping
    %converts to real space in x-direction
    v=ifft2(vnew);
    %Plots each timestep
    mesh(v); title(['Time ',num2str(n)]); axis([0 N 0 N -1 1]);
    xlabel x; ylabel y; zlabel u;
    view(43,22); drawnow;
end
```

8.2.3 Exercises

Many of these exercises are taken from Uecker [59]. Another introductory source of information on these equations is Trefethen and Embree [57].

1) Burgers equation is given by:

$$\frac{\partial u}{\partial t} = \nu\frac{\partial^2 u}{\partial x^2} - u\frac{\partial u}{\partial x}$$

where $\nu \in \mathbb{R}^+$ and u has periodic boundary conditions. Solve this equation using an implicit-explicit method. If you take ν to be small, ensure that a sufficient number of grid points are used to get the correct numerical solution. A simple way to check this is to keep increasing the number of grid points and checking that there is no change in the solution. Another way to check this is to calculate the Fourier coefficients and check that the highest ones decay to machine precision.

2) The Kuramoto-Sivashinsky equation is given by:

$$\frac{\partial u}{\partial t} = -\frac{\partial^2 u}{\partial x^2} - \frac{\partial^4 u}{\partial x^4} - u\frac{\partial u}{\partial x}$$

where u has periodic boundary conditions.

   a) What does this equation model and what type of behavior do you expect its solutions to have?

   b) Find numerical solutions to this equation using an implicit-explicit method.

3) The 1D Gray-Scott equations are given by:

$$\frac{\partial u}{\partial t} = d_1\frac{\partial^2 u}{\partial x^2} - uv^2 + f(1-u),$$

$$\frac{\partial v}{\partial t} = d_2\frac{\partial^2 v}{\partial x^2} + uv^2 - (f+k)v$$

where d_1, d_2, f and k are constants.

   a) What does this equation model and what type of behavior do you expect its solutions to have?

   b) Find numerical solutions to this equation using an implicit-explicit method. Try several different values of d_1, d_2, f and k and compare the resulting patterns to what you can find in the literature.

4) The 2D Swift-Hohenberg equation is given by:

$$\frac{\partial u}{\partial t} = -\Delta^2 u + 2\Delta u + (\alpha - 1)u - u^3,$$

   a) What does this equation model and what type of behavior do you expect its solutions to have?

   b) Find numerical solutions to this equation using an implicit-explicit method for several values of α.

5) The 2D Gray-Scott equations are given by:

$$\frac{\partial u}{\partial t} = d_1\Delta u - uv^2 + f(1-u)$$

$$\frac{\partial v}{\partial t} = d_2 \Delta v + uv^2 - (f+k)v$$

where d_1, d_2, f and k are constants.

a) What does this equation model and what type of behavior do you expect its solutions to have?

b) Find numerical solutions to this equation using an implicit-explicit method.

6) The 2D Complex Ginzburg-Landau equation is given by:

$$\frac{\partial A}{\partial t} = A + (1+i\alpha)\Delta A - (1+i\beta)|A|^2 A.$$

An introductory tutorial to this equation can be found at `http://codeinthehole.com/static/tutorial/index.html`

a) What does this equation model and what type of behavior do you expect its solutions to have?

b) Find numerical solutions to this equation using an implicit-explicit method for several values of α and β.

Chapter 9

Nonlinear Ordinary Differential Equations and Iteration

The implicit explicit method avoids the direct solution of nonlinear problems. This can be advantageous for some problems, but can also lead to severe time step restrictions in others. Furthermore, the resulting numerical schemes can sometimes have undesirable qualitative properties. For this reason, we need to describe methods that allow us to solve the nonlinear equations generated in fully-implicit numerical schemes.

We consider an ordinary differential equation

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y) \tag{9.1}$$

for $t \in [t_0, t^*]$, and for which $f(t, y)$ is not necessarily a linear function of y. We want to use an implicit numerical method to obtain an approximate solution of this problem – for example backward Euler's method. If we want to demonstrate the convergence of the numerical scheme, we need to demonstrate convergence of functional iteration which we use to find the solution for the nonlinear equation term in using backward Euler's method.

The results that follow are primarily taken from Iserles [29], although this material is also often found in calculus texts such as Lax, Burstein and Lax [37], and Hughes et al. [26]. We will let t_i denote the time at time step i, y_i denote the approximate solution at time step i and h denote the time step. We will assume f is Lipschitz continuous, a condition that is weaker than differentiable but stronger than continuous, which we will give a precise definition of. There are two classical iteration methods:

- fixed-point iteration
- Newton's (Newton-Raphson) method.

We will prove convergence of these two methods (a proof of the convergence of the modified Newton-Raphson method is in Iserles [29, p. 130]). We will analyze the specific problem $y'(t) = y^2$ with initial data $y(0) = 1$ and $t \in [0, 0.99]$.

9.1 Exact Solution to an Example Nonlinear Ordinary Differential Equation

We consider

$$\frac{\mathrm{d}y}{\mathrm{d}t} = y^2 \tag{9.2}$$

with initial data $y(t=0)=1$ and $t \in [0, 0.99]$. Whenever the solution $y(t)$ exists, it will be positive all the time, because the initial value is positive and $\frac{\mathrm{d}y}{\mathrm{d}t}$ is positive.

To integrate this equation explicitly, we use separation of variables to find that

$$\int_{y(0)}^{y(t)} \frac{1}{\tilde{y}^2}\mathrm{d}\tilde{y} = \int_0^t \mathrm{d}\tau \tag{9.3}$$

which implies

$$-\frac{1}{y(t)} = t + c \tag{9.4}$$

where c is the constant of integration. Using our initial data we get $c = -1$, so

$$y(t) = \frac{1}{1-t} \tag{9.5}$$

is our exact solution for this problem. We will use this exact solution to compare the numerical solutions obtained by the different iterative methods. Notice that this exact solution becomes infinite as $t \to 1$.

9.2 Definitions Required to Prove Convergence

Definition 9.2.1. The Lipschitz Condition *A function $f(x) : x \in D \subset \mathbb{R}$ is Lipschitz if $\|f(x_1) - f(x_2)\| \leq \lambda \|x_1 - x_2\|$ for all x_1 and x_2 in the domain D.*

There are two specific definitions of the Lipschitz condition.

Definition 9.2.2. Locally Lipschitz Condition *The function $f(x)$ is called locally Lipschitz if, for each $z \in \mathbb{R}$, there exists an $L > 0$ such that f is Lipschitz on the open ball of center z and radius L.*

Definition 9.2.3. Globally Lipschitz Condition *If $f(x)$ is Lipschitz on all of the space $\mathbb{R}$ (i.e. The open ball is $\mathbb{R}$ in above definition), then f is globally Lipschitz.*

Note the fundamental difference between the local and global versions of the Lipschitz-condition. Whereas in the local version the Lipschitz "constant" (λ) and the open ball depend on each point $x \in \mathbb{R}$, in the global version the "constant" (λ) is fixed and the open ball is $\mathbb{R}$. In particular, a globally Lipschitz function is locally Lipschitz continuous, but the converse is not true.

9.3 Existence and Uniqueness of Solutions to Ordinary Differential Equations

Peano's theorem states that if $f(x)$ is continuous, then a solution to the ordinary differential equation $x'(t) = f(x)$ with initial condition $x(t_0) = x_0$ exists at least in some neighbourhood of time t_0 – this solution need not be unique. Picard's theorem states that if $f(x)$ is locally Lipschitz, then the solution for the ordinary differential equation $x'(t) = f(x)$ with initial condition $x(t_0) = x_0$ is unique when it exists. A comprehensive statement of these theorems is in Iserles [29, p. 445], and there are proofs of these theorems in many books on ordinary differential equations (for example Birkhoff and Rota [2, Chap. 6, pg. 192]).

9.4 Backward Euler

We recall that the backward Euler method is given by

$$y^{n+1} = y^n + hf(y^{n+1}). \tag{9.6}$$

Note that if f is nonlinear, we need to solve a nonlinear equation in each step advancing the solution (numerical). It is usually hard to solve a nonlinear equation exactly using analytical methods, so we also use numerical methods. For our example equation, we get

$$y^{n+1} = y^n + h\left(y^{n+1}\right)^2 \tag{9.7}$$

This example has the advantage that we can find its solutions algebraically, so we can then examine the behavior of numerical schemes.

9.5 Convergence of Functional Iteration

We often use functional iteration to solve nonlinear equations. We recall that there are two popular methods: fixed-point iteration and Newton's method.

9.5.1 Convergence of the Fixed-Point Method

We want to find a root of $x = f(x)$. We try to use the fixed-point method and to construct a sequence $x_{n+1} = f(x_n)$ where $n = 0, 1, 2 \ldots$.

Theorem 9.5.1. *Let $f(x)$ have a fixed-point $\tilde{x} = f(\tilde{x})$, be Lipschitz continuous for $x \in (a, b) \subset \mathbb{R}$ with Lipschitz constant $k < 1$ and $f(x)$ be continuous on $[a, b]$. Then the fixed point method $x_{n+1} = f(x_n)$ converges to the unique fixed-point of $\tilde{x} = x_\infty = f(x_\infty)$ for $x \in [a, b]$.*

Proof. Since $f(x)$ is Lipschitz continuous, we find that,

$$|x_{n+1} - x_\infty| = |f(x_n) - f(x_\infty)| \leq k\,|x_n - x_\infty| \tag{9.8}$$

for $n = 1, 2 \ldots$. Hence by induction we conclude that

$$|x_{n+1} - x_\infty| \leq k^n\,|x_1 - x_\infty|\,. \tag{9.9}$$

Since $k < 1$, $\lim_{n\to\infty} k^n|x_1 - x_\infty| = 0$, so we obtain a solution $x_\infty = f(x_\infty)$, where x_∞ is the fixed point. We can show that the limit is unique by supposing that there are two different limits and reaching a contradiction. □

For a proof of the existence of the fixed-point under the assumptions used in this theorem, see a book on numerical analysis, such as Bradie [4] or Iserles [29].

Regarding our problem, we apply fixed-point iteration, we want to find the root of an equation of the form:

$$w = hw^2 + \beta = f(w). \tag{9.10}$$

When the timestep h is small enough then $f'(w) = 2hw \leq 200h < 1$. So fixed-point iteration is convergent provided the time-step is small enough. We note that eq. (9.10) has two roots, and so the domain of the initial iterate plays an important role in determining which root is choosen.

9.5.2 Convergence of Newton's Method

We now consider Newton's method. We want to find a root, x^* of $f(x)$ such that $f(x^*) = 0$. Newton's method is a fixed-point method where the iterates are constructed by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{9.11}$$

where $n = 0, 1, 2 \ldots$. If the function $f(x)$ is sufficiently well behaved, then Newton's method has a quadratic rate of convergence.

Theorem 9.5.2. *Suppose $f(x)$ is twice continuously differentiable and that its second derivative is bounded. Suppose also that there exists x^* for which $f(x^*) = 0$. Suppose $f'(x) \neq 0$ in the interval $[x^* - |x^* - x_0|\,, x^* + |x^* - x_0|]$, $f''(x)$ is finite in the same interval and $|x_0 - x^*|$ is small. Then, Newton's method is of quadratic convergence.*

Proof.

$$f(x^*) = f(x_n) + f'(x_n)(x^* - x_n) + \frac{1}{2!}f''(z_n)(x^* - x_n)^2 \tag{9.12}$$

by Taylor expansion with Lagrange form remainder. In the above $z_n \in [x_n, x^*]$. Since $f(x^*) = 0$, we have

$$0 = f(x_n) + f'(x_n)(x^* - x_n) + \frac{1}{2!}f''(z_n)(x^* - x_n)^2, \tag{9.13}$$

so

$$\frac{f(x_n)}{f'(x_n)} + (x^* - x_n) = -\frac{1}{2!}\frac{f''(z_n)}{f'(z_n)}(x^* - x_n)^2. \tag{9.14}$$

Plug in the formula for x_{n+1}, from eq. (9.11) we have

$$x^* - x_{n+1} = -\frac{1}{2!}\frac{f''(z_n)}{f'(z_n)}(x^* - x_n)^2. \tag{9.15}$$

Let

$$e_n = |x^* - x_n|. \tag{9.16}$$

We have

$$e_{n+1} = \left|\frac{1}{2!}\frac{f''(z_n)}{f'(z_n)}\right| e_n^2 \tag{9.17}$$

and by our assumption, we know there is a constant c such that

$$\left|\frac{1}{2!}\frac{f''(z_n)}{f'(z_n)}\right| < c. \tag{9.18}$$

Hence we have $e_{n+1} < me_n^2$ for some finite constant m. So Newton's method is convergent provided $e_0 = |x_0 - x^*|$ is sufficiently small. $\square$

Regarding our problem, we consider

$$f(y) = y - hy^2 - \beta. \tag{9.19}$$

Hence $f'(y) = 1 - 2hy \neq 0$ and $f"(y)$ is finite, so our problem satisfies all assumptions if we choose our initial data and initial iterates suitably. Hence the Newton iterations will converge and give an approximation to the nonlinear term in backward Euler's method.

9.6 Convergence of the Theta Method

The backward Euler, forward Euler and Crank-Nicolson methods are special case of the theta method, so we will first prove the convergence of the theta method to encompass these three methods. The theta method is the following algorithm,

$$y^{n+1} = y^n + h[\theta f(t^n, y^n) + (1-\theta) f(t^{n+1}, y^{n+1})] \tag{9.20}$$

where $n = 0, 1, \ldots$ and $\theta \in [0, 1]$. Notice that for $\theta = 1/2$ we obtain the Crank-Nicolson method or trapezoidal rule.

First, substituting the exact solution $y(t)$ and using the Taylor expansion we have

$$\begin{aligned}&y(t^{n+1}) - y(t^n) - h[\theta f(t^n, y(t^n)) + (1-\theta) f(t^{n+1}, y(t^{n+1}))] \\ &= y(t^{n+1}) - y(t^n) - h[\theta y'(t^n) + (1-\theta) y'(t^{n+1})] \\ &= [y(t^n) + hy'(t^n) + \frac{1}{2}h^2 y''(t^n) + \frac{1}{6}h^3 y'''(t^n)] \\ &\quad - y(t^n) - h\{\theta y'(t^n) + (1-\theta)[y'(t^n) + hy''(t^n) + \frac{1}{2}h^2 y'''(t^n)]\} + \mathcal{O}(h^4) \\ &= \left(\theta - \frac{1}{2}\right) h^2 y''(t^n) + \left(\frac{1}{2}\theta - \frac{1}{3}\right) h^3 y'''(t^n) + \mathcal{O}(h^4).\end{aligned} \tag{9.21}$$

Subtracting the last expression from

$$y^{n+1} - y^n - h[\theta f(t^n, y^n) + (1-\theta) f(t^{n+1}, y^{n+1})] = 0, \tag{9.22}$$

we have that when h is small enough

$$\begin{aligned}&e^{n+1,h} \\ &= e^{n,h} + \theta h[f(t^n, y(t^n) + e^{n,h}) - f(t^n, y(t^n))] \\ &\quad + (1-\theta)h[f(t^{n+1}, y(t^{n+1}) + e^{n+1,h}) - f(t^{n+1}, y(t^{n+1}))] \\ &\begin{cases} -\frac{1}{12}h^3 y'''(t^n) + \mathcal{O}(h^4), & \theta = \frac{1}{2} \\ +(\theta - \frac{1}{2})h^2 y''(t^n) + \mathcal{O}(h^3), & \theta \neq \frac{1}{2} \end{cases}\end{aligned} \tag{9.23}$$

where $e^i = y^i - y(t^i)$. Using the triangle inequality and by the Lipschitz continuity of f, there exist constants c and λ such that

$$\begin{aligned}&\left\|e^{n+1,h}\right\| \\ &\leq \left\|e^{n,h}\right\| + \theta h\lambda \left\|e^{n,h}\right\| + (1-\theta)h\lambda \left\|e^{n+1,h}\right\| + \begin{cases} ch^3 & \theta = \frac{1}{2} \\ ch^2 & \theta \neq \frac{1}{2} \end{cases}.\end{aligned} \tag{9.24}$$

When $\theta = \frac{1}{2}$, the theta method reduces to the trapezoidal rule. It is possible to show that the Crank-Nicolson method has second order convergence, see for example, Iserles [29]. Now let's consider $\theta \neq \frac{1}{2}$,

$$\left\|e^{n+1,h}\right\| \leq \frac{1 + \theta h\lambda}{1 - (1-\theta)h\lambda} \left\|e^{n,h}\right\| + \frac{c}{1 - (1-\theta)h\lambda} h^2. \tag{9.25}$$

We claim that

$$\left\|e^{n,h}\right\| \leq \frac{c}{\lambda}\left[\left(\frac{1 + \theta h\lambda}{1 - (1-\theta)h\lambda}\right)^n - 1\right] h \tag{9.26}$$

We prove this statement by induction. When $n = 0$, $\left\|e^{n,h}\right\| = 0$, since the initial conditions is exactly calculated. Now suppose this statement is true for $n = k$, where $k \geq 0$ and is a integer. We want to show this statement is true for $n = k + 1$. Consider

$$\left\|e^{k+1,h}\right\| \leq \frac{1 + \theta h\lambda}{1 - (1-\theta)h\lambda} \left\|e^{k,h}\right\| + \frac{c}{1 - (1-\theta)h\lambda} h^2, \tag{9.27}$$

then plug in

$$\left\|e^{kn,h}\right\| \leq \frac{c}{\lambda}\left[\left(\frac{1+\theta h\lambda}{1-(1-\theta)h\lambda}\right)^k - 1\right] h. \tag{9.28}$$

We have

$$\begin{aligned}
&\left\|e^{k+1,h}\right\| \\
&\leq \frac{c}{\lambda}\left[\left(\frac{1+\theta h\lambda}{1-(1-\theta)h\lambda}\right)^{k+1} - \frac{1+\theta h\lambda}{1-(1-\theta)h\lambda}\right] h + \frac{c}{1-(1-\theta)h\lambda}h^2 \\
&= \frac{c}{\lambda}\left[\left(\frac{1+\theta h\lambda}{1-(1-\theta)h\lambda}\right)^{k+1} - 1\right] h.
\end{aligned} \tag{9.29}$$

So our claim is true for all n . Note that

$$\begin{aligned}
\frac{1+\theta h\lambda}{1-(1-\theta)h\lambda} &= 1 + \frac{h\lambda}{1-(1-\theta)h\lambda} \\
&\leq \exp\left(\frac{h\lambda}{1-(1-\theta)h\lambda}\right)
\end{aligned} \tag{9.30}$$

by a Taylor expansion of the exponential function. Thus, we have

$$\begin{aligned}
\left\|e^{n,h}\right\| &\leq \frac{c}{\lambda}\left[\left(\frac{1+\theta h\lambda}{1-(1-\theta)h\lambda}\right)^n - 1\right] h \\
&\leq \frac{c}{\lambda}\left(\frac{1+\theta h\lambda}{1-(1-\theta)h\lambda}\right)^n h \\
&\leq \frac{ch}{\lambda}\exp\left(\frac{nh\lambda}{1-(1-\theta)h\lambda}\right).
\end{aligned} \tag{9.31}$$

By our condition, $nh \leq t^*$. Therefore

$$\left\|e^{n,h}\right\| \leq \frac{ch}{\lambda}\exp\left(\frac{t^*\lambda}{1-(1-\theta)h\lambda}\right). \tag{9.32}$$

So we have $\lim_{h\to 0}\left\|e^{n,h}\right\| = 0$ and $0 \leq nh \leq t^*$. Hence the theta method is convergent of order 1 when $\theta \neq \frac{1}{2}$.

Note that the backward Euler method is a special case of the theta method when $\theta = 0$, so backward Euler's method is convergent of order 1. We arrive at our theorem.

Theorem 9.6.1. *Backward Euler's method is convergent of order 1.*

Remark 9.6.1. *If f is globally Lipschitz, then we can apply the above argument with respect to any time interval. If f is only locally Lipschitz, then we need to analyze the situation more carefully. First, by Picard's theorem, there is a unique solution of this ordinary differential equation for a short amount of time. Indeed, we just need to know that the Lipschitz constant is finite without necessarily needing to know the exact value.*

Remark 9.6.2. *If one did not know of Picard's theorem, one could deduce the existence and uniqueness of solutions to ODEs by using time discretization.*

Now we consider $y' = y^2$ and $t \in [0, 0.99]$. The exact solution of this problem is $y(t) = \frac{1}{1-t}$. So $1 \leq y \leq 100$. In our problem, $f = y^2$ is clearly analytic and it is locally Lipschitz. It is easy to show f is not globally Lipschitz. If a function $f(x)$ is globally Lipschitz condition then there is a finite constant λ such that

$$\frac{\|f(x) - f(y)\|}{\|x - y\|} \leq \lambda \tag{9.33}$$

for all $x, y \in \mathbb{R}$. In our problem, let $x = 0$ and $\|y\| \to \infty$, it is easy to check

$$\frac{\|f(x) - f(y)\|}{\|x - y\|} \to \infty. \tag{9.34}$$

We now discuss how one can find local Lipschitz constants λ. When f is differentiable, we often just differentiate f and find the maximum value of its derivative in the domain of interest. In our example, f is simple and we only need to know that the Lipschitz constant is finite. So we use a more rough method to show that the Lipshitz constant is finite,

$$\left\|f(y^1) - f(y^2)\right\| = \left\|y^1 + y^2\right\| \left\|y^1 - y^2\right\| \leq \left(\left\|y^1\right\| + \left\|y^2\right\|\right) \left\|y^1 - y^2\right\|. \tag{9.35}$$

So it suffices to find the maximal value of $\|y\|$ in this problem. In our problem, $y(t)$ is continuous. Furthermore, $y(t)$ will be positive all the time, because the initial value is positive and y' is positive. A continuous function has finite maximal value in a closed and bounded set. Note that the exact solution of our problem is $y(t) = \frac{1}{1-t}$, so $1 \leq y \leq 100$. So we know that the Lipschitz constant in our problem is finite.

Finally, we get the convergence of functional iteration and backward Euler's method of our problem. Thus our numerical scheme for $y' = y^2$ with initial data $y(0) = 1$ and $t \in [0, 0.99]$ is convergent.

Corollary 9.6.1. *By the theorems for existence and uniqueness of the solution for ordinary differential equations and Theorem 4.1 ,Theorem 4.2 and Theorem 4.3, we arrive at our final goal that the numerical solution generated by backward Euler's method with functional iteration exists and is unique when the time-step, $h0$ approaches zero.*

Remark 9.6.3. *This requires careful choice of initial iterates when doing functional iteration.*

Remark 9.6.4. *Typically, the exact solution of an ODE is not known, although it is possible to deduce local Lipschitz continuity. Should the solution become infinite, a numerical method will either not converge or display very large values if the approximate solution closely approximates the exact solution. Some care is required in interpreting such numerical simulations in these cases.*

9.7 Example Programs which use Iteration to Solve a Nonlinear Ordinary Differential Equation

The following two Matlab programs demonstrate backward Euler's method for the example equation. The first one uses fixed-point iteration to solve for the nonlinear term and the second one uses Newton's method to solve for the nonlinear term.

Listing 9.1: A Matlab program to demonstrate fixed-point iteration.

```
% A program to solve y'=y^2 using the backward Euler
% method and fixed point iteration
% This is not optimized and is very simple

clear all; format compact; format short;
set(0,'defaultaxesfontsize',25,'defaultaxeslinewidth',.7,...
'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,...
'defaultaxesfontweight','bold')

n=10000; % Number of timesteps
Tmax=0.99; % Maximum time
y0=1; % Initial value
tol=0.1^10; % Tolerance for fixed point iterations
dt=Tmax/n; % Time step
y=zeros(1,n); % vector for discrete solution
t=zeros(1,n); % vector for times of discrete solution
y(1)=y0;
t(1)=0;
tic,         % start timing
for i=1:n
    yold=y(i); ynew=y(i); err=1;
    while err>tol
        ynew=dt*yold^2+y(i);
        err=abs(ynew-yold);
        yold=ynew;
    end
    y(i+1)=ynew;
    t(i+1)=t(i)+dt;
end
toc,         % stop timing
yexact=1./(1-t); max(abs(y-yexact)), % print the maximum error
figure(1); clf; plot(t,y,'r+',t,yexact,'b-.');
xlabel Time; ylabel Solution; legend('Backward Euler','Exact solution');
title('Numerical solution of dy/dt=y^2');
```

Listing 9.2: A Matlab program to demonstrate Newton iteration.

```
% A program to solve y'=y^2 using the backward Euler
% method and Newton iteration
% This is not optimized and is very simple

set(0,'defaultaxesfontsize',25,'defaultaxeslinewidth',.7,...
'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,...
    'defaultaxesfontweight','bold')

n=100000; % Number of timesteps
Tmax=0.99; % Maximum time
y0=1; % Initial value
tol=0.1^10; % Tolerance for fixed point iterations
dt=Tmax/n; % Time step
y=zeros(1,n); % vector for discrete solution
t=zeros(1,n); % vector for times of discrete solution
y(1)=y0;
t(1)=0;
tic,        % start timing
for i=1:n
    yold=y(i); ynew=y(i); err=1;
    while err>tol
        ynew=yold-(yold-y(i)-dt*yold^2)/(1-2*dt*yold);
        err=abs(ynew-yold);
        yold=ynew;
    end
    y(i+1)=ynew;
    t(i+1)=t(i)+dt;
end
toc,        % stop timing
yexact=1./(1-t); max(abs(y-yexact)), % print maximum error
figure(1); clf; plot(t,y,'r+',t,yexact,'b-.');
xlabel Time; ylabel Solution;
legend('Backward Euler','Exact solution');
title('Numerical solution of dy/dt=y^2');
```

9.8 Exercises

1) Run the fixed-point iteration program in Matlab and check that the outcome is reasonable. Now investigate how changing the number of time steps taken to go from a time of 0 to a time of 0.99, and the tolerance for fixed point iterations affects the maximum error. In particular try a range of 1,000-1,000,000 (in powers of 10) for the number of time steps and a tolerance ranging from $10^{-1} - 10^{-7}$ (in powers of 10^{-1}). You should observe that there is an "ideal" combination of subdivisions and tolerance to minimize the error. What are these combinations? Do this whole process again using Newton iteration instead. How have the answers changed?

2) Write a Matlab program to solve $y' = y^2$ with $y(0) = 1$ using the Crank-Nicolson method and fixed point iteration. Explain why there are two fixed-points to which the fixed-point iteration can converge. Which of these fixed-points gives the correct

approximation to the solution of the differential equation? Comment on how the choice of initial iterate for the fixed-point iteration determines the fixed-point to which the method converges.

3) a) Show that the differential equation $y' = \sqrt{|y|}$, with $y(0) = 0$ is not Lipschitz continuous.

   b) Find at least two analytical solutions to this differential equation.

   c) Compute a numerical solution to this differential equations using the forward Euler method.

   d) Compute a numerical solution to this differential equations using the backward Euler method. Be sure to try different initial guesses for the fixed-point iteration, not just the value at the previous time step; you should be able to calculate the influence of the choice of initial iterate on the selection of solution by the numerical method. Comment on this.

   e) Compute a numerical solution to this differential equations using the implicit midpoint rule. Be sure to try different initial guesses for the fixed point iteration, not just the value at the previous time step; you should be able to calculate the influence of the choice of initial iterate on the selection of "solution" by the numerical method. Comment on this.

   f) Repeat (d) and (e) with Newton iteration.

   g) Comment on the applicability of numerical methods for solving differential equations without unique solutions.

4) Modify the program for the 1-D Allen-Cahn equation so that it uses the Crank-Nicolson and fixed-point iteration for the nonlinear term. You will need to calculate the nonlinear term in real space, so that your resulting scheme is

$$\frac{\hat{u}^{n+1,k+1} - \hat{u}^n}{\delta t} = \frac{\hat{u}^{n+1,k+1}_{xx} + \hat{u}^n_{xx}}{2} + \frac{1}{2}\left[\widehat{u^{n+1,k} - \left(u^{n+1,k}\right)^3}\right] + \frac{1}{2}\left[\widehat{u^n - \left(u^n\right)^3}\right], \qquad (9.36)$$

where n denotes the time step and k denotes the iterate. Stop the iterations once the maximum difference between successive iterates is sufficiently small.

5) Modify the program for the 2-D Allen-Cahn equation so that it uses the Crank-Nicolson method and fixed-point iteration for the nonlinear term. You will need to calculate the nonlinear term in real space.

Chapter 10

Fortran Programs

10.1 Example Programs

To do parallel programming using OpenMP or MPI (Message passing interface), we typically need to use a lower level language than Matlab such as Fortran. Another possible choice of language is C, however Fortran has superior array handling capabilities compared to C, and has a similar syntax to Matlab, so is typically easier to use for scientific computations which make heavy use of regular arrays. It is therefore useful to introduce a few simple programs in Fortran before we begin studying how to create parallel programs. A good recent reference on Fortran is Metcalf, Reid and Cohen [44]. We recognize that most people will be unfamiliar with Fortran and probably more familiar with Matlab[1], C or C++, but we expect that the example codes will make it easy for anyone with some introductory programming background. A recent guide which describes how to write efficient parallel Fortran code is Levesque and Wagenbreth[41]. Our programs are written to be run on the Flux cluster at the University of Michigan. More information on this cluster can be found at `http://cac.engin.umich.edu/resources/systems/flux/` and at `http://cac.engin.umich.edu/started/index.html`. Below are four files you will need to run this.

1) A makefile to compile the Fortran code on Flux in listing 10.1. This should be saved as *makefile*. Before using the makefile to compile the code, you will need to type
   `module load fftw/3.2.1-intel`
   at the command line prompt once logged into Flux. Then place the makefile and heat.f90 in the same directory, the example files below assume this directory is
   `$HOME/ParallelMethods/Heat`
   and type
   `make`
   to compile the file. Once the file is compiled type
   `qsub fluxsubscript`
   to get the cluster to run your program and then output the results. The programs that

[1]Although Matlab is written in C, it was originally written in Fortran and so has a similar style to Fortran.

follow use the library FFTW to do the fast Fourier Transforms. More information on this library can be found at `http://www.fftw.org/`.

Listing 10.1: An example makefile for compiling a Fourier spectral Fortran heat equation program.

```
#define the complier
COMPILER = mpif90
# compilation settings, optimization, precision, parallelization
  FLAGS = -O0

# libraries
LIBS = -L${FFTW_LINK} -lfftw3 -lm
# source list for main program
SOURCES =  heat.f90

test: $(SOURCES)
    ${COMPILER} -o heat $(FLAGS) $(SOURCES) $(LIBS)

clean:
  rm *.o
```

2) The Fortran program in listing 10.2 – this should be saved as *heat.f90*

Listing 10.2: A Fortran Fourier spectral program to solve the heat equation using backward Euler timestepping.

```
  !
     --------------------------------------------------------------------

  !
  !
  ! PURPOSE
  !
  ! This program solves heat equation in 1 dimension
  ! u_t=\alpha*u_{xx}
  ! using a the backward Euler method for x\in[0,2\pi]
  !
  ! The boundary conditions are u(0)=u(2\pi)
  ! The initial condition is u=sin(x)
  !
  ! .. Parameters ..
  !  Nx = number of modes in x - power of 2 for FFT
  !  Nt = number of timesteps to take
  !  Tmax = maximum simulation time
  !  plotgap      = number of timesteps between plots
  !  FFTW_IN_PLACE  = value for FFTW input
  !  FFTW_MEASURE   = value for FFTW input
  !  FFTW_EXHAUSTIVE  = value for FFTW input
```

```
!  FFTW_PATIENT   = value for FFTW input
!  FFTW_ESTIMATE  = value for FFTW input
!  FFTW_FORWARD     = value for FFTW input
!  FFTW_BACKWARD  = value for FFTW input
!  pi = 3.14159265358979323846264338327950288419716939937510d0
!  L        = width of box
!  alpha      = heat conductivity
! .. Scalars ..
!  i        = loop counter in x direction
!  n        = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  start      = variable to record start time of program
!  finish     = variable to record end time of program
!  count_rate   = variable for clock count rate
!  planfx     = Forward 1d fft plan in x
!  planbx     = Backward 1d fft plan in x
!  dt       = timestep
! .. Arrays ..
!  u        = approximate REAL solution
!  v        = Fourier transform of approximate solution
!  vna        = temporary field
! .. Vectors ..
!  kx       = fourier frequencies in x direction
!  x        = x locations
!  time       = times at which save data
!  name_config    = array to store filename for data to be saved
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!
   --------------------------------------------------------------------
! External routines required
!
! External libraries required
! FFTW3  -- Fast Fourier Transform in the West Library
!     (http://www.fftw.org/)

PROGRAM main

! Declare variables
IMPLICIT NONE
```

```
INTEGER(kind=4),  PARAMETER   ::  Nx=64
INTEGER(kind=4),  PARAMETER ::  Nt=20
REAL(kind=8), PARAMETER &
  ::  pi=3.14159265358979323846264338327950288419716939937510d0
REAL(kind=8), PARAMETER ::  L=5.0d0
REAL(kind=8), PARAMETER ::  alpha=0.50d0
REAL(kind=8)  ::  dt=0.2d0/REAL(Nt,kind(0d0))
COMPLEX(KIND=8), DIMENSION(:),ALLOCATABLE ::  kx
REAL(kind=8), DIMENSION(:),ALLOCATABLE  ::  x
COMPLEX(KIND=8), DIMENSION(:,:),ALLOCATABLE ::  u,v
REAL(kind=8), DIMENSION(:),ALLOCATABLE  ::  time
COMPLEX(KIND=8), DIMENSION(:),ALLOCATABLE ::  vna
INTEGER(kind=4) ::  i,j,k,n
INTEGER(kind=4) :: start, finish, count_rate, AllocateStatus
INTEGER(kind=4), PARAMETER  :: FFTW_IN_PLACE = 8, FFTW_MEASURE = 0,
    &
  FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32, FFTW_ESTIMATE = 64
INTEGER(kind=4), PARAMETER :: FFTW_FORWARD = -1, FFTW_BACKWARD=1
COMPLEX(KIND=8), DIMENSION(:),ALLOCATABLE :: fftfx,fftbx
INTEGER(kind=8) :: planfx,planbx
CHARACTER*100 :: name_config

CALL system_clock(start,count_rate)
ALLOCATE(kx(1:Nx),x(1:Nx),u(1:Nx,1:1+Nt),v(1:Nx,1:1+Nt),&
      time(1:1+Nt),vna(1:Nx),fftfx(1:Nx),fftbx(1:Nx),&
      stat=AllocateStatus)
  IF (AllocateStatus .ne. 0) STOP

  ! set up ffts
CALL dfftw_plan_dft_1d(planfx,Nx,fftfx(1:Nx),fftbx(1:Nx),&
    FFTW_FORWARD,FFTW_ESTIMATE)
CALL dfftw_plan_dft_1d(planbx,Nx,fftbx(1:Nx),fftfx(1:Nx),&
    FFTW_BACKWARD,FFTW_ESTIMATE)

PRINT *,'Setup FFTs'

! setup fourier frequencies
DO i=1,1+Nx/2
  kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/L
END DO
kx(1+Nx/2)=0.00d0
DO i = 1,Nx/2 -1
  kx(i+1+Nx/2)=-kx(1-i+Nx/2)
END DO
DO i=1,Nx
  x(i)=(-1.00d0 + 2.00d0*REAL(i-1,kind(0d0))/REAL(Nx,KIND(0d0)))*pi
     *L
END DO

PRINT *,'Setup grid and fourier frequencies and splitting
   coefficients'
```

```

u(1:Nx,1)=sin(x(1:Nx))
  ! transform initial data
CALL dfftw_execute_dft_(planfx,u(1:Nx,1),v(1:Nx,1))
PRINT *,'Got initial data, starting timestepping'
time(1)=0.0d0

vna(1:Nx)=v(1:Nx,1)
PRINT *,'Starting timestepping'
DO n=1,Nt
  DO i=1,Nx
    vna(i)=vna(i)/(1-dt*kx(i)*kx(i))
  END DO
  PRINT *,'storing plot data ',n
  time(n+1)=time(n)+dt
  v(1:Nx,n+1)=vna(1:Nx)
  CALL dfftw_execute_dft_(planbx,v(1:Nx,n+1),u(1:Nx,n+1))
  u(1:Nx,n+1)=u(1:Nx,n+1)/REAL(Nx,KIND(0d0))  ! normalize
END DO
PRINT *,'Finished time stepping'
CALL system_clock(finish,count_rate)
PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),'for
   execution'

! Write data out to disk

name_config = 'u.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,1+Nt
  DO i=1,Nx
    WRITE(11,*) REAL(u(i,j))
  END DO
END DO
CLOSE(11)

name_config = 'tdata.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,1+Nt
  WRITE(11,*) time(j)
END DO
CLOSE(11)

name_config = 'xcoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO i=1,Nx
  WRITE(11,*) x(i)
END DO
CLOSE(11)
```

```

  PRINT *,'Saved data'
  DEALLOCATE(kx,x,u,v,&
        time,vna,fftfx,fftbx,&
        stat=AllocateStatus)
    IF (AllocateStatus .ne. 0) STOP

  CALL dfftw_destroy_plan(planbx)
  CALL dfftw_destroy_plan(planfx)
  CALL dfftw_cleanup()
  PRINT *,'Program execution complete'
  END PROGRAM main
```

3) An example submission script to use on the cluster in Listing 10.3 – this should be saved as *fluxsubscript*. More examples can be found at `http://cac.engin.umich.edu/resources/software/pbs.html`. To use it, please change the email address from `your_uniqname@umich.edu` to an email address at which you can receive notifications of when jobs start and are finished.

Listing 10.3: An example submission script for use on Flux.

```
#!/bin/bash
#PBS -N heatequation
#PBS -l nodes=1,walltime=00:10:00
#PBS -l qos=math471f11_flux
#PBS -A math471f11_flux
#PBS -q flux
#PBS -M your_uniqname@umich.edu
#PBS -m abe
#PBS -V
# Create a local directory to run and copy your files to local.
# Let PBS handle your output
mkdir /tmp/${PBS_JOBID}
cp ${HOME}/ParallelMethods/Heat/heatequation /tmp/${PBS_JOBID}/
   heatequation
cd /tmp/${PBS_JOBID}
./heatequation
#Clean up your files
cd
cd ParallelMethods/Heat
# Retrieve your output
cp /tmp/${PBS_JOBID}/u.dat ${HOME}/ParallelMethods/Heat/u.dat
cp /tmp/${PBS_JOBID}/xcoord.dat ${HOME}/ParallelMethods/Heat/xcoord.
   dat
cp /tmp/${PBS_JOBID}/tdata.dat ${HOME}/ParallelMethods/Heat/tdata.dat

/bin/rm -rf /tmp/${PBS_JOBID}
```

4) A Matlab plotting script[2] to generate Fig. 10.1 is in listing 10.4.

Listing 10.4: A Matlab program to plot the computed results.

```matlab
% A Matlab program to plot the computed results

clear all; format compact, format short,
set(0,'defaultaxesfontsize',18,'defaultaxeslinewidth',.9,...
    'defaultlinelinewidth',3.5,'defaultpatchlinewidth',5.5);

% Load data
load('./u.dat');
load('./tdata.dat');
load('./xcoord.dat');
Tsteps = length(tdata);

Nx = length(xcoord); Nt = length(tdata);

u = reshape(u,Nx,Nt);

% Plot data
figure(3); clf; mesh(tdata,xcoord,u); xlabel t; ylabel x; zlabel('u')
    ;
```

10.2 Exercises

1) Please read the resources on the web page `http://cac.engin.umich.edu/started/index.html` to learn how to use the Flux cluster.

2) Modify the Fortran program for the 1-D heat equation to solve the Allen-Cahn equation, with your choice of time stepping scheme. Create a plot of the output of your run. Include the source code and plot in your solutions.

3) Modify the Fortran program for the 1-D heat equation to solve the 2-D heat equation with your choice of time stepping scheme. Your program should save the field at each time step rather than putting all the fields in a single large array. Create a plot of the initial and final states of your run. Include the source code and plots in your solutions.

[2]For many computational problems, one can visualize the results with 10-100 times less computational power than was needed to generate the results, so for problems which are not too large, it is much easier to use a high level language like Matlab to post-process the data.

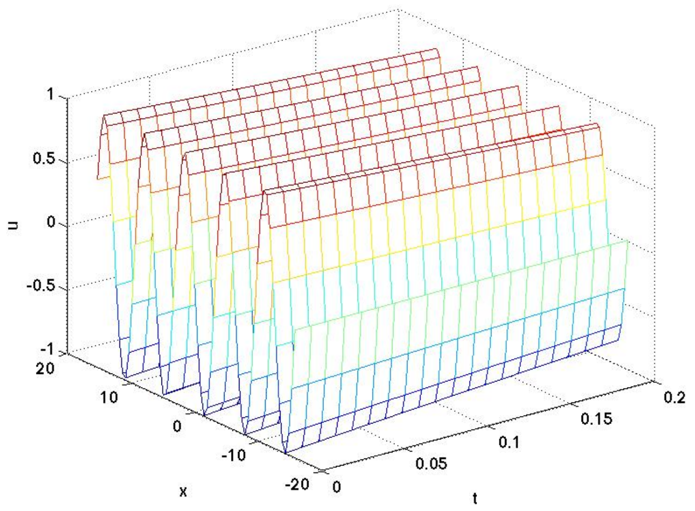

Figure 10.1: The solution to the heat equation computed by Fortran and post-processed by Matlab.

Chapter 11

Introduction to Parallel Programming

11.1 Overview of OpenMP and MPI

To solve large computational problems quickly, it is necessary to take advantage of multiple cores on a CPU (central processing units) and multiple CPUs. Most programs written up until now are sequential and compilers will not typically automatically generate parallel executables, so programmers need to modify the original serial computer code to take advantage of extra processing power. Two standards which specify what libraries that allow for parallel programming should do are OpenMP and MPI (the message passing interface). In this section, we cover the minimal amount of information required to understand, run and modify the programs in this tutorial. More detailed tutorials can be found at `https://computing.llnl.gov/tutorials/` and at `http://www.citutor.org`.

OpenMP is used for parallel programming on shared memory architectures – each compute process has a global view of memory. It allows one to incrementally parallelize an existing Fortran, C or C++ code by adding directives to the original code. It is therefore easy to use. However some care is required in getting good performance when using OpenMP. It is easy to add directives to a serial code, but thought is required in creating a program which will show improved performance and give correct results when made to run in parallel. For the numerical solution of multidimensional partial differential equations on regular grids, it is easy to perform efficient and effective loop based parallelism, so a complete understanding of all the features of OpenMP is not required. OpenMP typically allows one to use 10's of computational cores, in particular allowing one to take advantage of multicore laptops, desktops and workstations.

MPI is used for parallel programming on distributed-memory architectures – when separate compute processes have access to their own local memory and processes must explicitly receive data held in memory belonging to other processes which have sent the data. MPI is a library which allows one to parallelize Fortran, C and C++ programs by adding function calls which explicitly move data from one process to another. Careful thought is required in converting a serial program to a parallel MPI program because the data needs to be decomposed onto different processes, so it is usually difficult to incrementally parallelize a program that uses MPI. The best way to parallelize a program which will use MPI is problem dependent. When solving large problems, one typically does not have enough memory on each process to simply replicate all the data. Thus one wants to split up the data (known as domain decomposition) in such a way as to minimize the amount of message passing that is required to perform a computation correctly. Programming this can be rather complicated and time consuming. Fortunately, by using the 2DECOMP&FFT library [38, 35] which is written on top of MPI, we can avoid having to program many of the data passing operations when writing Fourier spectral codes and still benefit from being able to solve partial differential equations on up to O(10^5) processor cores.

11.2 OpenMP

Please read the tutorial at `https://computing.llnl.gov/tutorials/openMP/`, then answer the following questions:

11.2.1 OpenMP Exercises

1) What is OpenMP?

2) Download a copy of the latest OpenMP specifications from `www.openmp.org`. What version number is the latest specification?

3) Explain what each of the following OpenMP directives does:

   i) !$OMP PARALLEL
   ii) !$OMP END PARALLEL
   iii) !$OMP PARALLEL DO
   iv) !$OMP END PARALLEL DO
   v) !$OMP BARRIER
   vi) !$OMP MASTER
   vii) !$OMP END MASTER

4) Try to understand and then run the Hello World program in listing 11.1 on 1, 2, 6 and 12 threads. Put the output of each run in your solutions, the output will be in a file of the form
   `helloworld.o**********`
   where the last entries above are digits corresponding to the number of the run. An example makefile to compile this on Flux is in listing 11.2. An example submission script is in listing 11.3. To change the number of OpenMP processes that the program will run on from say 2 to 6, change
   `ppn=2`

to
`ppn=6`
and also change the value of the OMP_NUM_THREADS variable from
`OMP_NUM_THREADS=2`
to
`OMP_NUM_THREADS=6`
On Flux, there is a maximum of 12 cores per node, so the largest useful number of threads for most applications is 12.

Listing 11.1: A Fortran program taken from `http://en.wikipedia.org/wiki/OpenMP`, which demonstrates parallelism using OpenMP.

```
!
    ---------------------------------------------------------------------

!
!
! PURPOSE
!
! This program uses OpenMP to print hello world from all available
! threads
!
! .. Parameters ..
!
! .. Scalars ..
!  id        = thread id
!  nthreads      = total number of threads
!
! .. Arrays ..
!
! .. Vectors ..
!
! REFERENCES
! http:// en.wikipedia.org/wiki/OpenMP
!
! ACKNOWLEDGEMENTS
! The program below was modified from one available at the internet
! address in the references. This internet address was last checked
! on 30 December 2011
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!
!
    ---------------------------------------------------------------------
```

```
! External routines required
!
! External libraries required
! OpenMP library
PROGRAM hello90
USE omp_lib
IMPLICIT NONE
INTEGER:: id, nthreads
!$OMP PARALLEL PRIVATE(id)
id = omp_get_thread_num()
nthreads = omp_get_num_threads()
PRINT *, 'Hello World from thread', id
!$OMP BARRIER
IF ( id == 0 ) THEN
  PRINT*, 'There are', nthreads, 'threads'
END IF
!$OMP END PARALLEL
END PROGRAM
```

Listing 11.2: An example makefile for compiling the helloworld program in listing 11.1.

```
#define the complier
COMPILER = ifort
# compilation settings, optimization, precision, parallelization
  FLAGS = -O0 -openmp

# libraries
LIBS =
# source list for main program
SOURCES =  helloworld.f90

test: $(SOURCES)
    ${COMPILER} -o helloworld $(FLAGS) $(SOURCES)

clean:
  rm *.o

clobber:
  rm helloworld
```

Listing 11.3: An example submission script for use on Flux.

```
#!/bin/bash
#PBS -N helloworld
#PBS -l nodes=1:ppn=2,walltime=00:02:00
#PBS -q flux
#PBS -l qos=math471f11_flux
#PBS -A math471f11_flux
#PBS -M your_uniqname@umich.edu
```

```
#PBS -m abe
#PBS -V
#
# Create a local directory to run and copy your files to local.
# Let PBS handle your output
mkdir /tmp/${PBS_JOBID}
cp ${HOME}/ParallelMethods/helloworldOMP/helloworld /tmp/${PBS_JOBID
   }/helloworld
cd /tmp/${PBS_JOBID}

export OMP_NUM_THREADS=2
./helloworld

#Clean up your files
cd ${HOME}/ParallelMethods/helloworldOMP
/bin/rm -rf /tmp/${PBS_JOBID}
```

5) Add OpenMP directives to the loops in the 2-D heat equation solver. Run the resulting program on 1,3,6 and 12 threads and record the time it takes to the program to finish. Make a plot of the final iterate.

11.3 MPI

A copy of the current MPI standard can be found at `http://www.mpi-forum.org/`. It allows for parallelization of Fortran, C and C++ programs. There are newer parallel programming languages such as Co-Array Fortran (CAF) and Unified Parallel C (UPC) which allow the programmer to view memory as a single addressable space even on a distributed-memory machine. However, computer hardware limitations imply that most of the programming concepts used when writing MPI programs will be required to write programs in CAF and UPC. Compiler technology for these languages is also not as well developed as compiler technology for older languages such as Fortran and C, so at the present time, Fortran and C dominate high performance computing. An introduction to the essential concepts required for writing and using MPI programs can be found at `http://www.shodor.org/refdesk/Resources/Tutorials/`. More information on MPI can be found in Gropp, Lusk and Skjellum [22], Gropp, Lusk and Thakur [23] and at `https://computing.llnl.gov/tutorials/mpi/`. There are many resources available online, however once the basic concepts have been mastered, what is most useful is an index of MPI commands, usually a search engine will give you sources of listings, however we have found the following sites useful:

- `http://www.mpi.forum.org/docs/mpi-11-html/node182.html`
- `http://publib.boulder.ibm.com/infocenter/zos/v1r13/index.jsp?topic=%2Fcom.ibm.zos.r13.fomp200%2Fipezps00172.htm`
- `http://www.open-mpi.org/doc/v1.4/`

11.3.1 MPI Exercises

1) What does MPI stand for?

2) Please read the tutorials at `http://www.shodor.org/refdesk/Resources/Tutorials/BasicMPI/` and at `https://computing.llnl.gov/tutorials/mpi/`, then explain what the following commands do:

   - `USE mpi` or `INCLUDE 'mpif.h'`
   - `MPI_INIT`
   - `MPI_COMM_SIZE`
   - `MPI_COMM_RANK`
   - `MPI_FINALIZE`

3) What is the version number of the current MPI standard?

3) Try to understand the Hello World program in listing 11.4. Explain how it differs from 11.1. Run the program in listing 11.4 on 1, 2, 6, 12 and 24 MPI processes[1]. Put the output of each run in your solutions, the output will be in a file of the form
   `helloworld.o**********`
   where the last entries above are digits corresponding to the number of the run. An example makefile to compile this on Flux is in listing 11.5. An example submission script is in listing 11.6. To change the number of MPI processes that the program will run on from say 2 to 6, change
   `ppn=2`
   to
   `ppn=6`
   and also change the submission script from
   `mpirun -np 2 ./helloworld`
   to
   `mpirun -np 6 ./helloworld`.

   On Flux, there is a maximum of 12 cores per node, so if more than 12 MPI processes are required, one needs to change the number of nodes as well. The total number of cores required is equal to the number of nodes multiplied by the number of processes per node. Thus to use 24 processes change
   `nodes=1:ppn=2`
   to
   `nodes=2:ppn=12`
   and also change the submission script from
   `mpirun -np 2 ./helloworld`
   to
   `mpirun -np 24 ./helloworld`.

[1]One can run this program on many more than 24 processes, however, the output becomes quite excessive

Listing 11.4: A Fortran program which demonstrates parallelizm using MPI.

```
! --------------------------------------------------------------------
!
!
! PURPOSE
!
! This program uses MPI to print hello world from all available
! processes
!
! .. Parameters ..
!
! .. Scalars ..
!  myid        = process id
!  numprocs      = total number of MPI processes
!  ierr        = error code
!
! .. Arrays ..
!
! .. Vectors ..
!
! REFERENCES
! http:// en.wikipedia.org/wiki/OpenMP
!
! ACKNOWLEDGEMENTS
! The program below was modified from one available at the internet
! address in the references. This internet address was last checked
! on 30 December 2011
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!
! --------------------------------------------------------------------
! External routines required
!
! External libraries required
! MPI library
PROGRAM hello90
USE MPI
IMPLICIT NONE
INTEGER(kind=4) :: myid, numprocs, ierr

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
```

```fortran

  PRINT*, 'Hello World from process', myid
  CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
  IF ( myid == 0 ) THEN
    PRINT*, 'There are ', numprocs, ' MPI processes'
  END IF
  CALL MPI_FINALIZE(ierr)
  END PROGRAM
```

Listing 11.5: An example makefile for compiling the helloworld program in listing 11.4.

```makefile
#define the complier
COMPILER = mpif90
# compilation settings, optimization, precision, parallelization
  FLAGS = -O0

# libraries
LIBS =
# source list for main program
SOURCES =  helloworld.f90

test: $(SOURCES)
    ${COMPILER} -o helloworld $(FLAGS) $(SOURCES)

clean:
  rm *.o

clobber:
  rm helloworld
```

Listing 11.6: An example submission script for use on Flux.

```bash
#!/bin/bash
#PBS -N helloworld
#PBS -l nodes=1:ppn=2,walltime=00:02:00
#PBS -q flux
#PBS -l qos=math471f11_flux
#PBS -A math471f11_flux
#PBS -M your_uniqname@umich.edu
#PBS -m abe
#PBS -V
#
# Create a local directory to run and copy your files to local.
# Let PBS handle your output
mkdir /tmp/${PBS_JOBID}
cp ${HOME}/ParallelMethods/helloworldMPI/helloworld /tmp/${PBS_JOBID
   }/helloworld
cd /tmp/${PBS_JOBID}

```

```
mpirun -np 2 ./helloworld

#Clean up your files
cd ${HOME}/ParallelMethods/helloworldMPI
/bin/rm -rf /tmp/${PBS_JOBID}
```

11.4 A first parallel program: Monte Carlo Integration

To introduce the basics of parallel programming in a context that is a little more complicated than *Hello World*, we will consider Monte Carlo integration. We review important concepts from probability and Riemann integration, and then give example algorithms and explain why parallelization may be helpful.

11.4.1 Probability

Definition 11.4.1. *$f : U \subset \mathbb{R}^2 \to \mathbb{R}_+$ is a* **probability density function** *if*

$$\int\int_U f \mathrm{d}A = 1$$

Definition 11.4.2. *If f is a probability density function which takes the set $U \subset \mathbb{R}^2$, then the probability of events in the set $W \subset U$ occurring is*

$$P(W) = \int\int_W f \mathrm{d}A.$$

Example 11.4.1. *The joint density for it to snow x inches tomorrow and for Kelly to win y dollar in the lottery tomorrow is given by*

$$f = \frac{c}{(1+x)(100+y)}$$

for

$$x, y \in [0, 100] \times [0, 100]$$

and $f = 0$ otherwise. Find c.

Definition 11.4.3. *Suppose X is a random variable with probability density function $f_1(x)$ and Y is a random variable with a probability density function $f_2(y)$. Then X and Y are* **independent random variables** *if their joint density function is*

$$f(x, y) = f_1(x) f_2(y).$$

Example 11.4.2. *The probability it will snow tomorrow and the probability Kelly will win the lottery tomorrow are independent random variables.*

Definition 11.4.4. *If $f(x,y)$ is a probability density function for the random variables X and Y, the* **X mean** *is*

$$\mu_1 = \bar{X} = \int\int xf\mathrm{d}A$$

and the **Y mean** *is*

$$\mu_2 = \bar{Y} = \int\int yf\mathrm{d}A.$$

Remark 11.4.1. *The X mean and the Y mean are the expected values of X and Y.*

Definition 11.4.5. *If $f(x,y)$ is a probability density function for the random variables X and Y, the* **X variance** *is*

$$\sigma_1^2 = \overline{(X-\bar{X})^2} = \int\int (x-\bar{X})^2 f\mathrm{d}A$$

and the **Y variance** *is*

$$\sigma_2^2 = \overline{(Y-\bar{Y})^2} = \int\int (y-\bar{Y})^2 f\mathrm{d}A.$$

Definition 11.4.6. *The standard deviation is defined to be the square root of the variance.*

Example 11.4.3. *Find an expression for the probability that it will snow more than 1.1 times the expected snowfall and also that Kelly will win more than 1.2 times the expected amount in the lottery.*

11.4.2 Exercise

1) A class is graded on a curve. It is assumed that the class is a representative sample of the population, the probability density function for the numerical score x is given by

$$f(x) = C\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

For simplicity we assume that x can take on the values $-\infty$ and ∞, though in actual fact the exam is scored from 0 to 100.

   a) Determine C using results from your previous homework.

   b) Suppose there are 240 students in the class and the mean and standard deviation for the class is not reported. As an enterprising student, you poll 60 of your fellow students (we shall suppose they are selected randomly). You find that the mean for these 60 students is 55% and the standard deviation is 10%. Use the Student's t distribution `http://en.wikipedia.org/wiki/Student%27s_t-distribution` to estimate the 90% confidence interval for the actual sample mean. Make a sketch of the t-distribution probability density function and shade the region which corresponds to the 90% confidence interval for the sample mean.[2]

[2]The Student's t distribution is implemented in many numerical packages such as Maple, Mathematica, Matlab, R, Sage etc., so if you need to use to obtain numerical results, it is helpful to use on of these packages.

Remark Fortunately, all the students are hard working, so the possibility of a negative score, although possible, is extremely low, and so we neglect it to make the above computation easier.

11.4.3 Riemann Integration

Recall that we can approximate integrals by Riemann sums. There are many integrals one cannot evaluate analytically, but for which a numerical answer is required. In this section, we shall explore a simple way of doing this on a computer. Suppose we want to find

$$I2d = \int_0^1 \int_0^4 x^2 + 2y^2 \mathrm{d}y\mathrm{d}x.$$

If we do this analytically we find

$$I2d = 44.$$

Let us suppose we have forgotten how to integrate, and so we do this numerically. We can do so using the following Matlab code:

Listing 11.7: A Matlab program which demonstrates how to approximate an integral by a sum.

```
% A program to approximate an integral

clear all; format compact; format short;

nx=1000;      % number of points in x
xend=1;      % last discretization point
xstart=0;    % first discretization point
dx=(xend-xstart)/(nx-1);     % size of each x sub-interval

ny=4000;      % number of points in y
yend=4;      % last discretization point
ystart=0;    % first discretization point
dy=(yend-ystart)/(ny-1);     % size of each y sub-interval

% create vectors with points for x and y
for i=1:nx
    x(i)=xstart+(i-1)*dx;
end
for j=1:ny
    y(j)=ystart+(j-1)*dy;
end

% Approximate the integral by a sum
I2d=0;
for i=1:nx
    for j=1:ny
        I2d=I2d+(x(i)^2+2*y(j)^2)*dy*dx;
    end
end
% print out final answer
I2d
```

We can do something similar in three dimensions. Suppose we want to calculate

$$I3d = \int_0^1 \int_0^1 \int_0^4 x^2 + 2y^2 + 3z^2 \mathrm{d}z \mathrm{d}y \mathrm{d}x.$$

Analytically we find that

$$I3d = 68$$

11.4.4 Exercises

1) Modify the Matlab code to perform the three dimensional integral.

2) Try and determine how the accuracy of either the two or three dimensional method varies as the number of subintervals is changed.

11.4.5 Monte Carlo Integration

[3] It is possible to extend the above integration schemes to higher and higher dimensional integrals. This can become computationally intensive and an alternate method of integration based on probability is often used. The method we will discuss is called the *Monte Carlo method*. The idea behind it is based on the concept of the *average value* of a function, which you learned in single-variable calculus. Recall that for a continuous function $f(x)$, the **average value** $\bar{f}$ of f over an interval $[a, b]$ is defined as

$$\bar{f} \;=\; \frac{1}{b-a} \int_a^b f(x)\,dx \;. \tag{11.1}$$

The quantity $b - a$ is the length of the interval $[a, b]$, which can be thought of as the "volume" of the interval. Applying the same reasoning to functions of two or three variables, we define the **average value** of $f(x, y)$ over a region R to be

$$\bar{f} \;=\; \frac{1}{A(R)} \iint\limits_R f(x,y)\,dA \;, \tag{11.2}$$

where $A(R)$ is the area of the region R, and we define the **average value** of $f(x, y, z)$ over a solid S to be

$$\bar{f} \;=\; \frac{1}{V(S)} \iiint\limits_S f(x,y,z)\,dV \;, \tag{11.3}$$

[3]This section is taken from Chapter 3 of Vector Calculus by Michael Corral which is available at `http://www.mecmath.net/` and where Java and Sage programs for doing Monte Carlo integration can be found.

where $V(S)$ is the volume of the solid S. Thus, for example, we have

$$\iint\limits_R f(x,y)\,dA \;=\; A(R)\bar{f}\ . \tag{11.4}$$

The average value of $f(x,y)$ over R can be thought of as representing the sum of all the values of f divided by the number of points in R. Unfortunately there are an infinite number (in fact, *uncountably* many) points in any region, i.e. they can not be listed in a discrete sequence. But what if we took a very large number N of *random* points in the region R (which can be generated by a computer) and then took the average of the values of f for those points, and used that average as the value of $\bar{f}$? This is exactly what the Monte Carlo method does. So in formula (11.4) the approximation we get is

$$\iint\limits_R f(x,y)\,dA \;\approx\; A(R)\bar{f} \pm A(R)\sqrt{\frac{\overline{f^2}-(\bar{f})^2}{N}}\ , \tag{11.5}$$

where

$$\bar{f} \;=\; \frac{\sum_{i=1}^{N} f(x_i,y_i)}{N} \quad \text{and} \quad \overline{f^2} \;=\; \frac{\sum_{i=1}^{N} (f(x_i,y_i))^2}{N}\ , \tag{11.6}$$

with the sums taken over the N random points (x_1,y_1), ..., (x_N,y_N). The $\pm$ "error term" in formula (11.5) does not really provide hard bounds on the approximation. It represents a single *standard deviation* from the *expected* value of the integral. That is, it provides a *likely* bound on the error. Due to its use of random points, the Monte Carlo method is an example of a *probabilistic* method (as opposed to *deterministic* methods such as the Riemann sum approximation method, which use a specific formula for generating points).

For example, we can use the formula in eq. (11.5) to approximate the volume V under the surface $z = x^2 + 2y^2$ over the rectangle $R = (0,1) \times (0,4)$. Recall that the actual volume is 44. Below is a Matlab code that calculates the volume using Monte Carlo integration

Listing 11.8: A Matlab program which demonstrates how to use the Monte Carlo method to calculate the volume below $z = x^2 + 2y^2$, with $(x,y) \in (0,1) \times (0,4)$.

```
% A program to approximate an integral using the Monte Carlos method

% This program can be made much faster by using Matlab's matrix and vector
% operations, however to allow easy translation to other languages we have
% made it as simple as possible.

Numpoints=65536;    % number of random points

I2d=0; % Initialize value
I2dsquare=0; % initial variance
for n=1:Numpoints
    % generate random number drawn from a uniform distribution on (0,1)
    x=rand(1);
    y=rand(1)*4;
    I2d=I2d+x^2+2*y^2;
    I2dsquare=I2dsquare+(x^2+2*y^2)^2;
end
% we sclae the integral by the total area and divide by the number of
% points used
I2d=I2d*4/Numpoints
% we also output an estimated error
I2dsquare=I2dsquare*4/Numpoints;
EstimError=4*sqrt( (I2d^2-I2dsquare)/Numpoints)
```

The results of running this program with various numbers of random points are shown below:

```
N = 16: 41.3026 +/- 30.9791
N = 256: 47.1855 +/- 9.0386
N = 4096: 43.4527 +/- 2.0280
N = 65536: 44.0026 +/- 0.5151
```

As you can see, the approximation is fairly good. As $N \to \infty$, it can be shown that the Monte Carlo approximation converges to the actual volume (on the order of $O(\sqrt{N})$, in computational complexity terminology).

In the above example the region R was a rectangle. To use the Monte Carlo method for a nonrectangular (bounded) region R, only a slight modification is needed. Pick a rectangle $\tilde{R}$ that encloses R, and generate random points in that rectangle as before. Then use those points in the calculation of $\bar{f}$ only if they are inside R. There is no need to calculate the area of R for formula (11.5) in this case, since the exclusion of points not inside R allows you to use the area of the rectangle $\tilde{R}$ instead, similar to before.

For instance, one can show that the volume under the surface $z = 1$ over the nonrectangular region $R = \{(x, y) : 0 \le x^2 + y^2 \le 1\}$ is π. Since the rectangle $\tilde{R} = [-1, 1] \times [-1, 1]$ contains R, we can use a similar program to the one we used, the largest change being a check to see if $y^2 + x^3 \le 1$ for a random point (x, y) in $[-1, 1] \times [-1, 1]$. A Matlab code listing which demonstrates this is below:

Listing 11.9: A Matlab program which demonstrates how to use the Monte Carlo method to calculate the area of an irregular region and also to calculate π.

```
% A program to approximate an integral using the Monte Carlos method

% This program can be made much faster by using Matlab's matrix and vector
% operations, however to allow easy translation to other languages we have
% made it as simple as possible.

Numpoints=256;   % number of random points

I2d=0; % Initialize value
I2dsquare=0; % initial variance
```

```
for n=1:Numpoints
    % generate random number drawn from a uniform distribution on (0,1)
      and
    % scale this to (-1,1)
    x=2*rand(1)-1;
    y=2*rand(1) -1;
    if ((x^2+y^2) <1)
        I2d=I2d+1;
        I2dsquare=I2dsquare+1;
    end
end
% We scale the integral by the total area and divide by the number of
% points used
I2d=I2d*4/Numpoints
% we also output an estimated error
I2dsquare=I2dsquare*4/Numpoints;
EstimError=4*sqrt( (I2d^2-I2dsquare)/Numpoints)
```

The results of running the program with various numbers of random points are shown below:

```
N = 16: 3.5000 +/- 2.9580
N = 256: 3.2031 +/- 0.6641
N = 4096: 3.1689 +/- 0.1639
N = 65536: 3.1493 +/- 0.0407
```

To use the Monte Carlo method to evaluate triple integrals, you will need to generate random triples (x, y, z) in a parallelepiped, instead of random pairs (x, y) in a rectangle, and use the volume of the parallelepiped instead of the area of a rectangle in formula (11.5). For a more detailed discussion of numerical integration methods, please take a further course in mathematics.

11.4.6 Exercises

1) Write a program that uses the Monte Carlo method to approximate the double integral $\iint\limits_R e^{xy}\, dA$, where $R = [0,1] \times [0,1]$. Show the program output for $N = 10$, 100, 1000, 10000, 100000 and 1000000 random points.

2) Write a program that uses the Monte Carlo method to approximate the triple integral $\iiint\limits_S e^{xyz}\, dV$, where $S = [0,1] \times [0,1] \times [0,1]$. Show the program output for $N = 10$, 100, 1000, 10000, 100000 and 1000000 random points.

3) Use the Monte Carlo method to approximate the volume of a sphere of radius 1.

11.4.7 Parallel Monte Carlo Integration

As you may have noticed, the algorithms are simple, but can require very many grid points to become accurate. It is therefore useful to run these algorithms on a parallel computer. We will demonstrate a parallel Monte Carlo calculation of π. Before we can do this, we need to learn how to use a parallel computer[4].

We now examine a Fortran program for calculating π. These programs are taken from `http://chpc.wustl.edu/mpi-fortran.html`, where further explanation can be found. The original source of these programs appears to be Using MPI by Gropp, Lusk and Skjellum.

Serial

Listing 11.10: A serial Fortran program which demonstrates how to calculate π using a Monte Carlo method.

```





!---------------------------------------------------------------------
!
!
! PURPOSE
!
! This program use a monte carlo method to calculate pi
!
! .. Parameters ..
!  npts        = total number of Monte Carlo points
!  xmin        = lower bound for integration region
!  xmax              = upper bound for integration region
! .. Scalars ..
!  i                 = loop counter
!  f         = average value from summation
!  sum               = total sum
!  randnum           = random number generated from (0,1) uniform
!                      distribution
!  x                 = current Monte Carlo location
! .. Arrays ..
!
! .. Vectors ..
!
! REFERENCES
! http://chpc.wustl.edu/mpi-fortran.html
! Gropp, Lusk and Skjellum, "Using MPI" MIT press (1999)
```

[4]Many computers and mobile telephones produced today have 2 or more cores and so can be considered parallel, but here we mean computers with over hundreds of cores.

```
!
! ACKNOWLEDGEMENTS
! The program below was modified from one available at the internet
! address in the references. This internet address was last checked
! on 30 March 2012
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
! None
PROGRAM monte_carlo
  IMPLICIT NONE

  INTEGER(kind=8), PARAMETER      :: npts = 1e10
  REAL(kind=8), PARAMETER     :: xmin=0.0d0,xmax=1.0d0
  INTEGER(kind=8)         :: i
  REAL(kind=8)          :: f,sum, randnum,x

  DO i=1,npts
    CALL random_number(randnum)
    x = (xmax-xmin)*randnum + xmin
    sum = sum + 4.0d0/(1.0d0 + x**2)
  END DO
  f = sum/npts
  PRINT*,'PI calculated with ',npts,' points = ',f

  STOP
  END
```

Listing 11.11: An example makefile for compiling the program in listing 11.10.

```
#define the complier
COMPILER = mpif90
# compilation settings, optimization, precision, parallelization
  FLAGS = -O0

# libraries
LIBS =
# source list for main program
SOURCES =  montecarloserial.f90

test: $(SOURCES)
    ${COMPILER} -o montecarloserial $(FLAGS) $(SOURCES)
```

```

clean:
  rm *.o

clobber:
  rm  montecarloserial
```

Listing 11.12: An example submission script for use on Trestles located at the San Diego Supercomputing Center.

```
#!/bin/bash
# the queue to be used.
#PBS -q shared
# specify your project allocation
#PBS -A mia122
# number of nodes and number of processors per node requested
#PBS -l nodes=1:ppn=1
# requested Wall-clock time.
#PBS -l walltime=00:05:00
# name of the standard out file to be "output-file".
#PBS -o job_output
# name of the job
#PBS -N MCserial
# Email address to send a notification to, change "youremail"
    appropriately
#PBS -M youremail@umich.edu
# send a notification for job abort, begin and end
#PBS -m abe
#PBS -V
cd $PBS_O_WORKDIR #change to the working directory
mpirun_rsh -np 1 -hostfile $PBS_NODEFILE  montecarloserial
```

Parallel

Listing 11.13: A parallel Fortran program which demonstrates how to calculate π using MPI.

```





  !-------------------------------------------------------------------
  !
  !
  ! PURPOSE
  !
  ! This program uses MPI to do a parallel monte carlo calculation of pi
```

```
!
! .. Parameters ..
!  npts        = total number of Monte Carlo points
!  xmin        = lower bound for integration region
!  xmax              = upper bound for integration region
! .. Scalars ..
!  mynpts      = this processes number of Monte Carlo points
!  myid        = process id
!  nprocs      = total number of MPI processes
!  ierr        = error code
!  i                 = loop counter
!  f         = average value from summation
!  sum               = total sum
!  mysum             = sum on this process
!  randnum           = random number generated from (0,1) uniform
!                      distribution
!  x                 = current Monte Carlo location
!  start       = simulation start time
!  finish      = simulation end time
! .. Arrays ..
!
! .. Vectors ..
!
! REFERENCES
! http://chpc.wustl.edu/mpi-fortran.html
! Gropp, Lusk and Skjellum, "Using MPI" MIT press (1999)
!
! ACKNOWLEDGEMENTS
! The program below was modified from one available at the internet
! address in the references. This internet address was last checked
! on 30 March 2012
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!
!----------------------------------------------------------------------
! External routines required
!
! External libraries required
! MPI library
  PROGRAM monte_carlo_mpi
  USE MPI
  IMPLICIT NONE

  INTEGER(kind=8), PARAMETER  :: npts = 1e10
  REAL(kind=8), PARAMETER    :: xmin=0.0d0,xmax=1.0d0
  INTEGER(kind=8)       :: mynpts
  INTEGER(kind=4)            :: ierr, myid, nprocs
```

```fortran
    INTEGER(kind=8)             :: i
    REAL(kind=8)              :: f,sum,mysum,randnum
    REAL(kind=8)              :: x, start, finish

    ! Initialize MPI
    CALL MPI_INIT(ierr)
    CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
    CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
    start=MPI_WTIME()

  ! Calculate the number of points each MPI process needs to generate
    IF (myid .eq. 0) THEN
      mynpts = npts - (nprocs-1)*(npts/nprocs)
    ELSE
      mynpts = npts/nprocs
    ENDIF

    ! set initial sum to zero
    mysum = 0.0d0
  ! use loop on local process to generate portion of Monte Carlo integral
    DO i=1,mynpts
      CALL random_number(randnum)
      x = (xmax-xmin)*randnum + xmin
      mysum = mysum + 4.0d0/(1.0d0 + x**2)
    ENDDO

  ! Do a reduction and sum the results from all processes
    CALL MPI_REDUCE(mysum,sum,1,MPI_DOUBLE_PRECISION,MPI_SUM,&
          0,MPI_COMM_WORLD,ierr)
    finish=MPI_WTIME()

    ! Get one process to output the result and running time
    IF (myid .eq. 0) THEN
         f = sum/npts
         PRINT*,'PI calculated with ',npts,' points = ',f
         PRINT*,'Program took ', finish-start, ' for Time stepping'
    ENDIF

    CALL MPI_FINALIZE(ierr)

    STOP
    END PROGRAM
```

Listing 11.14: An example makefile for compiling the program in listing 11.13.

```makefile
#define the complier
COMPILER = mpif90
# compilation settings, optimization, precision, parallelization
  FLAGS = -O0

```

```
# libraries
LIBS =
# source list for main program
SOURCES =  montecarloparallel.f90

test: $(SOURCES)
    ${COMPILER} -o montecarloparallel $(FLAGS) $(SOURCES)

clean:
  rm *.o

clobber:
  rm  montecarloparallel
```

Listing 11.15: An example submission script for use on Trestles located at the San Diego Supercomputing Center.

```
#!/bin/bash
# the queue to be used.
#PBS -q normal
# specify your project allocation
#PBS -A mia122
# number of nodes and number of processors per node requested
#PBS -l nodes=1:ppn=32
# requested Wall-clock time.
#PBS -l walltime=00:05:00
# name of the standard out file to be "output-file".
#PBS -o job_output
# name of the job, you may want to change this so it is unique to you
#PBS -N MPI_MCPARALLEL
# Email address to send a notification to, change "youremail"
    appropriately
#PBS -M youremail@umich.edu
# send a notification for job abort, begin and end
#PBS -m abe
#PBS -V

# change to the job submission directory
cd $PBS_O_WORKDIR
# Run the job
mpirun_rsh -np 32 -hostfile $PBS_NODEFILE  montecarloparallel
```

11.4.8 Exercises

1) Explain why using Monte Carlo to evaluate

$$\int_0^1 \frac{1}{1+x^2}\mathrm{d}x$$

allows you to find π and, in your own words, explain what the serial and parallel programs do.

2) Find the time it takes to run the Parallel Monte Carlo program on 32, 64, 128, 256 and 512 cores.

3) Use a parallel Monte Carlo integration program to evaluate

$$\iint x^2 + y^6 + \exp(xy)\cos(y\exp(x))\mathrm{d}A$$

over the unit circle.

4) Use a parallel Monte Carlo integration program to approximate the volume of the ellipsoid $\frac{x^2}{9} + \frac{y^2}{4} + \frac{z^2}{1} = 1$. Use either OpenMP or MPI.

5) Write parallel programs to find the volume of the 4 dimensional sphere

$$1 \geq \sum_{i=1}^{4} x_i^2.$$

Try both Monte Carlo and Riemann sum techniques. Use either OpenMP or MPI.

Chapter 12

The Cubic Nonlinear Schrödinger Equation

12.1 Background

The cubic nonlinear Schrödinger equation occurs in a variety of areas, including, quantum mechanics, nonlinear optics and surface water waves. A general introduction can be found at `http://en.wikipedia.org/wiki/Schrodinger_equation` and `http://en.wikipedia.org/wiki/Nonlinear_Schrodinger_equation`. A mathematical introduction to Schrödinger equations can be found in Sulem and Sulem [53] and Yang [61]. In this section we will introduce the idea of operator splitting and then go on to explain how this can be applied to the nonlinear Schrödinger equation in one, two and three dimensions. In one dimension, one can show that the cubic nonlinear Schrödinger equation is subcritical, and hence one has solutions which exist for all time. In two dimensions, it is H^1 critical, and so solutions may exhibit blow-up of the H^1 norm, that is the integral of the square of the gradient of the solution can become infinite in finite time. Finally, in three dimensions, the nonlinear Schrödinger equation is L^2 supercritical, and so the integral of the square of the solution can also become infinite in finite time. For an introduction to norms and Hilbert spaces, see a textbook on partial differential equations or analysis, such as Evans [17], Linares and Ponce [40], Lieb and Loss [39] or Renardy and Rogers [50]. A question of interest is how this blow-up occurs and numerical simulations are often used to understand this; see Sulem and Sulem [53] for examples of this. The cubic nonlinear Schrödinger equation[1] is given by

$$i\psi_t + \Delta\psi \pm |\psi|^2\psi = 0, \tag{12.1}$$

where ψ is the wave function and Δ is the Laplacian operator, so in one dimension it is ∂_{xx}, in two dimensions, $\partial_{xx} + \partial_{yy}$ and in three dimensions it is $\partial_{xx} + \partial_{yy} + \partial_{zz}$. The $+$ corresponds to the focusing cubic nonlinear Schrödinger equation and the $-$ corresponds to the defocusing cubic nonlinear Schrödinger equation. This equation has many conserved

[1]To simplify the presentation, we primarily consider the focusing cubic nonlinear Schrödinger equation.

quantities, including the "mass",

$$\int_\Omega |\psi|^2 \mathrm{d}^n \boldsymbol{x} \tag{12.2}$$

and the "energy",

$$\int_\Omega \frac{1}{2}|\nabla\psi|^2 \mp \frac{1}{4}|\psi|^4 \mathrm{d}^n \boldsymbol{x} \tag{12.3}$$

where n is the dimension and Ω is the domain of the solution. As explained by Klein [31], these two quantities can provide useful checks on the accuracy of numerically generated solutions.

12.2 Splitting

We will consider a numerical method to solve this equation known as splitting. This method occurs in several applications, and is a useful numerical method when the equation can be split into two separate equations, each of which can either be solved exactly, or each part is best solved by a different numerical method. Introductions to splitting can be found in Holden et al. [27], McLachlan and Quispel [43], Thalhammer [55], Shen, Tang and Wang [52], Weideman and Herbst [60] and Yang [61], and also at `http://en.wikipedia.org/wiki/Split-step_method`. For those interested in a comparison of time stepping methods for the nonlinear Schrödinger equation, see Klein [31]. To describe the basic idea of the method, we consider an example given in Holden et al. [28], which is the ordinary differential equation,

$$u_t = u(u-1), \quad u(t=0) = 0.8. \tag{12.4}$$

We can solve this equation relatively simply by separation of variables to find that

$$u(t) = \frac{4}{4+\exp(t)}. \tag{12.5}$$

Now, an interesting observation is that we can also solve the equations $u_t = u^2$ and $u_t = -u$ individually. For the first we get that $u(t) = \frac{u(0)}{1-tu(0)}$ and for the second we get that $u(t) = u(0)\exp(-t)$. The principle behind splitting is to solve these two separate equations alternately for short periods of time. We will describe Strang splitting, although there are other forms of splitting, such as Godunov splitting and also additive splittings. We will not describe these here, but refer you to the previously mentioned references, in particular Holden et al. [27]. To understand how we can solve the differential equation using splitting, consider the linear ordinary differential equation

$$u_t = u + 2u, \quad u(0) = 1. \tag{12.6}$$

We can first solve $p_t = p$ for a time $\delta t/2$ and then using $q(0) = p(\delta t/2)$, we solve $q_t = 2q$ also for a time δt to get $q(\delta t)$ and finally solve $r_t = r$ for a time $\delta t/2$ with initial data $r(0) = q(\delta t)$. Thus in this case $p(\delta t) = \exp(\delta t/2)$, $q(\delta t) = p(\delta t/2)\exp(2\delta t) =$

$\exp(5\delta t/2)$ and $u(\delta t) \approx r(\delta t/2) = q(\delta t)\exp(\delta t/2) = \exp(3\delta t)$, which in this case is the exact solution. One can perform a similar splitting for matrix differential equations. Consider solving $\boldsymbol{u}_t = (\boldsymbol{A} + \boldsymbol{B})\boldsymbol{u}$, where $\boldsymbol{A}$ and $\boldsymbol{B}$ are $n \times n$ matrices, the exact solution is $\boldsymbol{u} = \exp\left((\boldsymbol{A} + \boldsymbol{B})t\right)\boldsymbol{u}(t = 0)$, and an approximate solution produced after one time step of splitting is $u(\delta t) \approx u(0)\exp(\boldsymbol{A}\delta t)\exp(\boldsymbol{B}\delta t)$, which is not in general equal to $u(t = 0)\exp\left((\boldsymbol{A} + \boldsymbol{B})\delta t\right)$ unless the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ commute[2], and so the error in doing splitting in this case is of the form $(\boldsymbol{AB} - \boldsymbol{BA})\delta t$[3]. Listing B.7 uses Matlab to demonstrate how to do splitting for eq. (12.4).

Listing 12.1: A Matlab program which uses Strang splitting to solve an ODE.

```matlab
% A program to solve the u_t=u(u-1) using a
% Strang Splitting method

clear all; format compact; format short;
set(0,'defaultaxesfontsize',30,'defaultaxeslinewidth',.7,...
    'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,...
    'defaultaxesfontweight','bold')
Nt = 1000;                              % number of time slices
tmax = 1;                               % maximum time
dt=tmax/Nt;                             % increment between times
time=(linspace(1,Nt,Nt)-1)*dt;          % time
uexact=4./(4+exp(time));                % exact solution
u(1)=0.8

for i=1:Nt-1
    c=-1/u(i);
    utemp=-1/(c+0.5*dt);
    utemp2=utemp*exp(-dt);
    c=-1/utemp2;
    u(i+1)=-1/(c+0.5*dt);
end
figure(1)
plot(time,u,'r+',time,uexact,'b-');
```

12.3 Exercises

1) Modify the Matlab code to calculate the error at time 1 for several different choices of timestep. Numerically verify that Strang splitting is second order accurate.

2) Modify the Matlab code to use Godunov splitting where one solves $u1_t = u1$ for a time δt and then using $u1(\delta t)$ as initial data solves $u2_t = 2u2$ also for a time δt to get the

[2]That is$\boldsymbol{AB} = \boldsymbol{BA}$.

[3]One can derive this by using the series expansion of the exponential function, $\exp(\boldsymbol{A}t) = \sum_{n=0}^{\infty} \frac{(\boldsymbol{A}t)^n}{n!}$, and subtracting $\exp((\boldsymbol{A} + \boldsymbol{B})\delta t)$ from $\exp(\boldsymbol{A}\delta t)\exp(\boldsymbol{B}\delta t)$.

approximation to $u(\delta t)$. Calculate the error at time 1 for several different choices of timestep. Numerically verify that Godunov splitting is first order accurate.

12.4 Serial

For the nonlinear Schrödinger equation

$$i\psi_t \pm |\psi|^2\psi + \Delta\psi = 0, \tag{12.7}$$

we first solve

$$i\psi_t + \Delta\psi = 0 \tag{12.8}$$

exactly using the Fourier transform to get $\psi(\delta t/2, \cdot)$. We then solve

$$i\psi_t \pm |\psi|^2\psi = 0 \tag{12.9}$$

with $\psi(\delta t/2, \cdot)$ as initial data for a time step of δt. As explained by Klein [31] and Thalhammer [55], this can be solved exactly in real space because in eq. (12.9), $|\psi|^2$ is a conserved quantity at every point in space and time. To show this, let ψ^* denote the complex conjugate of ψ, so that

$$\frac{\mathrm{d}|\psi|^2}{\mathrm{d}t} = \psi^*\frac{\mathrm{d}\psi}{\mathrm{d}t} + \frac{\mathrm{d}\psi^*}{\mathrm{d}t}\psi = \psi^*\left(\pm i|\psi|^2\psi\right) + \left(\pm i|\psi|^2\psi\right)^*\psi = 0. \tag{12.10}$$

Another half step using eq. (12.8) is then computed using the solution produced by solving eq. (12.9) to obtain the approximate solution at time δt. Example Matlab codes demonstrating splitting follow.

12.4.1 Example Matlab Programs for the Nonlinear Schrödinger Equation

The program in listing 12.2 computes an approximation to an explicitly known exact solution to the focusing nonlinear Schrödinger equation.

Listing 12.2: A Matlab program which uses Strang splitting to solve the one dimensional nonlinear Schrödinger equation.

```
% A program to solve the nonlinear Schr\"{o}dinger equation using a
% splitting method

clear all; format compact; format short;
set(0,'defaultaxesfontsize',30,'defaultaxeslinewidth',.7,...
    'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,...
    'defaultaxesfontweight','bold')

Lx = 20;             % period  2*pi * L
```

```
Nx = 16384;          % number of harmonics
Nt = 1000;           % number of time slices
dt = 0.25*pi/Nt;     % time step
U=zeros(Nx,Nt/10);

Es = -1; % focusing or defocusing parameter

% initialise variables
x = (2*pi/Nx)*(-Nx/2:Nx/2 -1)'*Lx;       % x coordinate
kx = 1i*[0:Nx/2-1 0 -Nx/2+1:-1]'/Lx;     % wave vector
k2x = kx.^2;                         % square of wave vector
% initial conditions
t=0; tdata(1)=t;
u=4*exp(1i*t)*(cosh(3*x)+3*exp(8*1i*t)*cosh(x))...
    ./(cosh(4*x)+4*cosh(2*x)+3*cos(8*t));
v=fft(u);
figure(1); clf; plot(x,u);xlim([-2,2]); drawnow;
U(:,1)=u;

% mass
ma = fft(abs(u).^2);
ma0 = ma(1);

% solve pde and plot results
for n =2:Nt+1

    vna=exp(0.5*1i*dt*k2x).*v;
    una=ifft(vna);
    pot=2*(una.*conj(una));
    unb=exp(-1i*Es*dt*pot).*una;
    vnb=fft(unb);
    v=exp(0.5*1i*dt*k2x).*vnb;
    t=(n-1)*dt;

    if (mod(n,10)==0)
        tdata(n/10)=t;
        u=ifft(v);
        U(:,n/10)=u;
        uexact=4*exp(1i*t)*(cosh(3*x)+3*exp(8*1i*t)*cosh(x))...
            ./(cosh(4*x)+4*cosh(2*x)+3*cos(8*t));
        figure(1); clf; plot(x,abs(u).^2); ...
            xlim([-0.5,0.5]); title(num2str(t));
        figure(2); clf; plot(x,abs(u-uexact).^2);...
            xlim([-0.5,0.5]); title(num2str(t));
        drawnow;
        ma = fft(abs(u).^2);
        ma = ma(1);
        test = log10(abs(1-ma/ma0))
    end
end
figure(3); clf; mesh(tdata(1:(n-1)/10),x,abs(U(:,1:(n-1)/10)).^2);
```

Listing 12.3: A Matlab program which uses Strang splitting to solve the two dimensional nonlinear Schrödinger equation.

```
% A program to solve the 2D nonlinear Schr\"{o}dinger equation using a
% splitting method

clear all; format compact; format short;
set(0,'defaultaxesfontsize',30,'defaultaxeslinewidth',.7,...
    'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,'
        defaultaxesfontweight','bold')

% set up grid
tic
Lx = 20;          % period  2*pi*L
Ly = 20;          % period  2*pi*L
Nx = 2*256;       % number of harmonics
Ny = 2*256;       % number of harmonics
Nt = 100;         % number of time slices
dt = 5.0/Nt;      % time step

Es = 1.0;

% initialise variables
x = (2*pi/Nx)*(-Nx/2:Nx/2 -1)'*Lx;           % x coordinate
kx = 1i*[0:Nx/2-1 0 -Nx/2+1:-1]'/Lx;         % wave vector
y = (2*pi/Ny)*(-Ny/2:Ny/2 -1)'*Ly;           % y coordinate
ky = 1i*[0:Ny/2-1 0 -Ny/2+1:-1]'/Ly;         % wave vector
[xx,yy]=meshgrid(x,y);
[k2xm,k2ym]=meshgrid(kx.^2,ky.^2);
% initial conditions
u = exp(-(xx.^2+yy.^2));
v=fft2(u);
figure(1); clf; mesh(xx,yy,u); drawnow;
t=0; tdata(1)=t;

% mass
ma = fft2(abs(u).^2);
ma0 = ma(1,1);

% solve pde and plot results
for n =2:Nt+1
    vna=exp(0.5*1i*dt*(k2xm + k2ym)).*v;
    una=ifft2(vna);
    pot=Es*((abs(una)).^2);
    unb=exp(-1i*dt*pot).*una;
    vnb=fft2(unb);
    v=exp(0.5*1i*dt*(k2xm + k2ym)).*vnb;
    u=ifft2(v);
    t=(n-1)*dt;
    tdata(n)=t;
     if (mod(n,10)==0)
         figure(2); clf; mesh(xx,yy,abs(u).^2); title(num2str(t));
        drawnow;
        ma = fft2(abs(u).^2);
        ma = ma(1,1);
        test = log10(abs(1-ma/ma0))
    end
end
figure(4); clf; mesh(xx,yy,abs(u).^2);
toc
```

Listing 12.4: A Matlab program which uses Strang splitting to solve the three dimensional nonlinear Schrödinger equation.

```
% A program to solve the 3D nonlinear Schr\"{o}dinger equation using a
% splitting method

clear all; format compact; format short;
set(0,'defaultaxesfontsize',30,'defaultaxeslinewidth',.7,...
    'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,...
    'defaultaxesfontweight','bold')

% set up grid
tic
Lx = 4;          % period  2*pi*L
Ly = 4;          % period  2*pi*L
Lz = 4;          % period  2*pi*L
Nx = 64;         % number of harmonics
Ny = 64;         % number of harmonics
Nz = 64;         % number of harmonics
Nt = 100;        % number of time slices
dt = 1.0/Nt;     % time step

Es = 1.0;  % focusing or defocusing parameter

% initialise variables
x = (2*pi/Nx)*(-Nx/2:Nx/2 -1)'*Lx;          % x coordinate
kx = 1i*[0:Nx/2-1 0 -Nx/2+1:-1]'/Lx;        % wave vector
y = (2*pi/Ny)*(-Ny/2:Ny/2 -1)'*Ly;          % y coordinate
ky = 1i*[0:Ny/2-1 0 -Ny/2+1:-1]'/Ly;        % wave vector
z = (2*pi/Nz)*(-Nz/2:Nz/2 -1)'*Lz;          % y coordinate
kz = 1i*[0:Nz/2-1 0 -Nz/2+1:-1]'/Lz;        % wave vector
[xx,yy,zz]=meshgrid(x,y,z);
[k2xm,k2ym,k2zm]=meshgrid(kx.^2,ky.^2,kz.^2);

% initial conditions
u = exp(-(xx.^2+yy.^2+zz.^2));
v=fftn(u);
figure(1); clf; UP = abs(u).^2;
p1 = patch(isosurface(x,y,z,UP,.0025),...
    'FaceColor','yellow','EdgeColor','none');
p2 = patch(isocaps(x,y,z,UP,.0025),...
    'FaceColor','interp','EdgeColor','none');
isonormals(UP,p1); lighting phong;
xlabel('x'); ylabel('y'); zlabel('z');
axis equal; axis square; view(3); drawnow;
t=0; tdata(1)=t;

% mass
ma = fftn(abs(u).^2);
ma0 = ma(1,1,1);

% solve pde and plot results

for n =2:Nt+1
    vna=exp(0.5*1i*dt*(k2xm + k2ym + k2zm)).*v;
    una=ifftn(vna);
    pot=Es*((abs(una)).^2);
    unb=exp(-1i*dt*pot).*una;
    vnb=fftn(unb);
    v=exp(0.5*1i*dt*(k2xm + k2ym + k2zm)).*vnb;
    u=ifftn(v);
    t=(n-1)*dt;
    tdata(n)=t;
    if (mod(n,10)==0)
        figure(1); clf; UP = abs(u).^2;
        p1 = patch(isosurface(x,y,z,UP,.0025),...
            'FaceColor','yellow','EdgeColor','none');
        p2 = patch(isocaps(x,y,z,UP,.0025),...
            'FaceColor','interp','EdgeColor','none');
        isonormals(UP,p1); lighting phong;
        xlabel('x'); ylabel('y'); zlabel('z');
        axis equal; axis square; view(3); drawnow;
        ma = fftn(abs(u).^2);
        ma = ma(1,1,1);  test = log10(abs(1-ma/ma0))
     end
end
figure(4); clf; UP = abs(u).^2;
p1 = patch(isosurface(x,y,z,UP,.0025),...
    'FaceColor','yellow','EdgeColor','none');
p2 = patch(isocaps(x,y,z,UP,.0025),...
    'FaceColor','interp','EdgeColor','none');
isonormals(UP,p1); lighting phong;
xlabel('x'); ylabel('y'); zlabel('z');
axis equal; axis square;  view(3); drawnow;
toc
```

12.5 Example One-Dimensional Fortran Program for the Nonlinear Schrödinger Equation

Before considering parallel programs, we need to understand how to write a Fortran code for the one-dimensional nonlinear Schrödinger equation. Below is an example Fortran program followed by a Matlab plotting script to visualize the results. In compiling the Fortran program a standard Fortran compiler and the FFTW library are required. Since the commands required for this are similar to those in the makefile for the heat equation, we do not include them here.

Listing 12.5: A Fortran program to solve the 1D nonlinear Schrödinger equation using splitting.

```
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This program solves nonlinear Schrodinger equation in 1 dimension
! i*u_t+Es*|u|^2u+u_{xx}=0
! using a second order time spectral splitting scheme
!
! The boundary conditions are u(0)=u(2*L*\pi)
! The initial condition is u=exp(-x^2)
!
! .. Parameters ..
!  Nx       = number of modes in x - power of 2 for FFT
!  Nt       = number of timesteps to take
!  Tmax       = maximum simulation time
!  plotgap      = number of timesteps between plots
!  FFTW_IN_PLACE  = value for FFTW input
!  FFTW_MEASURE   = value for FFTW input
!  FFTW_EXHAUSTIVE  = value for FFTW input
!  FFTW_PATIENT   = value for FFTW input
!  FFTW_ESTIMATE  = value for FFTW input
!  FFTW_FORWARD     = value for FFTW input
!  FFTW_BACKWARD  = value for FFTW input
!  pi = 3.14159265358979323846264338327950288419716939937510d0
!  L        = width of box
!  ES       = +1 for focusing and -1 for defocusing
! .. Scalars ..
!  i        = loop counter in x direction
!  n        = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  start      = variable to record start time of program
!  finish     = variable to record end time of program
!  count_rate   = variable for clock count rate
!  planfx     = Forward 1d fft plan in x
!  planbx     = Backward 1d fft plan in x
!  dt       = timestep
! .. Arrays ..
!  u        = approximate solution
!  v        = Fourier transform of approximate solution
! .. Vectors ..
!  una        = temporary field
!  unb        = temporary field
!  vna        = temporary field
!  pot        = potential
!  kx       = fourier frequencies in x direction
!  x        = x locations
!  time       = times at which save data
!  name_config    = array to store filename for data to be saved
!  fftfx      = array to setup x Fourier transform
!  fftbx      = array to setup x Fourier transform
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
! FFTW3  -- Fast Fourier Transform in the West Library
!     (http://www.fftw.org/)


PROGRAM main

! Declare variables
IMPLICIT NONE
INTEGER(kind=4), PARAMETER  :: Nx=8*256
INTEGER(kind=4), PARAMETER  :: Nt=200
REAL(kind=8), PARAMETER   &
  ::  pi=3.14159265358979323846264338327950288419716939937510d0
REAL(kind=8), PARAMETER :: L=5.0d0
REAL(kind=8), PARAMETER :: Es=1.0d0
REAL(kind=8)  :: dt=2.0d0/Nt
COMPLEX(kind=8),  DIMENSION(:), ALLOCATABLE :: kx
REAL(kind=8),  DIMENSION(:), ALLOCATABLE  :: x
COMPLEX(kind=8),  DIMENSION(:,:), ALLOCATABLE :: u
COMPLEX(kind=8),  DIMENSION(:,:), ALLOCATABLE :: v
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  :: una,vn
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  :: unb,pot
REAL(kind=8), DIMENSION(:), ALLOCATABLE   :: time
INTEGER(kind=4) :: i,j,k,n,modes,AllocateStatus
INTEGER(kind=4) :: start, finish, count_rate
INTEGER(kind=4), PARAMETER  :: FFTW_IN_PLACE = 8, FFTW_MEASURE = 0, &
  FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32, FFTW_ESTIMATE = 64
INTEGER(kind=4), PARAMETER  :: FFTW_FORWARD = -1, FFTW_BACKWARD=1
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  :: fftfx,fftbx
INTEGER(kind=8) :: planfx,planbx
  CHARACTER*100 :: name_config

CALL system_clock(start,count_rate)
ALLOCATE(kx(1:Nx),x(1:Nx),u(1:Nx,1:Nt+1),v(1:Nx,1:Nt+1),&
  una(1:Nx),vn(1:Nx),unb(1:Nx),pot(1:Nx),time(1:Nt+1),&
  fftfx(1:Nx),fftbx(1:Nx),stat=AllocateStatus)
IF (allocatestatus .ne. 0) STOP
! set up ffts
CALL dfftw_plan_dft_1d_(planfx,Nx,fftfx(1:Nx),fftbx(1:Nx),&
  FFTW_FORWARD,FFTW_PATIENT)
CALL dfftw_plan_dft_1d_(planbx,Nx,fftbx(1:Nx),fftfx(1:Nx),&
  FFTW_BACKWARD,FFTW_PATIENT)
PRINT *,'Setup FFTs'
  ! setup fourier frequencies
DO i=1,1+Nx/2
  kx(i)= cmplx(0.0d0,1.0d0)*(i-1.0d0)/L
END DO
kx(1+Nx/2)=0.0d0
DO i = 1,Nx/2 -1
  kx(i+1+Nx/2)=-kx(1-i+Nx/2)
END DO
DO i=1,Nx
  x(i)=(-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)))*pi*L
END DO
PRINT *,'Setup grid and fourier frequencies'

DO i=1,Nx
  u(i,1)=exp(-1.0d0*(x(i)**2))
END DO
! transform initial data
CALL dfftw_execute_dft_(planfx,u(1:Nx,1),v(1:Nx,1))
PRINT *,'Got initial data, starting timestepping'
time(1)=0.0d0
DO n=1,Nt
  time(n+1)=n*dt
  DO i=1,Nx
    vn(i)=exp(0.5d0*dt*kx(i)*kx(i)*cmplx(0.0d0,1.0d0))*v(i,n)
  END DO
  CALL dfftw_execute_dft_(planbx,vn(1:Nx),una(1:Nx))
  ! normalize
  DO i=1,Nx
    una(i)=una(1:Nx)/REAL(Nx,kind(0d0))
```

```
      pot(i)=Es*una(i)*conjg(una(i))
      unb(i)=exp(cmplx(0.0d0,-1.0d0)*dt*pot(i))*una(i)
    END DO
    CALL dfftw_execute_dft_(planfx,unb(1:Nx),vn(1:Nx))
    DO i=1,Nx
      v(i,n+1)=exp(0.50d0*dt*kx(i)*kx(i)*cmplx(0.0d0,1.0d0))*vn(i)
    END DO
    CALL dfftw_execute_dft_(planbx,v(1:Nx,n+1),u(1:Nx,n+1))
    ! normalize
    DO i=1,Nx
      u(i,n+1)=u(i,n+1)/REAL(Nx,kind(0d0))
    END DO
  END DO
  PRINT *,'Finished time stepping'
  CALL system_clock(finish,count_rate)
  PRINT*,'Program took ',&
    REAL(finish-start,kind(0d0))/REAL(count_rate,kind(0d0)),'for execution
      '

  name_config = 'u.dat'
  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
  REWIND(11)
  DO j=1,Nt
    DO i=1,Nx
      WRITE(11,*) abs(u(i,j))**2
    END DO
  END DO
  CLOSE(11)

  name_config = 'tdata.dat'
  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
  REWIND(11)
  DO j=1,Nt
    WRITE(11,*) time(j)
  END DO
  CLOSE(11)

  name_config = 'xcoord.dat'
  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
  REWIND(11)
  DO i=1,Nx
    WRITE(11,*) x(i)
  END DO
  CLOSE(11)

  PRINT *,'Saved data'

  CALL dfftw_destroy_plan_(planbx)
  CALL dfftw_destroy_plan_(planfx)
  CALL dfftw_cleanup_()

```

```
DEALLOCATE(kx,x,u,v,una,vn,unb,&
      pot,time,fftfx,fftbx,&
        stat=AllocateStatus)
IF (allocatestatus .ne. 0) STOP
PRINT *,'deallocated memory'
PRINT *,'Program execution complete'
END PROGRAM main
```

Listing 12.6: A Matlab program which plots a numerical solution to a 1D nonlinear Schrödinger equation generated by listing 12.5.

```
% A program to plot the computed results

clear all; format compact, format short,
set(0,'defaultaxesfontsize',18,'defaultaxeslinewidth',.9,...
    'defaultlinelinewidth',3.5,'defaultpatchlinewidth',5.5);

% Load data
load('./u.dat');
load('./tdata.dat');
load('./xcoord.dat');
Tsteps = length(tdata);

Nx = length(xcoord); Nt = length(tdata);

u = reshape(u,Nx,Nt);

% Plot data
figure(3); clf; mesh(tdata,xcoord,u); xlabel t; ylabel x; zlabel('|u|^2');
```

12.6 Shared Memory Parallel: OpenMP

We recall that OpenMP is a set of compiler directives that can allow one to easily make a Fortran, C or C++ program run on a shared memory machine – that is a computer for which all compute processes can access the same globally addressed memory space. It allows for easy parallelization of serial programs which have already been written in one of the aforementioned languages.

We will demonstrate one form of parallelizm for the two dimensional nonlinear Schrödinger equation in which we will parallelize the loops using OpenMP commands, but will use the threaded FFTW library to parallelize the transforms for us. The example programs are in listing 12.7, A second method to parallelize the loops and Fast Fourier transforms explicitly using OpenMP commands is outlined in the exercises.

Listing 12.7: An OpenMP Fortran program to solve the 2D nonlinear Schrödinger equation

using splitting and threaded FFTW.

```
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This program solves nonlinear Schrodinger equation in 2 dimensions
! i*u_t+Es*|u|^2u+u_{xx}+u_{yy}=0
! using a second order time spectral splitting scheme
!
! The boundary conditions are u(x=0,y)=u(2*Lx*\pi,y),
! u(x,y=0)=u(x,y=2*Ly*\pi)
! The initial condition is u=exp(-x^2-y^2)
!
! .. Parameters ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  Nt        = number of timesteps to take
!  Tmax        = maximum simulation time
!  plotgap       = number of timesteps between plots
!  FFTW_IN_PLACE  = value for FFTW input
!  FFTW_MEASURE   = value for FFTW input
!  FFTW_EXHAUSTIVE  = value for FFTW input
!  FFTW_PATIENT   = value for FFTW input
!  FFTW_ESTIMATE  = value for FFTW input
!  FFTW_FORWARD     = value for FFTW input
!  FFTW_BACKWARD  = value for FFTW input
!  pi = 3.14159265358979323846264338327950288419716939937510d0
!  Lx        = width of box in x direction
!  Ly        = width of box in y direction
!  ES        = +1 for focusing and -1 for defocusing
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!  n         = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  numthreads    = number of openmp threads
!  ierr        = error return code
!  start       = variable to record start time of program
!  finish      = variable to record end time of program
!  count_rate    = variable for clock count rate
!  planfx      = Forward 1d fft plan in x
!  planbx      = Backward 1d fft plan in x
!  planfy      = Forward 1d fft plan in y
!  planby      = Backward 1d fft plan in y
!  dt        = timestep
! .. Arrays ..
!  u         = approximate solution
!  v         = Fourier transform of approximate solution
!  unax        = temporary field
!  vnax        = temporary field
!  vnbx        = temporary field
!  vnay        = temporary field
!  vnby        = temporary field
!  potx        = potential
! .. Vectors ..
!  kx        = fourier frequencies in x direction
!  ky        = fourier frequencies in y direction
!  x         = x locations
!  y         = y locations
!  time        = times at which save data
!  name_config    = array to store filename for data to be saved
!  fftfx       = array to setup x Fourier transform
!  fftbx       = array to setup x Fourier transform
!  fftfy       = array to setup y Fourier transform
!  fftby       = array to setup y Fourier transform
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
! FFTW3  -- Fast Fourier Transform in the West Library
!     (http://www.fftw.org/)
! OpenMP library
PROGRAM main
USE omp_lib
IMPLICIT NONE
! Declare variables
INTEGER(kind=4), PARAMETER  ::  Nx=1024
INTEGER(kind=4), PARAMETER  ::  Ny=1024
INTEGER(kind=4), PARAMETER  ::  Nt=20
INTEGER(kind=4), PARAMETER  ::  plotgap=5
REAL(kind=8), PARAMETER   ::  &
pi=3.14159265358979323846264338327950288419716939937510d0
REAL(kind=8), PARAMETER   ::  Lx=2.0d0
REAL(kind=8), PARAMETER   ::  Ly=2.0d0
REAL(kind=8), PARAMETER   ::  Es=1.0d0
REAL(kind=8)        ::  dt=0.10d0/Nt
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  kx
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  ky
REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  x
REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  y
COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  unax,vnax,vnbx,potx
COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  vnay,vnby
REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  time
INTEGER(kind=4)        ::  i,j,k,n,allocatestatus,ierr
INTEGER(kind=4)        ::  start, finish, count_rate, numthreads
INTEGER(kind=8), PARAMETER  ::  FFTW_IN_PLACE=8, FFTW_MEASURE=0,&
                FFTW_EXHAUSTIVE=8, FFTW_PATIENT=32,&
                        FFTW_ESTIMATE=64
INTEGER(kind=8),PARAMETER   ::  FFTW_FORWARD=-1, FFTW_BACKWARD=1
INTEGER(kind=8)        ::  planfxy,planbxy
  CHARACTER*100        ::  name_config,number_file

numthreads=omp_get_max_threads()
PRINT *,'There are ',numthreads,' threads.'

ALLOCATE(kx(1:Nx),ky(1:Nx),x(1:Nx),y(1:Nx),unax(1:Nx,1:Ny),&
    vnax(1:Nx,1:Ny),potx(1:Nx,1:Ny),time(1:1+Nt/plotgap),&
    stat=allocatestatus)
IF (allocatestatus .ne. 0) stop
PRINT *,'allocated memory'

! set up multithreaded ffts
CALL dfftw_init_threads_(ierr)
PRINT *,'Initiated threaded FFTW'
CALL dfftw_plan_with_nthreads_(numthreads)
PRINT *,'Inidicated number of threads to be used in planning'
CALL dfftw_plan_dft_2d_(planfxy,Nx,Ny,unax(1:Nx,1:Ny),vnax(1:Nx,1:Ny),&
            FFTW_FORWARD,FFTW_ESTIMATE)
CALL dfftw_plan_dft_2d_(planbxy,Nx,Ny,vnax(1:Nx,1:Ny),unax(1:Nx,1:Ny),&
            FFTW_BACKWARD,FFTW_ESTIMATE)
PRINT *,'Setup FFTs'

! setup fourier frequencies
!$OMP PARALLEL PRIVATE(i,j)
!$OMP DO SCHEDULE(static)
DO i=1,1+Nx/2
  kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
END DO
!$OMP END DO
kx(1+Nx/2)=0.0d0
!$OMP DO SCHEDULE(static)
DO i = 1,Nx/2 -1
  kx(i+1+Nx/2)=-kx(1-i+Nx/2)
END DO
!$OMP END DO
!$OMP DO SCHEDULE(static)
  DO i=1,Nx
  x(i)=(-1.0d0+2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)) )*pi*Lx
END DO
!$OMP END DO
```

```fortran
!$OMP DO SCHEDULE(static)
DO j=1,1+Ny/2
  ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
END DO
!$OMP END DO
ky(1+Ny/2)=0.0d0
!$OMP DO SCHEDULE(static)
DO j = 1,Ny/2 -1
  ky(j+1+Ny/2)=-ky(1-j+Ny/2)
END DO
!$OMP END DO
!$OMP DO SCHEDULE(static)
  DO j=1,Ny
  y(j)=(-1.0d0+2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)) )*pi*Ly
END DO
!$OMP END DO
PRINT *,'Setup grid and fourier frequencies'
!$OMP DO SCHEDULE(static)
DO j=1,Ny
  unax(1:Nx,j)=exp(-1.0d0*(x(1:Nx)**2 +y(j)**2))
END DO
!$OMP END DO
!$OMP END PARALLEL
name_config = 'uinitial.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,Ny
  DO i=1,Nx
    WRITE(11,*) abs(unax(i,j))**2
  END DO
END DO
CLOSE(11)
! transform initial data and do first half time step
CALL dfftw_execute_dft_(planfxy,unax(1:Nx,1:Ny),vnax(1:Nx,1:Ny))

PRINT *,'Got initial data, starting timestepping'
time(1)=0.0d0
CALL system_clock(start,count_rate)
DO n=1,Nt
  !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
  DO j=1,Ny
    DO i=1,Nx
      vnax(i,j)=exp(0.5d0*dt*(kx(i)*kx(i) + ky(j)*ky(j))&
        *cmplx(0.0d0,1.0d0))*vnax(i,j)
    END DO
  END DO
  !$OMP END PARALLEL DO
  CALL dfftw_execute_dft_(planbxy,vnax(1:Nx,1:Ny),unax(1:Nx,1:Ny))
  !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
  DO j=1,Ny
    DO i=1,Nx
```

```
        unax(i,j)=unax(i,j)/REAL(Nx*Ny,kind(0d0))
        potx(i,j)=Es*unax(i,j)*conjg(unax(i,j))
        unax(i,j)=exp(cmplx(0.0d0,-1.0d0)*dt*potx(i,j))&
            *unax(i,j)
      END DO
    END DO
    !$OMP END PARALLEL DO
    CALL dfftw_execute_dft_(planfxy,unax(1:Nx,1:Ny),vnax(1:Nx,1:Ny))
    !$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
    DO j=1,Ny
      DO i=1,Nx
        vnax(i,j)=exp(0.5d0*dt*(kx(i)*kx(i) + ky(j)*ky(j))&
            *cmplx(0.0d0,1.0d0))*vnax(i,j)
      END DO
    END DO
    !$OMP END PARALLEL DO
    IF (mod(n,plotgap)==0) then
      time(1+n/plotgap)=n*dt
      PRINT *,'time',n*dt
      CALL dfftw_execute_dft_(planbxy,vnax(1:Nx,1:Ny),unax(1:Nx,1:Ny))
      !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
      DO j=1,Ny
        DO i=1,Nx
          unax(i,j)=unax(i,j)/REAL(Nx*Ny,kind(0d0))
        END DO
      END DO
      !$OMP END PARALLEL DO
      name_config='./data/u'
      WRITE(number_file,'(i0)') 10000000+1+n/plotgap
      ind=index(name_config,' ') -1
      name_config=name_config(1:ind)//numberfile
      ind=index(name_config,' ') -1
      name_config=name_config(1:ind)//'.dat'
      OPEN(unit=11,FILE=name_config,status="UNKNOWN")
      REWIND(11)
      DO j=1,Ny
        DO i=1,Nx
          WRITE(11,*) abs(unax(i,j))**2
        END DO
      END DO
      CLOSE(11)
    END IF
  END DO
  PRINT *,'Finished time stepping'
  CALL system_clock(finish,count_rate)
  PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),&
    'for Time stepping'


  name_config = 'tdata.dat'
  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
```

```fortran
REWIND(11)
DO j=1,1+Nt/plotgap
  WRITE(11,*) time(j)
END DO
CLOSE(11)

name_config = 'xcoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO i=1,Nx
  WRITE(11,*) x(i)
END DO
CLOSE(11)

name_config = 'ycoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,Ny
  WRITE(11,*) y(j)
END DO
CLOSE(11)
PRINT *,'Saved data'

CALL dfftw_destroy_plan_(planbxy)
CALL dfftw_destroy_plan_(planfxy)
CALL dfftw_cleanup_threads_()

DEALLOCATE(unax,vnax,potx,stat=allocatestatus)
IF (allocatestatus .ne. 0) STOP
PRINT *,'Deallocated memory'

  PRINT *,'Program execution complete'
END PROGRAM main
```

Listing 12.8: An example makefile for compiling the OpenMP program in listing 12.7. The example assumes one is using Flux and has loaded environments for the GCC compiler as well as the GCC compiled version of FFTW. To use the Intel compiler to with this code, the OMP stack size needs to be explicitly set to be large enough. If one is using the the PGI compilers instead of the GCC compilers, change the flag $-fopenmp$ to $-mp$.

```makefile
#define the complier
COMPILER = gfortran
# compilation settings, optimization, precision, parallelization
FLAGS = -O3  -fopenmp



# libraries
LIBS = -L/usr/local/lib -lfftw3 -lm
# source list for main program
```

```
SOURCES =  NLSsplitting.f90

test: $(SOURCES)
    ${COMPILER} -o NLSsplitting $(FLAGS) $(SOURCES) $(LIBS)

clean:
  rm *.o

clobber:
  rm NLSsplitting
```

Listing 12.9: A Matlab program which plots a numerical solution to a 2D nonlinear Schrödinger equation generated by listing 12.7 or 12.11.

```
% A program to plot the computed results for the 2D NLS equation

clear all; format compact, format short,
set(0,'defaultaxesfontsize',18,'defaultaxeslinewidth',.9,...
    'defaultlinelinewidth',3.5,'defaultpatchlinewidth',5.5);

% Load data
load('./ufinal.dat');
load('./tdata.dat');
load('./ycoord.dat');
load('./xcoord.dat');

Ny = length(ycoord); Nx = length(xcoord); Nt = length(tdata);

ufinal = reshape(ufinal,Nx,Ny);

% Plot data
figure(3); clf; mesh(xcoord,ycoord,ufinal); xlabel x; ylabel y; zlabel('|u
    |^2');
```

Listing 12.10: An example submission script for use on Flux. Change your_username appropriately.

```
#!/bin/bash
#PBS -N NLS
#PBS -l nodes=1:ppn=2,walltime=00:03:00
#PBS -q flux
#PBS -l qos=math471f11_flux
#PBS -A math471f11_flux
#PBS -M your_username@umich.edu
#PBS -m abe
#PBS -V
#
# Create a local directory to run and copy your files to local.
# Let PBS handle your output
```

```
cp ${HOME}/parallelspectralintro/NLSsplitting /nobackup/your_username/
   NLSsplitting
cd /nobackup/your_username

export OMP_NUM_THREADS=2
./NLSsplitting

#Clean up your files
```

12.7 Exercises

1) Download the example Matlab programs which accompany the pre-print by Klein, Muite and Roidot [32]. Examine how the mass and energy for these Schrödinger like equations are computed. Add code to check conservation of mass and energy to the Matlab programs for the nonlinear Schrödinger equation.

2) The Gross-Pitaevskii equation[4] is given by

$$i\psi_t + |\psi|^2\psi + V(\boldsymbol{x})\psi = 0 \tag{12.11}$$

where we will take

$$V(\boldsymbol{x}) = \|\boldsymbol{x}\|_{l^2}^2 = \sum_{k=1}^{N} x_k^2 \tag{12.12}$$

in which N is the space dimension. Show that this equation can be solved by splitting it into

$$i\psi_t + \Delta\psi = 0 \tag{12.13}$$

and

$$i\psi_t + |\psi|^2\psi + V(\boldsymbol{x})\psi = 0. \tag{12.14}$$

Be sure to explain how eqs. (12.13),(12.14) are solved.

3) Modify the Matlab codes to solve the Gross-Pitaevskii equation in one, two and three dimensions.

4) Modify the serial Fortran codes to solve the Gross-Pitaevskii equation in one, two and three dimensions.

5) Listings 12.11 and 12.12 give an alternate method of parallelizing an OpenMP program. Make the program in listing 12.7 as efficient as possible and as similar to that in 12.11, but without changing the parallelization strategy. Compare the speed of the two different programs. Try to vary the number of grid points and cores used. Which code is faster on your system? Why do you think this is?

[4] http://en.wikipedia.org/wiki/Gross%E2%80%93Pitaevskii_equation

Listing 12.11: An OpenMP Fortran program to solve the 2D nonlinear Schrödinger equation using splitting.

```
!
    --------------------------------------------------------------------

!
!
! PURPOSE
!
! This program solves nonlinear Schrodinger equation in 2
    dimensions
! i*u_t+Es*|u|^2u+u_{xx}+u_{yy}=0
! using a second order time spectral splitting scheme
!
! The boundary conditions are u(x=0,y)=u(2*Lx*\pi,y),
! u(x,y=0)=u(x,y=2*Ly*\pi)
! The initial condition is u=exp(-x^2-y^2)
!
! .. Parameters ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  Nt        = number of timesteps to take
!  Tmax        = maximum simulation time
!  plotgap       = number of timesteps between plots
!  FFTW_IN_PLACE  = value for FFTW input
!  FFTW_MEASURE   = value for FFTW input
!  FFTW_EXHAUSTIVE  = value for FFTW input
!  FFTW_PATIENT   = value for FFTW input
!  FFTW_ESTIMATE  = value for FFTW input
!  FFTW_FORWARD      = value for FFTW input
!  FFTW_BACKWARD  = value for FFTW input
!  pi = 3.14159265358979323846264338327950288419716939937510d0
!  Lx        = width of box in x direction
!  Ly        = width of box in y direction
!  ES        = +1 for focusing and -1 for defocusing
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!  n         = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  start      = variable to record start time of program
!  finish     = variable to record end time of program
!  count_rate   = variable for clock count rate
!  planfx     = Forward 1d fft plan in x
!  planbx     = Backward 1d fft plan in x
!  planfy     = Forward 1d fft plan in y
!  planby     = Backward 1d fft plan in y
!  dt        = timestep
! .. Arrays ..
!  u         = approximate solution
!  v         = Fourier transform of approximate solution
!  unax        = temporary field
!  vnax        = temporary field
!  vnbx        = temporary field
!  vnay        = temporary field
!  vnby        = temporary field
!  potx        = potential
! .. Vectors ..
!  kx        = fourier frequencies in x direction
!  ky        = fourier frequencies in y direction
!  x         = x locations
!  y         = y locations
!  time        = times at which save data
!  name_config     = array to store filename for data to be saved
!  fftfx       = array to setup x Fourier transform
!  fftbx       = array to setup x Fourier transform
!  fftfy       = array to setup y Fourier transform
!  fftby       = array to setup y Fourier transform
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!
   ----------------------------------------------------------------------
! External routines required
!
! External libraries required
! FFTW3  -- Fast Fourier Transform in the West Library
!     (http://www.fftw.org/)
! OpenMP library

PROGRAM main
USE omp_lib
IMPLICIT NONE
! Declare variables
INTEGER(kind=4), PARAMETER  ::  Nx=2**8
INTEGER(kind=4), PARAMETER  ::  Ny=2**8
INTEGER(kind=4), PARAMETER  ::  Nt=20
INTEGER(kind=4), PARAMETER  ::  plotgap=5
REAL(kind=8), PARAMETER   :: &
  pi=3.14159265358979323846264338327950288419716939937510d0
REAL(kind=8), PARAMETER   ::  Lx=2.0d0
REAL(kind=8), PARAMETER   ::  Ly=2.0d0
```

```fortran
REAL(kind=8), PARAMETER   ::  Es=0.0d0
REAL(kind=8)         ::  dt=0.10d0/Nt
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  kx,ky
REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  x,y
COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  unax,vnax,vnbx,potx
COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  vnay,vnby
REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  time
INTEGER(kind=4)       ::  i,j,k,n,allocatestatus
INTEGER(kind=4)       ::  start, finish, count_rate
INTEGER(kind=8), PARAMETER  ::  FFTW_IN_PLACE=8, FFTW_MEASURE=0,&
                FFTW_EXHAUSTIVE=8, FFTW_PATIENT=32,&
                          FFTW_ESTIMATE=64
INTEGER(kind=8),PARAMETER   ::  FFTW_FORWARD=-1, FFTW_BACKWARD=1
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE :: fftfx,fftbx,fftfy,
   fftby
INTEGER(kind=8)       ::  planfx,planbx,planfy,planby
  CHARACTER*100       ::  name_config

ALLOCATE(kx(1:Nx),ky(1:Nx),x(1:Nx),y(1:Nx),unax(1:Nx,1:Ny),&
    vnax(1:Nx,1:Ny),vnbx(1:Nx,1:Ny),potx(1:Nx,1:Ny),fftfx(1:Nx),&
    fftbx(1:Nx),fftfy(1:Nx),fftby(1:Nx),vnay(1:Ny,1:Nx),&
    vnby(1:Ny,1:Nx),time(1:1+Nt/plotgap),stat=allocatestatus)
IF (allocatestatus .ne. 0) stop
PRINT *,'allocated memory'
  ! set up ffts
CALL dfftw_plan_dft_1d_(planfx,Nx,fftfx(1:Nx),fftbx(1:Nx),&
    FFTW_FORWARD,FFTW_ESTIMATE)
CALL dfftw_plan_dft_1d_(planbx,Nx,fftbx(1:Nx),fftfx(1:Nx),&
    FFTW_BACKWARD,FFTW_ESTIMATE)
CALL dfftw_plan_dft_1d_(planfy,Ny,fftfy(1:Ny),fftby(1:Ny),&
    FFTW_FORWARD,FFTW_ESTIMATE)
CALL dfftw_plan_dft_1d_(planby,Ny,fftby(1:Ny),fftfy(1:Ny),&
    FFTW_BACKWARD,FFTW_ESTIMATE)
PRINT *,'Setup FFTs'

! setup fourier frequencies
!$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
DO i=1,1+Nx/2
  kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
END DO
!$OMP END PARALLEL DO
kx(1+Nx/2)=0.0d0
!$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
DO i = 1,Nx/2 -1
  kx(i+1+Nx/2)=-kx(1-i+Nx/2)
END DO
!$OMP END PARALLEL DO
!$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
  DO i=1,Nx
  x(i)=(-1.0d0+2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)) )*pi*Lx
END DO
```

```fortran
!$OMP END PARALLEL DO
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,1+Ny/2
  ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
END DO
!$OMP END PARALLEL DO
ky(1+Ny/2)=0.0d0
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j = 1,Ny/2 -1
  ky(j+1+Ny/2)=-ky(1-j+Ny/2)
END DO
!$OMP END PARALLEL DO
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
  DO j=1,Ny
  y(j)=(-1.0d0+2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)) )*pi*Ly
END DO
!$OMP END PARALLEL DO
PRINT *,'Setup grid and fourier frequencies'
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,Ny
  DO i=1,Nx
    unax(i,j)=exp(-1.0d0*(x(i)**2 +y(j)**2))
  END DO
END DO
!$OMP END PARALLEL DO
name_config = 'uinitial.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,Ny
  DO i=1,Nx
    WRITE(11,*) abs(unax(i,j))**2
  END DO
END DO
CLOSE(11)
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,Ny
  DO i=1,Nx
    CALL dfftw_execute_dft_(planfx,unax(i,j),vnax(i,j))
  END DO
END DO
!$OMP END PARALLEL DO
vnay(1:Ny,1:Nx)=TRANSPOSE(vnax(1:Nx,1:Ny))
! transform initial data and do first half time step
!$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
DO i=1,Nx
  CALL dfftw_execute_dft_(planfy,vnay(1:Ny,i),vnby(1:Ny,i))
  DO j=1,Ny
    vnby(j,i)=exp(0.5d0*dt*(kx(i)*kx(i) + ky(j)*ky(j))&
        *cmplx(0.0d0,1.0d0))*vnby(j,i)
  END DO
  CALL dfftw_execute_dft_(planby,vnby(j,i),vnay(j,i))
```

```
END DO
!$OMP END PARALLEL DO
PRINT *,'Got initial data, starting timestepping'
time(1)=0.0d0
CALL system_clock(start,count_rate)
DO n=1,Nt
  vnbx(1:Nx,1:Ny)=TRANSPOSE(vnay(1:Ny,1:Nx))/REAL(Ny,kind(0d0))
  !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
  DO j=1,Ny
    CALL dfftw_execute_dft_(planbx,vnbx(1:Nx,j),unax(1:Nx,j))
    DO i=1,Nx
      unax(i,j)=unax(1:Nx,j)/REAL(Nx,kind(0d0))
      potx(i,j)=Es*unax(i,j)*conjg(unax(i,j))
      unax(i,j)=exp(cmplx(0.0d0,-1.0d0)*dt*potx(i,j))&
            *unax(i,j)
    END DO
    CALL dfftw_execute_dft_(planfx,unax(1:Nx,j),vnax(1:Nx,j))
  END DO
  !$OMP END PARALLEL DO
  vnby(1:Ny,1:Nx)=TRANSPOSE(vnax(1:Nx,1:Ny))
  !$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
  DO i=1,Nx
    CALL dfftw_execute_dft_(planfy,vnby(1:Ny,i),vnay(1:Ny,i))
    DO j=1,Ny
      vnby(j,i)=exp(dt*(kx(i)*kx(i) + ky(j)*ky(j))&
         *cmplx(0.0d0,1.0d0))*vnay(j,i)
    END DO
    CALL dfftw_execute_dft_(planby,vnby(1:Ny,i),vnay(1:Ny,i))
  END DO
  !$OMP END PARALLEL DO
  IF (mod(n,plotgap)==0) then
    time(1+n/plotgap)=n*dt
    PRINT *,'time',n*dt
  END IF
END DO
PRINT *,'Finished time stepping'
CALL system_clock(finish,count_rate)
PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),&
  'for Time stepping'

! transform back final data and do another half time step
vnbx(1:Nx,1:Ny)=transpose(vnay(1:Ny,1:Nx))/REAL(Ny,kind(0d0))
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,Ny
  CALL dfftw_execute_dft_(planbx,vnbx(1:Nx,j),unax(1:Nx,j))
  unax(1:Nx,j)=unax(1:Nx,j)/REAL(Nx,kind(0d0))
  potx(1:Nx,j)=Es*unax(1:Nx,j)*conjg(unax(1:Nx,j))
  unax(1:Nx,j)=exp(cmplx(0,-1)*dt*potx(1:Nx,j))*unax(1:Nx,j)
  CALL dfftw_execute_dft_(planfx,unax(1:Nx,j),vnax(1:Nx,j))
END DO
!$OMP END PARALLEL DO
```

```fortran
vnby(1:Ny,1:Nx)=TRANSPOSE(vnax(1:Nx,1:Ny))
!$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
DO i=1,Nx
  CALL dfftw_execute_dft_(planfy,vnby(1:Ny,i),vnay(1:Ny,i))
  vnby(1:Ny,i)=exp(0.5d0*dt*(kx(i)*kx(i) + ky(1:Ny)*ky(1:Ny))&
        *cmplx(0,1))*vnay(1:Ny,i)
  CALL dfftw_execute_dft_(planby,vnby(1:Ny,i),vnay(1:Ny,i))
END DO
!$OMP END PARALLEL DO
vnbx(1:Nx,1:Ny)=TRANSPOSE(vnay(1:Ny,1:Nx))/REAL(Ny,kind(0d0))
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,Ny
  CALL dfftw_execute_dft_(planbx,vnbx(1:Nx,j),unax(1:Nx,j))
  unax(1:Nx,j)=unax(1:Nx,j)/REAL(Nx,kind(0d0))
END DO
!$OMP END PARALLEL DO
name_config = 'ufinal.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,Ny
  DO i=1,Nx
    WRITE(11,*) abs(unax(i,j))**2
  END DO
END DO
CLOSE(11)

name_config = 'tdata.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,1+Nt/plotgap
  WRITE(11,*) time(j)
END DO
CLOSE(11)

name_config = 'xcoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO i=1,Nx
  WRITE(11,*) x(i)
END DO
CLOSE(11)

name_config = 'ycoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,Ny
  WRITE(11,*) y(j)
END DO
CLOSE(11)
PRINT *,'Saved data'

```

```
CALL dfftw_destroy_plan_(planbx)
CALL dfftw_destroy_plan_(planfx)
CALL dfftw_destroy_plan_(planby)
CALL dfftw_destroy_plan_(planfy)
CALL dfftw_cleanup_()

DEALLOCATE(unax,vnax,vnbx,potx, vnay,vnby,stat=allocatestatus)
IF (allocatestatus .ne. 0) STOP
PRINT *,'Deallocated memory'

  PRINT *,'Program execution complete'
END PROGRAM main
```

Listing 12.12: An example makefile for compiling the OpenMP program in listing 12.11. The example assumes one is using Flux and has loaded environments for the intel compiler as well as the Intel compiled version of FFTW. If one is using the freely available GCC compilers instead of the Intel compilers, change the flag $-openmp$ to $-fopenmp$.

```
#define the complier
COMPILER = gfortran
# compilation settings, optimization, precision, parallelization
FLAGS = -O0 -fopenmp


# libraries
LIBS = -L/usr/local/lib -lfftw3 -lm
# source list for main program
SOURCES =  NLSsplitting.f90

test: $(SOURCES)
    ${COMPILER} -o NLSsplitting $(FLAGS) $(SOURCES) $(LIBS)

clean:
  rm *.o

clobber:
  rm NLSsplitting
```

6) Modify the OpenMP Fortran codes to solve the Gross-Pitaevskii equation in two and three dimensions.

7) [5] Some quantum hydrodynamic models for plasmas are very similar to the nonlinear Schrödinger equation and can also be numerically approximated using splitting methods. A model for a plasma used by Eliasson and Shukla [16] is

$$i\Psi_t + \Delta\Psi + \phi\Psi - |\Psi|^{4/D}\Psi = 0 \tag{12.15}$$

[5]This question is due to a project by Joshua Kirschenheiter.

and

$$\Delta\phi = |\Psi|^2 - 1, \tag{12.16}$$

where Ψ is the effective wave function, ϕ the electrostatic potential and D the dimension, typically 1,2 or 3. This equation can be solved in a similar manner to the Davey-Stewartson equations in Klein, Muite and Roidot [32]. Specifically, first solve

$$iP_t + \Delta P = 0 \tag{12.17}$$

using the Fourier transform so that

$$P(\delta t) = \exp\left(-i\Delta\delta t\right) P(0).$$

where $P(0) = \Psi(0)$. Then solve

$$\phi = \Delta^{-1}\left(|P|^2 - 1\right) \tag{12.18}$$

using the Fourier transform. Finally, solve

$$iQ_t + \phi Q - |Q|^{4/D} Q = 0 \tag{12.19}$$

using the fact that at each grid point $\phi - |Q|^{4/D}$ is a constant, so the solution is

$$Q(\delta t) = \exp\left[i\left(\phi - |\Phi|^{4/D}\right)\delta t\right] Q(0)$$

with $Q(0) = P(\delta t)$ and $\Psi(\delta t) \approx Q(\delta t)$.

8) [6]The operator splitting method can be used for equations other than the nonlinear Schrödinger equation. Another equation for which operator splitting can be used is the complex Ginzburg-Landau equation

$$\frac{\partial A}{\partial t} = A + (1 + i\alpha)\Delta A - (1 + i\beta)|A|^2 A,$$

where A is a complex function, typically of one, two or three variables. An example one dimensional code is provided in listing 12.13, based on an earlier finite difference code by Blanes, Casa, Chartier and Miura, using the methods described in Blanes et al. [3]. By using complex coefficients, Blanes et al. [3] can create high order splitting methods for parabolic equations. Previous attempts to do this have failed since if only real coefficients are used, a backward step which is required for methods higher than second order leads to numerical instability. Modify the example code to solve the complex Ginzburg-Landau equation in one, two and then in three spatial dimensions. The linear part

$$\frac{\partial A}{\partial t} = A + (1 + i\alpha)\Delta A$$

[6]This question is due to a project by Kohei Harada and Matt Warnez.

can be solved explicitly using the Fourier transform. To solve the nonlinear part,

$$\frac{\partial A}{\partial t} = -(1+i\beta)|A|^2 A$$

consider

$$\frac{\partial |A|^2}{\partial t} = \frac{\partial A}{\partial t}A^* + \frac{\partial A^*}{\partial t}A = 2|A|^4$$

and solve this exactly for $|A|^2$. To recover the phase, observe that

$$\frac{\partial \log(A)}{\partial t} = -(1+i\beta)|A|^2$$

which can also be integrated explicitly since $|A|^2(t)$ is known.

Listing 12.13: A Matlab program which uses 16th order splitting to solve the cubic nonlinear Schrödinger equation.

```
% A program to solve the nonlinear Schr\"{o}dinger equation using a
% splitting method. The numerical solution is compared to an exact
% solution.
% S. Blanes, F. Casas, P. Chartier and A. Murua
% "Optimized high-order splitting methods for some classes of
  parabolic
% equations"
% ArXiv pre-print 1102.1622v2
% Forthcoming Mathematics of Computation

clear all; format compact; format short;
set(0,'defaultaxesfontsize',30,'defaultaxeslinewidth',.7,...
    'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,...
    'defaultaxesfontweight','bold')

% set up grid
Lx = 20;         % period  2*pi * L
Nx = 16384;      % number of harmonics
Nt = 2000;       % number of time slices
dt = 0.25*pi/Nt;% time step
U=zeros(Nx,Nt/10);
method=3; % splitting method: 1 Strang, 2 CCDV10, 3 Blanes et al 2012

% initialise variables
x = (2*pi/Nx)*(-Nx/2:Nx/2 -1)'*Lx;          % x coordinate
kx = 1i*[0:Nx/2-1 0 -Nx/2+1:-1]'/Lx;        % wave vector

% initial conditions
t=0; tdata(1)=t;
u=4*exp(1i*t)*(cosh(3*x)+3*exp(8*1i*t)*cosh(x))...
    ./(cosh(4*x)+4*cosh(2*x)+3*cos(8*t));
v=fft(u);
```

```
figure(1); clf; plot(x,u);xlim([-2,2]); drawnow;
U(:,1)=u;

% mass
ma = fft(abs(u).^2);
ma0 = ma(1);

if method==1,
    %
    % Strang-Splitting
    %
    s=2;
    a=[1;0];
    b=[1/2;1/2];
    %
elseif method==2,
    %
    % Method of Castella, Chartier, Descombes and Vilmart
    % BIT Numerical Analysis vol 49 pp 487-508, 2009
    %
    s=5;
    a=[1/4;1/4;1/4;1/4;0];
    b=[1/10-1i/30;4/15+2*1i/15;4/15-1i/5;4/15+2*1i/15;1/10-1i/30];
    %
elseif method==3,
    %
    % Method of Blanes, Casas, Chartier and Murua 2012
    %
    s=17;
    a=1/16*[1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;0];
    b=[0.028920177910074098791 - 0.005936580835725746103*1i;
        0.056654351383649876160 + 0.020841963949772627119*1i;
        0.067258385822722143569 - 0.039386393748812362460*1i;
        0.070333980553260772061 + 0.058952097930307840316*1i;
        0.077095100838099173580 - 0.038247636602014810025*1i;
        0.042022140317231098258 - 0.033116379859951038579*1i;
        0.050147397749937784280 + 0.061283684958324249562*1i;
        0.047750191909146143447 - 0.032332468814362628262*1i;
        0.119636547031757819706 + 0.015883426044923736862*1i;
        0.047750191909146143447 - 0.032332468814362628262*1i;
        0.050147397749937784280 + 0.061283684958324249562*1i;
        0.042022140317231098258 - 0.033116379859951038579*1i;
        0.077095100838099173580 - 0.038247636602014810025*1i;
        0.070333980553260772061 + 0.058952097930307840316*1i;
        0.067258385822722143569 - 0.039386393748812362460*1i;
        0.056654351383649876160 + 0.020841963949772627119*1i;
        0.028920177910074098791 - 0.005936580835725746103*1i];
end;


% solve pde and plot results
```

```
for n =2:Nt+1
    for m=1:(s-1)
        vna=exp(b(m)*1i*dt*kx.*kx).*v;
        una=ifft(vna);
        pot=(2*una.*conj(una));
        unb=exp(-1i*a(m)*(-1)*dt*pot).*una;
        v=fft(unb);
    end
    v=exp(b(s)*1i*dt*kx.*kx).*v;
    u=ifft(v);
    t=(n-1)*dt;
    if (mod(n,10)==0)
        tdata(n/10)=t;
        u=ifft(v);
        U(:,n/10)=u;
        uexact=...
            4*exp(1i*t)*(cosh(3*x)+3*exp(8*1i*t)*cosh(x))...
            ./(cosh(4*x)+4*cosh(2*x)+3*cos(8*t));
        figure(1); clf; plot(x,abs(u).^2); ...
            xlim([-0.5,0.5]); title(num2str(t));
        figure(2); clf; loglog(abs(v(1:Nx/2))); ...
            title('Fourier Coefficients');
        figure(3); clf; plot(x,abs(u-uexact).^2); ...
            xlim([-0.5,0.5]); title('error');
        drawnow;
        ma = fft(abs(u).^2);
        ma = ma(1);
        test = log10(abs(1-ma/ma0))
    end
end
figure(4); clf; mesh(tdata(1:(n-1)/10),x,abs(U(:,1:(n-1)/10)).^2);
xlim([0,t]);
```

12.8 Distributed Memory Parallel: MPI

For this section, we will use the library 2DECOMP&FFT available from `http://www.2decomp.org/index.html`. The website includes some examples which indicate how this library should be used, in particular the sample code at `http://www.2decomp.org/case_study1.html` is a very helpful indication of how one converts a code that uses FFTW to one that uses MPI and the aforementioned library.

Before creating a parallel MPI code using 2DECOMP&FFT, we will generate a serial Fortran code that uses splitting to solve the 3D nonlinear Schrödinger equation. Rather than using loop-based parallelization to do a sequence of one dimensional fast Fourier transforms, we will use FFTW's three dimensional FFT, so that the serial version and MPI parallel version have the same structure. The serial version is in listing 12.14. This file can be compiled in a similar manner to that in 10.1.

Listing 12.14: A Fortran program to solve the 3D nonlinear Schrödinger equation using splitting and FFTW.

```
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This program solves nonlinear Schrodinger equation in 3 dimensions
! i*u_t+Es*|u|^2u+u_{xx}+u_{yy}+u_{zz}=0
! using a second order time spectral splitting scheme
!
! The boundary conditions are u(x=0,y,z)=u(2*Lx*\pi,y,z),
! u(x,y=0,z)=u(x,y=2*Ly*\pi,z), u(x,y,z=0)=u(x,y,z=2*Lz*\pi)
! The initial condition is u=exp(-x^2-y^2)
!
! .. Parameters ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  Nz        = number of modes in z - power of 2 for FFT
!  Nt        = number of timesteps to take
!  Tmax        = maximum simulation time
!  plotgap       = number of timesteps between plots
!  FFTW_IN_PLACE  = value for FFTW input
!  FFTW_MEASURE   = value for FFTW input
!  FFTW_EXHAUSTIVE  = value for FFTW input
!  FFTW_PATIENT   = value for FFTW input
!  FFTW_ESTIMATE  = value for FFTW input
!  FFTW_FORWARD     = value for FFTW input
!  FFTW_BACKWARD  = value for FFTW input
!  pi = 3.14159265358979323846264338327950288419716939937510d0
!  Lx        = width of box in x direction
!  Ly        = width of box in y direction
!  Lz        = width of box in z direction
!  ES        = +1 for focusing and -1 for defocusing
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!  k         = loop counter in z direction
!  n         = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  start      = variable to record start time of program
!  finish     = variable to record end time of program
!  count_rate   = variable for clock count rate
!  count      = keep track of information written to disk
!  iol        = size of array to write to disk
!  planfxyz      = Forward 3d fft plan
!  planbxyz      = Backward 3d fft plan
!  dt        = timestep
!  modescalereal  = Number to scale after backward FFT
!  ierr       = error code
! .. Arrays ..
!  unax        = approximate solution
!  vnax        = Fourier transform of approximate solution
!  potx        = potential
! .. Vectors ..
!  kx       = fourier frequencies in x direction
!  ky       = fourier frequencies in y direction
!  x        = x locations
!  y        = y locations
!  time        = times at which save data
!  name_config     = array to store filename for data to be saved
!  fftfxy      = array to setup 2D Fourier transform
!  fftbxy      = array to setup 2D Fourier transform
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
! FFTW3  -- Fast Fourier Transform in the West Library
!     (http://www.fftw.org/)
PROGRAM main
! Declare variables
IMPLICIT NONE
INTEGER(kind=4), PARAMETER  ::  Nx=2**5
INTEGER(kind=4), PARAMETER  ::  Ny=2**5
INTEGER(kind=4), PARAMETER  ::  Nz=2**5
INTEGER(kind=4), PARAMETER  ::  Nt=50
INTEGER(kind=4), PARAMETER  ::  plotgap=10
REAL(kind=8), PARAMETER ::&
  pi=3.14159265358979323846264338327950288419716939937510d0
REAL(kind=8), PARAMETER ::  Lx=2.0d0,Ly=2.0d0,Lz=2.0d0
REAL(kind=8), PARAMETER ::  Es=1.0d0
REAL(kind=8)  ::  dt=0.10d0/Nt
REAL(kind=8) :: modescalereal
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  kx,ky,kz
REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  x,y,z
COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE  ::  unax,vnax,potx
REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  time
INTEGER(kind=4) ::  i,j,k,n,AllocateStatus,count,iol
  ! timing
INTEGER(kind=4) ::  start, finish, count_rate
```

```fortran
    ! fftw variables
INTEGER(kind=8), PARAMETER  ::  FFTW_IN_PLACE = 8, FFTW_MEASURE = 0, &
  FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32, FFTW_ESTIMATE = 64
INTEGER(kind=8),PARAMETER ::  FFTW_FORWARD = -1, FFTW_BACKWARD=1
INTEGER(kind=8) ::  planfxyz,planbxyz
CHARACTER*100 ::  name_config, number_file

CALL system_clock(start,count_rate)
ALLOCATE(unax(1:Nx,1:Ny,1:Nz),vnax(1:Nx,1:Ny,1:Nz),potx(1:Nx,1:Ny,1:Nz)
    ,&
      kx(1:Nx),ky(1:Ny),kz(1:Nz),x(1:Nx),y(1:Ny),z(1:Nz),&
      time(1:1+Nt/plotgap),stat=AllocateStatus)
IF (AllocateStatus .ne. 0) STOP
PRINT *,'allocated space'
modescalereal=1.0d0/REAL(Nx,KIND(0d0))
modescalereal=modescalereal/REAL(Ny,KIND(0d0))
modescalereal=modescalereal/REAL(Nz,KIND(0d0))

! set up ffts
CALL dfftw_plan_dft_3d_(planfxyz,Nx,Ny,Nz,unax(1:Nx,1:Ny,1:Nz),&
  vnax(1:Nx,1:Ny,1:Nz),FFTW_FORWARD,FFTW_ESTIMATE)
CALL dfftw_plan_dft_3d_(planbxyz,Nx,Ny,Nz,vnax(1:Nx,1:Ny,1:Nz),&
  unax(1:Nx,1:Ny,1:Nz),FFTW_BACKWARD,FFTW_ESTIMATE)

PRINT *,'Setup FFTs'

! setup fourier frequencies and grid points
DO i=1,1+Nx/2
  kx(i)= cmplx(0.0d0,1.0)*REAL(i-1,kind(0d0))/Lx
END DO
kx(1+Nx/2)=0.0d0
DO i = 1,Nx/2 -1
  kx(i+1+Nx/2)=-kx(1-i+Nx/2)
END DO
  DO i=1,Nx
  x(i)=(-1.0d0+2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)) )*pi*Lx
END DO
DO j=1,1+Ny/2
  ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
END DO
ky(1+Ny/2)=0.0d0
DO j = 1,Ny/2 -1
  ky(j+1+Ny/2)=-ky(1-j+Ny/2)
END DO
  DO j=1,Ny
  y(j)=(-1.0d0+2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)) )*pi*Ly
END DO
DO k=1,1+Nz/2
  kz(k)= cmplx(0.0d0,1.0d0)*REAL(k-1,kind(0d0))/Lz
END DO
kz(1+Nz/2)=0.0d0
```

```
DO k = 1,Nz/2 -1
  kz(k+1+Nz/2)=-kz(1-k+Nz/2)
END DO
  DO k=1,Nz
  z(k)=(-1.0d0+2.0d0*REAL(k-1,kind(0d0))/REAL(Nz,kind(0d0)) )*pi*Lz
END DO

PRINT *,'Setup grid and fourier frequencies'

DO k=1,Nz;  DO j=1,Ny; DO i=1,Nx
  unax(i,j,k)=exp(-1.0d0*(x(i)**2 +y(j)**2+z(k)**2))
END DO; END DO; END DO

name_config = 'uinitial.dat'
INQUIRE(iolength=iol) unax(1,1,1)
OPEN(unit=11,FILE=name_config,form="unformatted", &
     access="direct",recl=iol)
count=1
DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
  WRITE(11,rec=count) unax(i,j,k)
       count=count+1
END DO; END DO; END DO
CLOSE(11)

CALL dfftw_execute_dft_(planfxyz,unax(1:Nx,1:Ny,1:Nz),vnax(1:Nx,1:Ny,1:
   Nz))

PRINT *,'Got initial data, starting timestepping'
time(1)=0
DO n=1,Nt
  DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
      vnax(i,j,k)=exp(0.50d0*dt*&
        (kz(k)*kz(k) + kx(i)*kx(i) + ky(j)*ky(j))&
        *cmplx(0.0d0,1.0d0))*vnax(i,j,k)
  END DO; END DO; END DO
  CALL dfftw_execute_dft_(planbxyz,vnax(1:Nx,1:Ny,1:Nz),&
    unax(1:Nx,1:Ny,1:Nz))

  DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
    unax(i,j,k)=unax(i,j,k)*modescalereal
    potx(i,j,k)=Es*unax(i,j,k)*conjg(unax(i,j,k))
    unax(i,j,k)=exp(cmplx(0.0d0,-1.0d0)*dt*potx(i,j,k))&
        *unax(i,j,k)
  END DO; END DO; END DO
  CALL dfftw_execute_dft_(planfxyz,unax(1:Nx,1:Ny,1:Nz),&
    vnax(1:Nx,1:Ny,1:Nz))

  DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
    vnax(i,j,k)=exp(0.5d0*dt*&
        (kx(i)*kx(i) + ky(j)*ky(j)+ kz(k)*kz(k))&
          *cmplx(0.0d0,1.0d0))*vnax(i,j,k)
```

```fortran
  END DO; END DO; END DO
  IF (mod(n,plotgap)==0) THEN
    time(1+n/plotgap)=n*dt
    PRINT *,'time',n*dt
    CALL dfftw_execute_dft_(planbxyz,vnax(1:Nx,1:Ny,1:Nz),unax(1:Nx,1:Ny,1:Nz))
    DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
      unax(i,j,k)=unax(i,j,k)*modescalereal
    END DO; END DO; END DO
    name_config='./data/u'
    WRITE(number_file,'(i0)') 10000000+1+n/plotgap
    ind=index(name_config,' ') -1
    name_config=name_config(1:ind)//numberfile
    ind=index(name_config,' ') -1
    name_config=name_config(1:ind)//'.dat'
    OPEN(unit=11,FILE=name_config,status="UNKNOWN")
    REWIND(11)
    DO j=1,Ny
      DO i=1,Nx
        WRITE(11,*) abs(unax(i,j))**2
      END DO
    END DO
    CLOSE(11)

  END IF
END DO
PRINT *,'Finished time stepping'

  ! transform back final data and do another half time step
CALL system_clock(finish,count_rate)
PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),'for execution'

name_config = 'tdata.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,1+Nt/plotgap
  WRITE(11,*) time(j)
END DO
CLOSE(11)

name_config = 'xcoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO i=1,Nx
  WRITE(11,*) x(i)
END DO
CLOSE(11)

name_config = 'ycoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
```

```
  REWIND(11)
  DO j=1,Ny
    WRITE(11,*) y(j)
  END DO
  CLOSE(11)

  name_config = 'zcoord.dat'
  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
  REWIND(11)
  DO k=1,Nz
    WRITE(11,*) z(k)
  END DO
  CLOSE(11)
  PRINT *,'Saved data'

  CALL dfftw_destroy_plan_(planbxyz)
  CALL dfftw_destroy_plan_(planfxyz)
  CALL dfftw_cleanup_()

  DEALLOCATE(unax,vnax,potx,&
        kx,ky,kz,x,y,z,&
        time,stat=AllocateStatus)
  IF (AllocateStatus .ne. 0) STOP

  PRINT *,'Program execution complete'
  END PROGRAM main
```

In comparison to the previous programs, the program in listing 12.14 writes out its final data as a binary file. This is often significantly faster than writing out a text file, and the resulting file is usually much smaller in size. This is important when many such files are written and/or if individual files are large. Due to the formatting change, the binary file also needs to be read in slightly differently. The Matlab script in listing 12.15 shows how to do this.

Listing 12.15: A Matlab program which plots a numerical solution to a 3D nonlinear Schrödinger equation generated by listings 12.14 or 12.16.

```
% A program to plot the computed results

clear all; format compact, format short,
set(0,'defaultaxesfontsize',18,'defaultaxeslinewidth',.9,...
    'defaultlinelinewidth',3.5,'defaultpatchlinewidth',5.5);

% Load data
tdata=load('./tdata.dat');
x=load('./xcoord.dat');
y=load('./ycoord.dat');
z=load('./zcoord.dat');
Tsteps = length(tdata);

```

```
Nx = length(x); Nt = length(tdata);
Ny = length(y); Nz = length(z);
fid=fopen('./ufinal.datbin','r');
[fname,mode,mformat]=fopen(fid);
u=fread(fid,Nx*Ny*Nz,'double',mformat);
u = reshape(u,Nx,Ny,Nz);

% Plot data
figure (1); clf ; UP = abs(u).^2;
p1 = patch(isosurface(x,y,z,UP,.0025) ,...
  'FaceColor','yellow','EdgeColor','none');
p2 = patch(isocaps(x,y,z,UP,.0025) ,...
  'FaceColor','interp','EdgeColor','none');
isonormals(UP,p1); lighting phong;
xlabel('x'); ylabel('y'); zlabel('z');
axis equal; axis square; view(3); drawnow;
```

We now modify the above code to use MPI and the library 2DECOMP&FFT. The library 2DECOMP&FFT hides most of the details of MPI although there are a few commands which it is useful for the user to understand. These commands are:

- `USE mpi` or `INCLUDE 'mpif.h'`
- `MPI_INIT`
- `MPI_COMM_SIZE`
- `MPI_COMM_RANK`
- `MPI_FINALIZE`

The program is listed in listing 12.16, please compare this to the serial code in 12.14. The library 2DECOMP&FFT does a domain decomposition of the arrays so that separate parts of the arrays are on separate processors. The library can also perform a Fourier transform on the arrays even though they are stored on different processors – the library does all the necessary message passing and transpositions required to perform the Fourier transform. It should be noted that the order of the entries in the arrays after the Fourier transform is not necessarily the same as the order used by FFTW. However, the correct ordering of the entries is returned by the structure `decomp` and so this structure is used to obtain starting and stopping entries for the loops. We assume that the library 2DECOMP&FFT has been installed in an appropriate location.

Listing 12.16: A Fortran program to solve the 3D nonlinear Schrödinger equation using splitting and 2DECOMP&FFT.

```



```

```


!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This program solves nonlinear Schrodinger equation in 3 dimensions
! i*u_t+Es*|u|^2u+u_{xx}+u_{yy}+u_{zz}=0
! using a second order time spectral splitting scheme
!
! The boundary conditions are u(x=0,y,z)=u(2*Lx*\pi,y,z),
! u(x,y=0,z)=u(x,y=2*Ly*\pi,z), u(x,y,z=0)=u(x,y,z=2*Lz*\pi)
! The initial condition is u=exp(-x^2-y^2)
!
! .. Parameters ..
!  Nx       = number of modes in x - power of 2 for FFT
!  Ny       = number of modes in y - power of 2 for FFT
!  Nz       = number of modes in z - power of 2 for FFT
!  Nt       = number of timesteps to take
!  Tmax       = maximum simulation time
!  plotgap      = number of timesteps between plots
!  pi = 3.14159265358979323846264338327950288419716939937510d0
!  Lx       = width of box in x direction
!  Ly       = width of box in y direction
!  Lz       = width of box in z direction
!  ES       = +1 for focusing and -1 for defocusing
! .. Scalars ..
!  i        = loop counter in x direction
!  j        = loop counter in y direction
!  k        = loop counter in z direction
!  n        = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  start      = variable to record start time of program
!  finish     = variable to record end time of program
!  count_rate   = variable for clock count rate
!  dt       = timestep
!  modescalereal  = Number to scale after backward FFT
!  myid       = Process id
!  ierr       = error code
!  p_row      = number of rows for domain decomposition
!  p_col      = number of columns for domain decomposition
!  filesize     = total filesize
!  disp       = displacement to start writing data from
!  ind        = index in array to write
!  plotnum      = number of plot to save
!  numberfile   = number of the file to be saved to disk
!  stat       = error indicator when reading inputfile
! .. Arrays ..
!  u        = approximate solution
!  v        = Fourier transform of approximate solution
```

```fortran
!  pot         = potential
! .. Vectors ..
!  kx        = fourier frequencies in x direction
!  ky        = fourier frequencies in y direction
!  kz        = fourier frequencies in z direction
!  x         = x locations
!  y         = y locations
!  z         = z locations
!  time        = times at which save data
!  nameconfig   = array to store filename for data to be saved
!  InputFileName  = name of the Input File
! .. Special Structures ..
!  decomp      = contains information on domain decomposition
!         see http://www.2decomp.org/ for more information
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
! 2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
!     (http://www.2decomp.org/index.html)
! MPI library

PROGRAM main
USE decomp_2d
USE decomp_2d_fft
USE decomp_2d_io
USE MPI
! Declare variables
IMPLICIT NONE
INTEGER(kind=4)   ::  Nx=2**5
INTEGER(kind=4)   ::  Ny=2**5
INTEGER(kind=4)   ::  Nz=2**5
INTEGER(kind=4)   ::  Nt=50
INTEGER(kind=4)   ::  plotgap=10
REAL(kind=8), PARAMETER ::&
  pi=3.14159265358979323846264338327950288419716939937510d0
REAL(kind=8)    ::  Lx=2.0d0,Ly=2.0d0,Lz=2.0d0
REAL(kind=8)    ::  Es=1.0d0
REAL(kind=8)    ::  dt=0.0010d0
```

```
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  kx,ky,kz
REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  x,y,z
COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE  ::  u,v,pot
REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  time
INTEGER(KIND=4), DIMENSION(1:5) ::  intcomm
REAL(KIND=8), DIMENSION(1:5)  ::  dpcomm
REAL(kind=8) :: modescalereal
INTEGER(kind=4) ::  i,j,k,n,AllocateStatus,stat
INTEGER(kind=4) :: myid,numprocs,ierr
TYPE(DECOMP_INFO) ::  decomp
INTEGER(kind=MPI_OFFSET_KIND) :: filesize, disp
INTEGER(kind=4) ::  p_row=0, p_col=0
INTEGER(kind=4) ::  start, finish, count_rate, ind, plotnum
CHARACTER*50  ::  nameconfig
CHARACTER*20  ::  numberfile, InputFileName
    ! initialisation of MPI
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

IF(myid.eq.0) THEN
  CALL GET_ENVIRONMENT_VARIABLE(NAME='inputfile',VALUE=InputFileName,
     STATUS=stat)
END IF
CALL MPI_BCAST(stat,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

IF(stat.NE.0) THEN
  IF(myid.eq.0) THEN
      PRINT*,"Need to set environment variable inputfile to the name of
          the &
           file where the simulation parameters are set"
  END IF
  STOP
END IF
IF(myid.eq.0) THEN
  InputFileName='./INPUTFILE'
  OPEN(unit=11,FILE=trim(InputFileName),status="OLD")
  REWIND(11)
  READ(11,*) intcomm(1), intcomm(2), intcomm(3), intcomm(4), intcomm(5),
      &
        dpcomm(1), dpcomm(2),  dpcomm(3),  dpcomm(4),  dpcomm(5)
  CLOSE(11)
  PRINT *,"NX ",intcomm(1)
  PRINT *,"NY ",intcomm(2)
  PRINT *,"NZ ",intcomm(3)
  PRINT *,"NT ",intcomm(4)
  PRINT *,"plotgap ",intcomm(5)
  PRINT *,"Lx ",dpcomm(1)
  PRINT *,"Ly ",dpcomm(2)
  PRINT *,"Lz ",dpcomm(3)
  PRINT *,"Es ",dpcomm(4)
```

```
    PRINT *,"Dt ",dpcomm(5)
    PRINT *,"Read inputfile"
  END IF
  CALL MPI_BCAST(dpcomm,5,MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD,ierr)
  CALL MPI_BCAST(intcomm,5,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

  Nx=intcomm(1)
  Ny=intcomm(2)
  Nz=intcomm(3)
  Nt=intcomm(4)
  plotgap=intcomm(5)
  Lx=dpcomm(1)
  Ly=dpcomm(2)
  Lz=dpcomm(3)
  Es=dpcomm(4)
  DT=dpcomm(5)

  ! initialisation of 2decomp
  ! do automatic domain decomposition
  CALL decomp_2d_init(Nx,Ny,Nz,p_row,p_col)
  ! get information about domain decomposition choosen
  CALL decomp_info_init(Nx,Ny,Nz,decomp)
  ! initialise FFT library
  CALL decomp_2d_fft_init
  ALLOCATE(u(decomp%xst(1):decomp%xen(1),&
          decomp%xst(2):decomp%xen(2),&
          decomp%xst(3):decomp%xen(3)),&
            v(decomp%zst(1):decomp%zen(1),&
              decomp%zst(2):decomp%zen(2),&
              decomp%zst(3):decomp%zen(3)),&
            pot(decomp%xst(1):decomp%xen(1),&
              decomp%xst(2):decomp%xen(2),&
              decomp%xst(3):decomp%xen(3)),&
        kx(1:Nx),ky(1:Ny),kz(1:Nz),&
        x(1:Nx),y(1:Ny),z(1:Nz),&
        time(1:1+Nt/plotgap),stat=AllocateStatus)
  IF (AllocateStatus .ne. 0) STOP

  IF (myid.eq.0) THEN
    PRINT *,'allocated space'
  END IF

  modescalereal=1.0d0/REAL(Nx,KIND(0d0))
  modescalereal=modescalereal/REAL(Ny,KIND(0d0))
  modescalereal=modescalereal/REAL(Nz,KIND(0d0))

  ! setup fourier frequencies and grid points
  DO i=1,1+Nx/2
    kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
  END DO
  kx(1+Nx/2)=0.0d0
```

```
DO i = 1,Nx/2 -1
  kx(i+1+Nx/2)=-kx(1-i+Nx/2)
END DO
  DO i=1,Nx
  x(i)=(-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)))*pi*Lx
END DO
DO j=1,1+Ny/2
  ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
END DO
ky(1+Ny/2)=0.0d0
DO j = 1,Ny/2 -1
  ky(j+1+Ny/2)=-ky(1-j+Ny/2)
END DO
  DO j=1,Ny
  y(j)=(-1.0d0 + 2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)))*pi*Ly
END DO
DO k=1,1+Nz/2
  kz(k)= cmplx(0.0d0,1.0d0)*REAL(k-1,kind(0d0))/Lz
END DO
kz(1+Nz/2)=0.0d0
DO k = 1,Nz/2 -1
  kz(k+1+Nz/2)=-kz(1-k+Nz/2)
END DO
  DO k=1,Nz
  z(k)=(-1.0d0 + 2.0d0*REAL(k-1,kind(0d0))/REAL(Nz,kind(0d0)))*pi*Lz
END DO

IF (myid.eq.0) THEN
  PRINT *,'Setup grid and fourier frequencies'
END IF

DO k=decomp%xst(3),decomp%xen(3)
  DO j=decomp%xst(2),decomp%xen(2)
    DO i=decomp%xst(1),decomp%xen(1)
      u(i,j,k)=exp(-1.0d0*(x(i)**2 +y(j)**2+z(k)**2))
    END DO
  END DO
END DO

! write out using 2DECOMP&FFT MPI-IO routines
nameconfig='./data/u'
plotnum=0
WRITE(numberfile,'(i0)') 10000000+plotnum
ind=index(nameconfig,' ') -1
nameconfig=nameconfig(1:ind)//numberfile
ind=index(nameconfig,' ') -1
nameconfig=nameconfig(1:ind)//'.datbin'
CALL decomp_2d_write_one(1,u,nameconfig)

CALL decomp_2d_fft_3d(u,v,DECOMP_2D_FFT_FORWARD)
IF (myid.eq.0) THEN
```

```
    PRINT *,'Got initial data, starting timestepping'
  END IF
  CALL system_clock(start,count_rate)
  time(1)=0
  DO n=1,Nt
    ! Use Strang splitting
    DO k=decomp%zst(3),decomp%zen(3)
      DO j=decomp%zst(2),decomp%zen(2)
        DO i=decomp%zst(1),decomp%zen(1)
          v(i,j,k)=exp(0.50d0*dt*&
            (kz(k)*kz(k) + kx(i)*kx(i) + ky(j)*ky(j))&
            *cmplx(0.0d0,1.0d0))*v(i,j,k)
        END DO
      END DO
    END DO

    CALL decomp_2d_fft_3d(v,u,DECOMP_2D_FFT_BACKWARD)

    DO k=decomp%xst(3),decomp%xen(3)
      DO j=decomp%xst(2),decomp%xen(2)
        DO i=decomp%xst(1),decomp%xen(1)
          u(i,j,k)=u(i,j,k)*modescalereal
          pot(i,j,k)=Es*u(i,j,k)*conjg(u(i,j,k))
          u(i,j,k)=exp(cmplx(0.0d0,-1.0d0)*dt*pot(i,j,k))*u(i,j,k)
        END DO
      END DO
    END DO
    CALL decomp_2d_fft_3d(u,v,DECOMP_2D_FFT_FORWARD)

    DO k=decomp%zst(3),decomp%zen(3)
      DO j=decomp%zst(2),decomp%zen(2)
        DO i=decomp%zst(1),decomp%zen(1)
          v(i,j,k)=exp(dt*0.5d0*&
            (kx(i)*kx(i) +ky(j)*ky(j) +kz(k)*kz(k))&
            *cmplx(0.0d0,1.0d0))*v(i,j,k)
        END DO
      END DO
    END DO
    IF (mod(n,plotgap)==0) THEN
      time(1+n/plotgap)=n*dt
      IF (myid.eq.0) THEN
        PRINT *,'time',n*dt
      END IF
      CALL decomp_2d_fft_3d(v,u,DECOMP_2D_FFT_BACKWARD)
      u=u*modescalereal
      nameconfig='./data/u'
      plotnum=plotnum+1
      WRITE(numberfile,'(i0)') 10000000+plotnum
      ind=index(nameconfig,' ') -1
      nameconfig=nameconfig(1:ind)//numberfile
      ind=index(nameconfig,' ') -1
```

```fortran
      nameconfig=nameconfig(1:ind)//'.datbin'
      ! write out using 2DECOMP&FFT MPI-IO routines
      CALL decomp_2d_write_one(1,u,nameconfig)
    END IF
  END DO
  IF (myid.eq.0) THEN
    PRINT *,'Finished time stepping'
  END IF

  CALL system_clock(finish,count_rate)

  IF (myid.eq.0) THEN
    PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),'for
       execution'
  END IF

  IF (myid.eq.0) THEN
    ! Save times at which output was made in text format
    nameconfig = './data/tdata.dat'
    OPEN(unit=11,FILE=nameconfig,status="UNKNOWN")
    REWIND(11)
    DO j=1,1+Nt/plotgap
      WRITE(11,*) time(j)
    END DO
    CLOSE(11)
    ! Save x grid points in text format
    nameconfig = './data/xcoord.dat'
    OPEN(unit=11,FILE=nameconfig,status="UNKNOWN")
    REWIND(11)
    DO i=1,Nx
      WRITE(11,*) x(i)
    END DO
    CLOSE(11)
    ! Save y grid points in text format
    nameconfig = './data/ycoord.dat'
    OPEN(unit=11,FILE=nameconfig,status="UNKNOWN")
    REWIND(11)
    DO j=1,Ny
      WRITE(11,*) y(j)
    END DO
    CLOSE(11)
    ! Save z grid points in text format
    nameconfig = './data/zcoord.dat'
    OPEN(unit=11,FILE=nameconfig,status="UNKNOWN")
    REWIND(11)
    DO k=1,Nz
      WRITE(11,*) z(k)
    END DO
    CLOSE(11)
    PRINT *,'Saved data'
  END IF
```

```

! clean up
  CALL decomp_2d_fft_finalize
  CALL decomp_2d_finalize
DEALLOCATE(u,v,pot,&
      kx,ky,kz,x,y,z,&
      time,stat=AllocateStatus)
IF (AllocateStatus .ne. 0) STOP
IF (myid.eq.0) THEN
    PRINT *,'Program execution complete'
END IF
CALL MPI_FINALIZE(ierr)
END PROGRAM main
```

12.9 Exercises

1) The ASCII character set requires 7 bits per character and so at least 7 bits are required to store a digit between 0 and 9. A double precision number in IEEE arithmetic requires 64 bits to store a double precision number with approximately 15 decimal digits and approximately a 3 decimal digit exponent. How many bits are required to store a IEEE double precision number? Suppose a file has 10^6 double precision numbers. What is the minimum size of the file if the numbers are stored as IEEE double precision numbers? What is the minimum size of the file if the numbers are stored as characters?

2) Write an MPI code using 2DECOMP&FFT to solve the Gross-Pitaevskii equation in three dimensions.

3) Learn to use either VisIt (`https://wci.llnl.gov/codes/visit/`) or Paraview (`http://www.paraview.org/`) and write a script to visualize two and three dimensional output in a manner that is similar to the Matlab codes.

Chapter 13

The Two- and Three-Dimensional Navier-Stokes Equations

13.1 Background

The Navier-Stokes equations describe the motion of a fluid. In order to derive the Navier-Stokes equations we assume that a fluid is a continuum (not made of individual particles, but rather a continuous substance) and that mass and momentum are conserved. After making some assumptions and using Newton's second law on an incompressible fluid particle, the Navier-Stokes equations can be derived in their entirety. All details are omitted since there are many sources of this information, two sources that are particularly clear are Tritton [58] and Doering and Gibbon [15]; Gallavotti [19] should also be noted for introducing both mathematical and physical aspects of these equations, and Uecker [59] includes a quick derivation and some example Fourier Spectral Matlab codes. For a more detailed introduction to spectral methods for the Navier-Stokes equations see Canuto et al. [9]. The incompressible Navier-Stokes equations are

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t}+\mathbf{u}\cdot\nabla\mathbf{u}\right) = -\nabla p+\mu\Delta\mathbf{u}+\mathbf{f} \tag{13.1}$$

$$\nabla\cdot\mathbf{u} = 0. \tag{13.2}$$

In these equations, ρ is density, $\mathbf{u}(x,y,z)=(u,v,w)$ is the velocity with components in the x, y and z directions, p is pressure field, μ is dynamic viscosity (constant in incompressible case) and $\mathbf{f}$ is a body force (force that acts through out the volume). Equation (13.1) represents conservation of momentum and eq. (13.2) is the continuity equation which represents conservation of mass for an incompressible fluid.

13.2 The Two-Dimensional Case

We will first consider the two-dimensional case. A difficulty in simulating the incompressible Navier-Stokes equations is the numerical satisfaction of the incompressibility constraint in eq.

(13.2), this is sometimes referred to as a divergence free condition or a solenoidal constraint. To automatically satisfy this incompressibility constraint in two dimensions, where

$$\mathbf{u}(x, y) = (u(x, y), v(x, y))$$

it is possible to re-write the equations using a different formulation, the stream-function vorticity formulation. In this case, we let

$$u = \frac{\partial \psi}{\partial y} \quad v = -\frac{\partial \psi}{\partial x},$$

where $\psi(x, y)$ is the streamfunction. Level curves of the streamfunction represent streamlines[1] of the fluid field. Note that

$$\nabla \cdot \mathbf{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = \frac{\partial^2 \psi}{\partial x \partial y} - \frac{\partial^2 \psi}{\partial y \partial x} = 0,$$

so eq. (13.2) is automatically satisfied. Making this change of variables, we obtain a single scalar partial differential equation by taking the curl of the momentum equation, eq. (13.1). We define the vorticity ω, so that

$$\omega = \nabla \times \mathbf{u} = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = -\Delta \psi$$

and eq. (13.1) becomes

$$\begin{aligned} &\frac{\partial}{\partial x}\left[\rho\left(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right)\right] - \frac{\partial}{\partial y}\left[\rho\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right)\right] \\ =& \frac{\partial}{\partial x}\left[\mu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) + fy\right] - \frac{\partial}{\partial y}\left[\mu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + fx\right] \end{aligned}$$

where fx and fy represent the x and y components of the force $\mathbf{f}$. Since the flow is divergence free,

$$\frac{\partial u}{\partial x} = -\frac{\partial v}{\partial x}$$

and so can simplify the nonlinear term to get

$$\begin{aligned} &\frac{\partial}{\partial x}\left[\left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right)\right] - \frac{\partial}{\partial y}\left[\left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right)\right] \\ =& \frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + u\frac{\partial^2 v}{\partial x^2} + \frac{\partial v}{\partial x}\frac{\partial v}{\partial y} + v\frac{\partial^2 v}{\partial x \partial y} - \frac{\partial u}{\partial y}\frac{\partial u}{\partial x} - u\frac{\partial^2 u}{\partial x \partial y} - \frac{\partial v}{\partial y}\frac{\partial u}{\partial y} - v\frac{\partial^2 u}{\partial y^2} \\ =& u\left(\frac{\partial^2 v}{\partial x^2} - \frac{\partial^2 u}{\partial x \partial y}\right) + v\left(\frac{\partial^2 v}{\partial x \partial y} - \frac{\partial^2 u}{\partial^2 y}\right). \end{aligned}$$

[1]A streamline is a continuous curve along which the instantaneous velocity is tangent, see Tritton [58] for more on this.

We finally obtain

$$\rho\left(\frac{\partial\omega}{\partial t}+u\frac{\partial\omega}{\partial x}+v\frac{\partial\omega}{\partial y}\right)=\mu\Delta\omega+\frac{\partial fy}{\partial x}-\frac{\partial fx}{\partial y} \tag{13.3}$$

and

$$\Delta\psi=-\omega. \tag{13.4}$$

Note that in this formulation, the Navier-Stokes equation is like a forced heat equation for the vorticity with a nonlocal and nonlinear term. We can take advantage of this structure in finding numerical solutions by modifying our numerical programs which give approximate solutions to the heat equation.

A simple time discretization for this equation is the Crank-Nicolson method, where the nonlinear terms are solved for using fixed point iteration. A tutorial on convergence of time discretization schemes for the Navier-Stokes equations can be found in Temam [54]. The time discretized equations become

$$\begin{aligned}&\rho\Bigg[\frac{\omega^{n+1,k+1}-\omega^n}{\delta t} \qquad (13.5)\\ &+\frac{1}{2}\left(u^{n+1,k}\frac{\partial\omega^{n+1,k}}{\partial x}+v^{n+1,k}\frac{\partial\omega^{n+1,k}}{\partial y}+u^n\frac{\partial\omega^n}{\partial x}+v^n\frac{\partial\omega^n}{\partial y}\right)\Bigg]\\ &=\frac{\mu}{2}\Delta\left(\omega^{n+1,k+1}+\omega^n\right)+\left.\left(\frac{\partial fx}{\partial y}-\frac{\partial fy}{\partial x}\right)\right|_{t=(n+0.5)\delta t},\end{aligned}$$

and

$$\Delta\psi^{n+1,k+1}=-\omega^{n+1,k+1},\quad u^{n+1,k+1}=\frac{\partial\psi^{n+1,k+1}}{\partial y},\quad v^{n+1,k+1}=-\frac{\partial\psi^{n+1,k+1}}{\partial x}. \tag{13.6}$$

In these equations, the superscript n denotes the timestep and the superscript k denotes the iterate. Another choice of time discretization is the implicit midpoint rule which gives,

$$\begin{aligned}&\rho\Bigg[\frac{\omega^{n+1,k+1}-\omega^n}{\delta t} \qquad (13.7)\\ &+\left(\frac{u^{n+1,k}+u^n}{2}\right)\frac{\partial}{\partial x}\left(\frac{\omega^{n+1,k}+\omega^n}{2}\right)+\left(\frac{v^{n+1,k}+v^n}{2}\right)\frac{\partial}{\partial y}\left(\frac{\omega^{n+1,k}+\omega^n}{2}\right)\Bigg]\\ &=\frac{\mu}{2}\Delta\left(\omega^{n+1,k+1}+\omega^n\right)+\left.\left(\frac{\partial fx}{\partial y}-\frac{\partial fy}{\partial x}\right)\right|_{t=(n+0.5)\delta t},\end{aligned}$$

and

$$\Delta\psi^{n+1,k+1}=-\omega^{n+1,k+1},\quad u^{n+1,k+1}=\frac{\partial\psi^{n+1,k+1}}{\partial y},\quad v^{n+1,k+1}=-\frac{\partial\psi^{n+1,k+1}}{\partial x}. \tag{13.8}$$

13.3 The Three-Dimensional Case

Here $\mathbf{u} = (u(x,y,z,t), v(x,y,z,t), w(x,y,z,t))$ – unfortunately, it is not clear if this equation has a unique solution for reasonable boundary conditions and initial data. Numerical methods so far seem to indicate that the solution is unique, but in the absence of a proof, we caution the reader that we are *fearless engineers writing gigantic codes that are supposed to produce solutions to the Navier-Stokes equations when what we are really studying is the output of the algorithm* which we hope will tell us something about these equations[2] – in practice, although the mathematical foundations for this are uncertain, these codes do seem to give information about the motion of nearly incompressible fluids in many, although not all situations of practical interest. Further information on this aspect of these equations can be found in Doering and Gibbon [15].

We will again consider simulations with periodic boundary conditions to make it easy to apply the Fourier transform. This also makes it easier to enforce the incompressibility constraint by using an idea due to Orszag and Patterson [48] and also explained in Canuto et al. [9, p. 99]. If we take the divergence of the Navier-Stokes equations, we get

$$\nabla \cdot (\mathbf{u} \cdot \nabla \mathbf{u}) = -\Delta p \tag{13.9}$$

because $\nabla \cdot \mathbf{u} = 0$. Hence

$$p = -\Delta^{-1} \left[\nabla \cdot (\mathbf{u} \cdot \nabla \mathbf{u}) \right] \tag{13.10}$$

where Δ^{-1} is defined using the Fourier transform, thus if $f(x,y,z)$ is a mean zero, periodic scalar field and $\hat{f}$ is its Fourier transform, then

$$\widehat{\Delta^{-1} f} = \frac{\hat{f}}{k_x^2 + k_y^2 + k_z^2}$$

where k_x, k_y and k_z are the wavenumbers. The Navier-Stokes equations then become

$$\frac{\partial \mathbf{u}}{\partial t} = \frac{1}{\mathrm{Re}} \Delta \mathbf{u} - \mathbf{u} \cdot \nabla \mathbf{u} + \nabla \Delta^{-1} \left[\nabla \cdot (\mathbf{u} \cdot \nabla \mathbf{u}) \right], \tag{13.11}$$

for which the incompressibility constraint is satisfied, provided the initial data satisfy the incompressibility constraint.

To discretize (13.11) in time, we will use the implicit midpoint rule. This gives,

$$\begin{aligned} \frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\delta t} =& \frac{0.5}{\mathrm{Re}} \Delta \left(\frac{\mathbf{u}^{n+1} + \mathbf{u}^n}{2} \right) - 0.25 \left(\mathbf{u}^{n+1} + \mathbf{u}^n \right) \cdot \nabla \left(\mathbf{u}^{n+1} + \mathbf{u}^n \right) \\ & + 0.25 \nabla \left[\Delta^{-1} \left(\nabla \cdot \left[\left(\mathbf{u}^{n+1} + \mathbf{u}^n \right) \cdot \nabla \left(\mathbf{u}^{n+1} + \mathbf{u}^n \right) \right] \right) \right]. \end{aligned} \tag{13.12}$$

[2]This is paraphrased from Gallavoti[19, p. VIII]

It is helpful to test the correctness of the programs by comparing them to an exact solution. Shapiro [51] has found the following exact solution which is a good test for meteorological hurricane simulation programs, as well as for Navier-Stokes solvers with periodic boundary conditions

$$u = -\frac{A}{k^2+l^2}\left[\lambda l\cos(kx)\sin(ly)\sin(mz) + mk\sin(kx)\cos(ly)\cos(mz)\right]\exp\left(-\frac{\lambda^2 t}{\mathrm{Re}}\right)$$
$$v = \frac{A}{k^2+l^2}\left[\lambda k\sin(kx)\cos(ly)\sin(mz) - ml\cos(kx)\sin(ly)\cos(mz)\right]\exp\left(-\frac{\lambda^2 t}{\mathrm{Re}}\right)$$
$$w = A\cos(kx)\cos(ly)\sin(mz)\exp\left(-\frac{\lambda^2 t}{\mathrm{Re}}\right)$$

where the constant $\lambda = \sqrt{k^2+l^2+m^2}$ and l, k and m are constants choosen with the restriction that the solutions are periodic in space. Further examples of such solutions can be found in Majda and Bertozzi [42, sec. 2.3].

13.4 Serial Programs

We first write Matlab programs to demonstrate how to solve these equations on a single processor. The first program uses Crank-Nicolson timestepping to solve the two-dimensional Navier-Stokes equations and is in listing 13.1. To test the program, following Laizet and Lamballais[34] we use the exact Taylor-Green vortex solution on $(x, y) \in [0, 1] \times [0, 1]$ with periodic boundary conditions given by

$$u(x, y, t) = \sin(2\pi x)\cos(2\pi y)\exp(-8\pi^2\mu t) \quad (13.13)$$
$$v(x, y, t) = -\cos(2\pi x)\sin(2\pi y)\exp(-8\pi^2\mu t). \quad (13.14)$$

Listing 13.1: A Matlab program which finds a numerical solution to the 2D Navier Stokes equation.

```
% Numerical solution of the 2D incompressible Navier-Stokes on a
% Square Domain [0,1]x[0,1] using a Fourier pseudo-spectral method
% and Crank-Nicolson timestepping. The numerical solution is compared to
% the exact Taylor-Green Vortex solution of the Navier-Stokes equations
%
%Periodic free-slip boundary conditions and Initial conditions:
    %u(x,y,0)=sin(2*pi*x)cos(2*pi*y)
    %v(x,y,0)=-cos(2*pi*x)sin(2*pi*y)
%Analytical Solution:
    %u(x,y,t)=sin(2*pi*x)cos(2*pi*y)exp(-8*pi^2*t/Re)
    %v(x,y,t)=-cos(2*pi*x)sin(2*pi*y)exp(-8*pi^2*t/Re)
clear all; format compact; format short; clc; clf;

Re=1;%Reynolds number
```

```

%grid
Nx=64; h=1/Nx; x=h*(1:Nx);
Ny=64; h=1/Ny; y=h*(1:Ny)';
[xx,yy]=meshgrid(x,y);

%initial conditions
u=sin(2*pi*xx).*cos(2*pi*yy);
v=-cos(2*pi*xx).*sin(2*pi*yy);
u_y=-2*pi*sin(2*pi*xx).*sin(2*pi*yy);
v_x=2*pi*sin(2*pi*xx).*sin(2*pi*yy);
omega=v_x-u_y;

dt=0.0025; t(1)=0; tmax=.1;
nplots=ceil(tmax/dt);

%wave numbers for derivatives
k_x=2*pi*(1i*[(0:Nx/2-1)  0 1-Nx/2:-1]');
k_y=2*pi*(1i*[(0:Ny/2-1)  0 1-Ny/2:-1]);
k2x=k_x.^2;
k2y=k_y.^2;

%wave number grid for multiplying matricies
[kxx,kyy]=meshgrid(k2x,k2y);
[kx,ky]=meshgrid(k_x,k_y);

% use a high tolerance so time stepping errors
% are not dominated by errors in solution to nonlinear
% system
tol=10^(-10);

%compute \hat{\omega}^{n+1,k}
omegahat=fft2(omega);
%nonlinear term
nonlinhat=fft2(u.*ifft2(omegahat.*kx)+v.*ifft2(omegahat.*ky));
for i=1:nplots
    chg=1;
    % save old values
    uold=u; vold=v;  omegaold=omega; omegacheck=omega;
    omegahatold=omegahat; nonlinhatold=nonlinhat;
    while chg>tol
        %nonlinear {n+1,k}
        nonlinhat=fft2(u.*ifft2(omegahat.*kx)+v.*ifft2(omegahat.*ky));

        %Crank Nicolson timestepping
        omegahat=((1/dt + 0.5*(1/Re)*(kxx+kyy)).*omegahatold...
            -.5*(nonlinhatold+nonlinhat))...
            ./(1/dt -0.5*(1/Re)*(kxx+kyy));

        %compute \hat{\psi}^{n+1,k+1}
        psihat=-omegahat./(kxx+kyy);

        %NOTE: kxx+kyy has to be zero at the following points to avoid a
        % discontinuity. However, we suppose that the streamfunction has
        % mean value zero, so we set them equal to zero
        psihat(1,1)=0;
        psihat(Nx/2+1,Ny/2+1)=0;
        psihat(Nx/2+1,1)=0;
        psihat(1,Ny/2+1)=0;

        %computes {\psi}_x by differentiation via FFT
        dpsix = real(ifft2(psihat.*kx));
        %computes {\psi}_y by differentiation via FFT
        dpsiy = real(ifft2(psihat.*ky));

        u=dpsiy;    %u^{n+1,k+1}
        v=-dpsix;   %v^{n+1,k+1}

        %\omega^{n+1,k+1}
        omega=ifft2(omegahat);
        % check for convergence
        chg=max(max(abs(omega-omegacheck)))
        % store omega to check for convergence of next iteration
        omegacheck=omega;
    end
     t(i+1)=t(i)+dt;
     uexact_y=-2*pi*sin(2*pi*xx).*sin(2*pi*yy).*exp(-8*pi^2*t(i+1)/Re);
     vexact_x=2*pi*sin(2*pi*xx).*sin(2*pi*yy).*exp(-8*pi^2*t(i+1)/Re);
     omegaexact=vexact_x-uexact_y;
     figure(1); pcolor(omega);  xlabel x; ylabel y;
     title Numerical; colorbar; drawnow;
     figure(2); pcolor(omegaexact); xlabel x; ylabel y;
     title Exact; colorbar; drawnow;
     figure(3); pcolor(omega-omegaexact); xlabel x; ylabel y;
     title Error; colorbar; drawnow;
end
```

The second program uses the implicit midpoint rule to do timestepping for the three-dimensional Navier-Stokes equations and it is in listing 13.2. It also takes the Taylor-Green vortex as its initial condition since this has been extensively studied, and so provides a baseline case to compare results against.

Listing 13.2: A Matlab program which finds a numerical solution to the 3D Navier Stokes equation.

```
% A program to solve the 3D Navier stokes equations with periodic boundary
% conditions. The program is based on the Orszag-Patterson algorithm as
% documented on pg. 98 of C. Canuto, M.Y. Hussaini, A. Quarteroni and
% T.A. Zhang "Spectral Methods: Evolution to Complex Geometries and
% Applications to Fluid Dynamics" Springer (2007)
%
```

```
% The exact solution used to check the numerical method is in
% A. Shapiro "The use of an exact solution of the Navier-Stokes equations
% in a validation test of a three-dimensional nonhydrostatic numerical
% model" Monthly Weather Review vol. 121 pp. 2420-2425 (1993)

clear all; format compact; format short;
set(0,'defaultaxesfontsize',30,'defaultaxeslinewidth',.7,...
    'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,...
    'defaultaxesfontweight','bold')

% set up grid
tic
Lx = 1;          % period  2*pi*L
Ly = 1;          % period  2*pi*L
Lz = 1;          % period  2*pi*L
Nx = 64;         % number of harmonics
Ny = 64;         % number of harmonics
Nz = 64;         % number of harmonics
Nt = 10;        % number of time slices
dt = 0.2/Nt;     % time step
t=0;             % initial time
Re = 1.0;  % Reynolds number
tol=10^(-10);
% initialise variables
x = (2*pi/Nx)*(-Nx/2:Nx/2 -1)'*Lx;           % x coordinate
kx = 1i*[0:Nx/2-1 0 -Nx/2+1:-1]'/Lx;         % wave vector
y = (2*pi/Ny)*(-Ny/2:Ny/2 -1)'*Ly;           % y coordinate
ky = 1i*[0:Ny/2-1 0 -Ny/2+1:-1]'/Ly;         % wave vector
z = (2*pi/Nz)*(-Nz/2:Nz/2 -1)'*Lz;           % y coordinate
kz = 1i*[0:Nz/2-1 0 -Nz/2+1:-1]'/Lz;         % wave vector
[xx,yy,zz]=meshgrid(x,y,z);
[kxm,kym,kzm]=meshgrid(kx,ky,kz);
[k2xm,k2ym,k2zm]=meshgrid(kx.^2,ky.^2,kz.^2);

% initial conditions for Taylor-Green vortex
% theta=0;
% u=(2/sqrt(3))*sin(theta+2*pi/3)*sin(xx).*cos(yy).*cos(zz);
% v=(2/sqrt(3))*sin(theta-2*pi/3)*cos(xx).*sin(yy).*cos(zz);
% w=(2/sqrt(3))*sin(theta)*cos(xx).*cos(yy).*sin(zz);

% exact solution
sl=1; sk=1; sm=1; lamlkm=sqrt(sl.^2+sk.^2+sm.^2);
u=-0.5*(lamlkm*sl*cos(sk*xx).*sin(sl*yy).*sin(sm.*zz)...
            +sm*sk*sin(sk*xx).*cos(sl*yy).*cos(sm.*zz))...
            .*exp(-t*(lamlkm^2)/Re);

v=0.5*(lamlkm*sk*sin(sk*xx).*cos(sl*yy).*sin(sm.*zz)...
            -sm*sl*cos(sk*xx).*sin(sl*yy).*cos(sm.*zz))...
            *exp(-t*(lamlkm^2)/Re);

w=cos(sk*xx).*cos(sl*yy).*sin(sm*zz)*exp(-t*(lamlkm^2)/Re);
```

```
uhat=fftn(u);
vhat=fftn(v);
what=fftn(w);

ux=ifftn(uhat.*kxm);uy=ifftn(uhat.*kym);uz=ifftn(uhat.*kzm);
vx=ifftn(vhat.*kxm);vy=ifftn(vhat.*kym);vz=ifftn(vhat.*kzm);
wx=ifftn(what.*kxm);wy=ifftn(what.*kym);wz=ifftn(what.*kzm);

% calculate vorticity for plotting
omegax=wy-vz; omegay=uz-wx; omegaz=vx-uy;
omegatot=omegax.^2+omegay.^2+omegaz.^2;
figure(1); clf; n=0;
subplot(2,2,1); title(['omega x ',num2str(n*dt)]);
p1 = patch(isosurface(x,y,z,omegax,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
p2 = patch(isocaps(x,y,z,omegax,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
        isonormals(omegax,p1); lighting phong;
xlabel('x'); ylabel('y'); zlabel('z');
axis equal; axis square; view(3); colorbar;
subplot(2,2,2); title(['omega y ',num2str(n*dt)]);
p1 = patch(isosurface(x,y,z,omegay,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
p2 = patch(isocaps(x,y,z,omegay,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
        isonormals(omegay,p1); lighting phong;
xlabel('x'); ylabel('y'); zlabel('z');
axis equal; axis square; view(3); colorbar;
subplot(2,2,3); title(['omega z ',num2str(n*dt)]);
p1 = patch(isosurface(x,y,z,omegaz,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
p2 = patch(isocaps(x,y,z,omegaz,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
        isonormals(omegaz,p1); lighting phong;
xlabel('x'); ylabel('y'); zlabel('z');
axis equal; axis square; view(3); colorbar;
subplot(2,2,4); title(['|omega|^2 ',num2str(n*dt)]);
p1 = patch(isosurface(x,y,z,omegatot,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
p2 = patch(isocaps(x,y,z,omegatot,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
        isonormals(omegatot,p1); lighting phong;
xlabel('x'); ylabel('y'); zlabel('z'); colorbar;
axis equal; axis square; view(3);


for n=1:Nt
    uold=u; uxold=ux; uyold=uy; uzold=uz;
    vold=v; vxold=vx; vyold=vy; vzold=vz;
    wold=w; wxold=wx; wyold=wy; wzold=wz;
    rhsuhatfix=(1/dt+(0.5/Re)*(k2xm+k2ym+k2zm)).*uhat;
    rhsvhatfix=(1/dt+(0.5/Re)*(k2xm+k2ym+k2zm)).*vhat;
    rhswhatfix=(1/dt+(0.5/Re)*(k2xm+k2ym+k2zm)).*what;
    chg=1; t=t+dt;
    while (chg>tol)
        nonlinu=0.25*((u+uold).*(ux+uxold)...
                      +(v+vold).*(uy+uyold)...
                      +(w+wold).*(uz+uzold));
        nonlinv=0.25*((u+uold).*(vx+vxold)...
                      +(v+vold).*(vy+vyold)...
                      +(w+wold).*(vz+vzold));
        nonlinw=0.25*((u+uold).*(wx+wxold)...
                      +(v+vold).*(wy+wyold)...
                      +(w+wold).*(wz+wzold));
        nonlinuhat=fftn(nonlinu);
        nonlinvhat=fftn(nonlinv);
        nonlinwhat=fftn(nonlinw);
        phat=-1.0*(kxm.*nonlinuhat+kym.*nonlinvhat+kzm.*nonlinwhat)...
            ./(k2xm+k2ym+k2zm+0.1^13);
        uhat=(rhsuhatfix-nonlinuhat-kxm.*phat)...
            ./(1/dt - (0.5/Re)*(k2xm+k2ym+k2zm));
        vhat=(rhsvhatfix-nonlinvhat-kym.*phat)...
            ./(1/dt - (0.5/Re)*(k2xm+k2ym+k2zm));
        what=(rhswhatfix-nonlinwhat-kzm.*phat)...
            ./(1/dt - (0.5/Re)*(k2xm+k2ym+k2zm));
        ux=ifftn(uhat.*kxm);uy=ifftn(uhat.*kym);uz=ifftn(uhat.*kzm);
        vx=ifftn(vhat.*kxm);vy=ifftn(vhat.*kym);vz=ifftn(vhat.*kzm);
        wx=ifftn(what.*kxm);wy=ifftn(what.*kym);wz=ifftn(what.*kzm);
        utemp=u; vtemp=v; wtemp=w;
        u=ifftn(uhat); v=ifftn(vhat); w=ifftn(what);
        chg=max(abs(utemp-u))+max(abs(vtemp-v))+max(abs(wtemp-w));
    end
    % calculate vorticity for plotting
    omegax=wy-vz; omegay=uz-wx; omegaz=vx-uy;
    omegatot=omegax.^2+omegay.^2+omegaz.^2;
    figure(1); clf;
    subplot(2,2,1); title(['omega x ',num2str(t)]);
    p1 = patch(isosurface(x,y,z,omegax,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
    p2 = patch(isocaps(x,y,z,omegax,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
        isonormals(omegax,p1); lighting phong;
    xlabel('x'); ylabel('y'); zlabel('z');
    axis equal; axis square; view(3); colorbar;
    subplot(2,2,2); title(['omega y ',num2str(t)]);
    p1 = patch(isosurface(x,y,z,omegay,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
    p2 = patch(isocaps(x,y,z,omegay,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
        isonormals(omegay,p1); lighting phong;
    xlabel('x'); ylabel('y'); zlabel('z');
    axis equal; axis square; view(3); colorbar;
    subplot(2,2,3); title(['omega z ',num2str(t)]);
    p1 = patch(isosurface(x,y,z,omegaz,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
    p2 = patch(isocaps(x,y,z,omegaz,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
        isonormals(omegaz,p1); lighting phong;
    xlabel('x'); ylabel('y'); zlabel('z');
    axis equal; axis square; view(3); colorbar;
    subplot(2,2,4); title(['|omega|^2 ',num2str(t)]);
    p1 = patch(isosurface(x,y,z,omegatot,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
    p2 = patch(isocaps(x,y,z,omegatot,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
        isonormals(omegatot,p1); lighting phong;
    xlabel('x'); ylabel('y'); zlabel('z'); colorbar;
    axis equal; axis square; view(3);
end
toc

uexact=-0.5*(lamlkm*sl*cos(sk*xx).*sin(sl*yy).*sin(sm.*zz)...
            +sm*sk*sin(sk*xx).*cos(sl*yy).*cos(sm.*zz))...
            .*exp(-t*(lamlkm^2)/Re);

vexact=0.5*(lamlkm*sk*sin(sk*xx).*cos(sl*yy).*sin(sm.*zz)...
            -sm*sl*cos(sk*xx).*sin(sl*yy).*cos(sm.*zz))...
            *exp(-t*(lamlkm^2)/Re);

wexact=cos(sk*xx).*cos(sl*yy).*sin(sm*zz)*exp(-t*(lamlkm^2)/Re);


error=  max(max(max(abs(u-uexact))))+...
        max(max(max(abs(v-vexact))))+...
        max(max(max(abs(w-wexact))))
```

13.4.1 Exercises

1) Show that for the Taylor-Green vortex solution, the nonlinear terms in the two-dimensional Navier-Stokes equations cancel out exactly.

2) Write a Matlab program that uses the implicit midpoint rule instead of the Crank-Nicolson method to obtain a solution to the 2D Navier-Stokes equations. Compare your numerical solution with the Taylor-Green vortex solution.

3) Write a Fortran program that uses the implicit midpoint rule instead of the Crank-Nicolson method to obtain a solution to the 2D Navier-Stokes equations. Compare your numerical solution with the Taylor-Green vortex solution.

4) Write a Matlab program that uses the Crank-Nicolson method instead of the implicit midpoint rule to obtain a solution to the 3D Navier-Stokes equations.

5) Write a Fortran program that uses the Crank-Nicolson method instead of the implicit midpoint rule to obtain a solution to the 3D Navier-Stokes equations.

6) The Navier-Stokes equations as written in eqs. (13.3) and (13.4) also satisfy further integral properties. In particular show that

   a)
   $$\frac{\rho}{2}\frac{\mathrm{d}}{\mathrm{d}t}\|\omega\|_{l^2}^2 = -\mu\|\nabla\omega\|_{l^2}^2,$$
   where
   $$\|\omega\|_{l^2}^2 = \int\int(\omega)^2\mathrm{d}x\mathrm{d}y$$
   and
   $$\|\nabla\omega\|_{l^2}^2 = \int\int(\nabla\omega)\cdot(\nabla\omega)\mathrm{d}x\mathrm{d}y.$$
   HINT: multiply the Eq. (13.3) by ω then integrate by parts.

   b) Show that part (a) implies that
   $$\|\omega(t=T)\|_{l^2}^2 - \|\omega(t=0)\|_{l^2}^2 = -\mu\int_0^T\|\nabla\omega\|_{l^2}^2\mathrm{d}t$$

   c) Part (b) gives a property one can check when integrating the 2D Navier-Stokes equations. We now show that the implicit midpoint rule satisfies an analogous property. Multiply eq. (13.7) by $0.5(\omega^{n+1}+\omega^n)$, integrate by parts in space, then sum over time to deduce that
   $$\|\omega^N\|_{l^2}^2 - \|\omega^0\|_{l^2}^2 = -\frac{\mu}{4}\sum_{n=0}^{N-1}\left\|\nabla\left(\omega^n+\omega^{n+1}\right)\right\|_{l^2}^2\delta t.$$

   d) Deduce that this implies that the implicit midpoint rule time stepping method is unconditionally stable, provided the nonlinear terms can be solved for[3].

13.5 Parallel Programs: OpenMP

Rather than give fully parallelized example programs, we instead give a simple implementation in Fortran of the Crank-Nicolson and implicit midpoint rule algorithms for the two-dimensional and three dimensional Navier-Stokes equations that were presented in Matlab. The program for the two-dimensional equations is presented in listing 13.3 and an example

[3]We have not demonstrated convergence of the spatial discretization, so this result assumes that the spatial discretization has not been done.

Matlab script to plot the resulting vorticity fields is in listing 13.4. This program is presented in listing 13.5 and an example Matlab script to plot the resulting vorticity fields is in listing 13.6.

Listing 13.3: A Fortran program to solve the 2D Navier-Stokes equations.

```
PROGRAM main
!-----------------------------------------------------------------------
!
!
! PURPOSE
!
! This program numerically solves the 2D incompressible Navier-Stokes
! on a Square Domain [0,1]x[0,1] using pseudo-spectral methods and
! Crank-Nicolson timestepping. The numerical solution is compared to
! the exact Taylor-Green Vortex Solution.
!
! Periodic free-slip boundary conditions and Initial conditions:
! u(x,y,0)=sin(2*pi*x)cos(2*pi*y)
! v(x,y,0)=-cos(2*pi*x)sin(2*pi*y)
! Analytical Solution:
! u(x,y,t)=sin(2*pi*x)cos(2*pi*y)exp(-8*pi^2*nu*t)
! v(x,y,t)=-cos(2*pi*x)sin(2*pi*y)exp(-8*pi^2*nu*t)
!
! .. Parameters ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  Nt        = number of timesteps to take
!  Tmax        = maximum simulation time
!  FFTW_IN_PLACE  = value for FFTW input
!  FFTW_MEASURE   = value for FFTW input
!  FFTW_EXHAUSTIVE  = value for FFTW input
!  FFTW_PATIENT   = value for FFTW input
!  FFTW_ESTIMATE  = value for FFTW input
!  FFTW_FORWARD     = value for FFTW input
!  FFTW_BACKWARD  = value for FFTW input
!  pi = 3.14159265358979323846264338327950288419716939937510d0
!  mu        = viscosity
!  rho        = density
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!  n         = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  count       = keep track of information written to disk
!  iol         = size of array to write to disk
!  start       = variable to record start time of program
!  finish      = variable to record end time of program
!  count_rate    = variable for clock count rate
!  planfx      = Forward 1d fft plan in x
!  planbx     = Backward 1d fft plan in x
!  planfy     = Forward 1d fft plan in y
!  planby     = Backward 1d fft plan in y
!  dt       = timestep
! .. Arrays ..
!  u        = velocity in x direction
!  uold       = velocity in x direction at previous timestep
!  v        = velocity in y direction
!  vold       = velocity in y direction at previous timestep
!  u_y        = y derivative of velocity in x direction
!  v_x        = x derivative of velocity in y direction
!  omeg       = vorticity in real space
!  omegold      = vorticity in real space at previous
!           iterate
!  omegcheck    = store of vorticity at previous iterate
!  omegoldhat   = 2D Fourier transform of vorticity at previous
!           iterate
!  omegoldhat_x   = x-derivative of vorticity in Fourier space
!           at previous iterate
!  omegold_x    = x-derivative of vorticity in real space
!           at previous iterate
!  omegoldhat_y   = y-derivative of vorticity in Fourier space
!           at previous iterate
!  omegold_y    = y-derivative of vorticity in real space
!            at previous iterate
!  nlold      = nonlinear term in real space
!           at previous iterate
!  nloldhat     = nonlinear term in Fourier space
!           at previous iterate
!  omeghat      = 2D Fourier transform of vorticity
!           at next iterate
!  omeghat_x    = x-derivative of vorticity in Fourier space
!           at next timestep
!  omeghat_y    = y-derivative of vorticity in Fourier space
!           at next timestep
!  omeg_x     = x-derivative of vorticity in real space
!           at next timestep
!  omeg_y     = y-derivative of vorticity in real space
!           at next timestep
! .. Vectors ..
!  kx       = fourier frequencies in x direction
!  ky       = fourier frequencies in y direction
!  kxx        = square of fourier frequencies in x direction
!  kyy        = square of fourier frequencies in y direction
!  x        = x locations
!  y        = y locations
!  time       = times at which save data
!  name_config    = array to store filename for data to be saved
!  fftfx      = array to setup x Fourier transform
!  fftbx      = array to setup y Fourier transform
! REFERENCES
```

```fortran
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! This program has not been optimized to use the least amount of memory
! but is intended as an example only for which all states can be saved
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
! FFTW3  -- Fast Fourier Transform in the West Library
!     (http://www.fftw.org/)
! declare variables

IMPLICIT NONE
INTEGER(kind=4), PARAMETER  ::  Nx=256
INTEGER(kind=4), PARAMETER  ::  Ny=256
REAL(kind=8), PARAMETER   ::  dt=0.00125
REAL(kind=8), PARAMETER &
  ::  pi=3.14159265358979323846264338327950288419716939937510
REAL(kind=8), PARAMETER   ::  rho=1.0d0
REAL(kind=8), PARAMETER   ::  mu=1.0d0
REAL(kind=8), PARAMETER   ::  tol=0.1d0**10
REAL(kind=8)        ::  chg
INTEGER(kind=4), PARAMETER  ::  nplots=50
REAL(kind=8), DIMENSION(:), ALLOCATABLE     ::  time
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE    ::  kx,kxx
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE    ::  ky,kyy
REAL(kind=8), DIMENSION(:), ALLOCATABLE     ::  x
REAL(kind=8), DIMENSION(:), ALLOCATABLE     ::  y
COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE  :: &
  u,uold,v,vold,u_y,v_x,omegold, omegcheck, omeg,&
  omegoldhat, omegoldhat_x, omegold_x,&
    omegoldhat_y, omegold_y, nlold, nloldhat,&
  omeghat, omeghat_x, omeghat_y, omeg_x, omeg_y,&
  nl, nlhat, psihat, psihat_x, psi_x, psihat_y, psi_y
REAL(kind=8),DIMENSION(:,:), ALLOCATABLE  ::  uexact_y,vexact_x,
   omegexact
INTEGER(kind=4)               ::  i,j,k,n, allocatestatus, count, iol
INTEGER(kind=4)               ::  start, finish, count_rate
INTEGER(kind=4), PARAMETER            ::  FFTW_IN_PLACE = 8,
   FFTW_MEASURE = 0, &
                           FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32,    &
                                      FFTW_ESTIMATE = 64
  INTEGER(kind=4),PARAMETER             ::  FFTW_FORWARD = -1,
     FFTW_BACKWARD=1
COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE  ::  fftfx,fftbx
```

```fortran
INTEGER(kind=8)                  ::  planfxy,planbxy
CHARACTER*100                  ::  name_config

CALL system_clock(start,count_rate)
ALLOCATE(time(1:nplots),kx(1:Nx),kxx(1:Nx),ky(1:Ny),kyy(1:Ny),x(1:Nx),y
    (1:Ny),&
     u(1:Nx,1:Ny),uold(1:Nx,1:Ny),v(1:Nx,1:Ny),vold(1:Nx,1:Ny),u_y(1:Nx
        ,1:Ny),&
     v_x(1:Nx,1:Ny),omegold(1:Nx,1:Ny),omegcheck(1:Nx,1:Ny), omeg(1:Nx,1:
        Ny),&
     omegoldhat(1:Nx,1:Ny),omegoldhat_x(1:Nx,1:Ny), omegold_x(1:Nx,1:Ny),
         &
     omegoldhat_y(1:Nx,1:Ny),omegold_y(1:Nx,1:Ny), nlold(1:Nx,1:Ny),
        nloldhat(1:Nx,1:Ny),&
     omeghat(1:Nx,1:Ny), omeghat_x(1:Nx,1:Ny), omeghat_y(1:Nx,1:Ny),
        omeg_x(1:Nx,1:Ny),&
     omeg_y(1:Nx,1:Ny), nl(1:Nx,1:Ny), nlhat(1:Nx,1:Ny), psihat(1:Nx,1:Ny
        ), &
     psihat_x(1:Nx,1:Ny), psi_x(1:Nx,1:Ny), psihat_y(1:Nx,1:Ny), psi_y(1:
        Nx,1:Ny),&
     uexact_y(1:Nx,1:Ny), vexact_x(1:Nx,1:Ny), omegexact(1:Nx,1:Ny),fftfx
        (1:Nx,1:Ny),&
     fftbx(1:Nx,1:Ny),stat=AllocateStatus)
IF (AllocateStatus .ne. 0) STOP
PRINT *,'allocated space'

! set up ffts
CALL dfftw_plan_dft_2d_(planfxy,Nx,Ny,fftfx(1:Nx,1:Ny),fftbx(1:Nx,1:Ny)
    ,&
       FFTW_FORWARD,FFTW_EXHAUSTIVE)
CALL dfftw_plan_dft_2d_(planbxy,Nx,Ny,fftbx(1:Nx,1:Ny),fftfx(1:Nx,1:Ny)
    ,&
     FFTW_BACKWARD,FFTW_EXHAUSTIVE)

! setup fourier frequencies in x-direction
DO i=1,1+Nx/2
  kx(i)= 2.0d0*pi*cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))
END DO
kx(1+Nx/2)=0.0d0
DO i = 1,Nx/2 -1
  kx(i+1+Nx/2)=-kx(1-i+Nx/2)
END DO
DO i=1,Nx
  kxx(i)=kx(i)*kx(i)
END DO
DO i=1,Nx
  x(i)=REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0))
END DO

! setup fourier frequencies in y-direction
DO j=1,1+Ny/2
```

```
    ky(j)= 2.0d0*pi*cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))
  END DO
  ky(1+Ny/2)=0.0d0
  DO j = 1,Ny/2 -1
    ky(j+1+Ny/2)=-ky(1-j+Ny/2)
  END DO
  DO j=1,Ny
    kyy(j)=ky(j)*ky(j)
  END DO
  DO j=1,Ny
    y(j)=REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0))
  END DO
  PRINT *,'Setup grid and fourier frequencies'


  DO j=1,Ny
    DO i=1,Nx
      u(i,j)=sin(2.0d0*pi*x(i))*cos(2.0d0*pi*y(j))
      v(i,j)=-cos(2.0d0*pi*x(i))*sin(2.0d0*pi*y(j))
      u_y(i,j)=-2.0d0*pi*sin(2.0d0*pi*x(i))*sin(2.0d0*pi*y(j))
      v_x(i,j)=2.0d0*pi*sin(2.0d0*pi*x(i))*sin(2.0d0*pi*y(j))
      omeg(i,j)=v_x(i,j)-u_y(i,j)
    END DO
  END DO

  ! Vorticity to Fourier Space
  CALL dfftw_execute_dft_(planfxy,omeg(1:Nx,1:Ny),omeghat(1:Nx,1:Ny))

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !!!!!!!!!!!!!!!Initial nonlinear term !!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  ! obtain \hat{\omega}_x^{n,k}
  DO j=1,Ny
    omeghat_x(1:Nx,j)=omeghat(1:Nx,j)*kx(1:Nx)
  END DO
  ! obtain \hat{\omega}_y^{n,k}
  DO i=1,Nx
    omeghat_y(i,1:Ny)=omeghat(i,1:Ny)*ky(1:Ny)
  END DO
  ! convert to real space
  CALL dfftw_execute_dft_(planbxy,omeghat_x(1:Nx,1:Ny),omeg_x(1:Nx,1:Ny))
  CALL dfftw_execute_dft_(planbxy,omeghat_y(1:Nx,1:Ny),omeg_y(1:Nx,1:Ny))
  ! compute nonlinear term in real space
  DO j=1,Ny
    nl(1:Nx,j)=u(1:Nx,j)*omeg_x(1:Nx,j)/REAL(Nx*Ny,kind(0d0))+&
               v(1:Nx,j)*omeg_y(1:Nx,j)/REAL(Nx*Ny,kind(0d0))
  END DO
  CALL dfftw_execute_dft_(planfxy,nl(1:Nx,1:Ny),nlhat(1:Nx,1:Ny))
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```fortran
time(1)=0.0d0
PRINT *,'Got initial data, starting timestepping'
DO n=1,nplots
  chg=1
  ! save old values
  uold(1:Nx,1:Ny)=u(1:Nx,1:Ny)
  vold(1:Nx,1:Ny)=v(1:Nx,1:Ny)
  omegold(1:Nx,1:Ny)=omeg(1:Nx,1:Ny)
  omegcheck(1:Nx,1:Ny)=omeg(1:Nx,1:Ny)
  omegoldhat(1:Nx,1:Ny)=omeghat(1:Nx,1:Ny)
  nloldhat(1:Nx,1:Ny)=nlhat(1:Nx,1:Ny)
  DO WHILE (chg>tol)
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!nonlinear fixed (n,k+1)!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    ! obtain \hat{\omega}_x^{n+1,k}
    DO j=1,Ny
      omeghat_x(1:Nx,j)=omeghat(1:Nx,j)*kx(1:Nx)
    END DO
    ! obtain \hat{\omega}_y^{n+1,k}
    DO i=1,Nx
      omeghat_y(i,1:Ny)=omeghat(i,1:Ny)*ky(1:Ny)
    END DO
    ! convert back to real space
    CALL dfftw_execute_dft_(planbxy,omeghat_x(1:Nx,1:Ny),omeg_x(1:Nx,1:
       Ny))
    CALL dfftw_execute_dft_(planbxy,omeghat_y(1:Nx,1:Ny),omeg_y(1:Nx,1:
       Ny))
    ! calculate nonlinear term in real space
    DO j=1,Ny
      nl(1:Nx,j)=u(1:Nx,j)*omeg_x(1:Nx,j)/REAL(Nx*Ny,kind(0d0))+&
        v(1:Nx,j)*omeg_y(1:Nx,j)/REAL(Nx*Ny,kind(0d0))
    END DO
    ! convert back to fourier
    CALL dfftw_execute_dft_(planfxy,nl(1:Nx,1:Ny),nlhat(1:Nx,1:Ny))
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    ! obtain \hat{\omega}^{n+1,k+1} with Crank Nicolson timestepping
    DO j=1,Ny
      omeghat(1:Nx,j)=( (1.0d0/dt+0.5d0*(mu/rho)*(kxx(1:Nx)+kyy(j)))&
          *omegoldhat(1:Nx,j) - 0.5d0*(nloldhat(1:Nx,j)+nlhat(1:Nx,j)))
             /&
            (1.0d0/dt-0.5d0*(mu/rho)*(kxx(1:Nx)+kyy(j)))
    END DO

    ! calculate \hat{\psi}^{n+1,k+1}
    DO j=1,Ny
      psihat(1:Nx,j)=-omeghat(1:Nx,j)/(kxx(1:Nx)+kyy(j))
    END DO
```

```
      psihat(1,1)=0.0d0
        psihat(Nx/2+1,Ny/2+1)=0.0d0
      psihat(Nx/2+1,1)=0.0d0
      psihat(1,Ny/2+1)=0.0d0

      ! obtain \psi_x^{n+1,k+1} and \psi_y^{n+1,k+1}
      DO j=1,Ny
        psihat_x(1:Nx,j)=psihat(1:Nx,j)*kx(1:Nx)
      END DO
      CALL dfftw_execute_dft_(planbxy,psihat_x(1:Nx,1:Ny),psi_x(1:Nx,1:Ny)
         )
      DO i=1,Nx
        psihat_y(i,1:Ny)=psihat(i,1:Ny)*ky(1:Ny)
      END DO
      CALL dfftw_execute_dft_(planbxy,psihat_y(1:Ny,1:Ny),psi_y(1:Ny,1:Ny)
         )
      DO j=1,Ny
        psi_x(1:Nx,j)=psi_x(1:Nx,j)/REAL(Nx*Ny,kind(0d0))
        psi_y(1:Nx,j)=psi_y(1:Nx,j)/REAL(Nx*Ny,kind(0d0))
      END DO

      ! obtain \omega^{n+1,k+1}
      CALL dfftw_execute_dft_(planbxy,omeghat(1:Nx,1:Ny),omeg(1:Nx,1:Ny))
      DO j=1,Ny
        omeg(1:Nx,j)=omeg(1:Nx,j)/REAL(Nx*Ny,kind(0d0))
      END DO

      ! obtain u^{n+1,k+1} and v^{n+1,k+1} using stream function (\psi) in
          real space
      DO j=1,Ny
        u(1:Nx,j)=psi_y(1:Nx,j)
        v(1:Nx,j)=-psi_x(1:Nx,j)
      END DO

      ! check for convergence
      chg=maxval(abs(omeg-omegcheck))
      ! saves {n+1,k+1} to {n,k} for next iteration
      omegcheck=omeg
    END DO
    time(n+1)=time(n)+dt
    PRINT *,'TIME ',time(n+1)
  END DO

  DO j=1,Ny
    DO i=1,Nx
      uexact_y(i,j)=-2.0d0*pi*sin(2.0d0*pi*x(i))*sin(2.0d0*pi*y(j))*&
                exp(-8.0d0*mu*(pi**2)*nplots*dt)
      vexact_x(i,j)=2.0d0*pi*sin(2.0d0*pi*x(i))*sin(2.0d0*pi*y(j))*&
                exp(-8.0d0*mu*(pi**2)*nplots*dt)
      omegexact(i,j)=vexact_x(i,j)-uexact_y(i,j)
    END DO
```

```
END DO

name_config = 'omegafinal.datbin'
INQUIRE(iolength=iol) omegexact(1,1)
OPEN(unit=11,FILE=name_config,form="unformatted", access="direct",recl=
   iol)
count = 1
DO j=1,Ny
  DO i=1,Nx
    WRITE(11,rec=count) REAL(omeg(i,j),KIND(0d0))
    count=count+1
  END DO
END DO
CLOSE(11)

name_config = 'omegaexactfinal.datbin'
OPEN(unit=11,FILE=name_config,form="unformatted", access="direct",recl=
   iol)
count = 1
DO j=1,Ny
  DO i=1,Nx
    WRITE(11,rec=count) omegexact(i,j)
    count=count+1
  END DO
END DO
CLOSE(11)

name_config = 'xcoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO i=1,Nx
  WRITE(11,*) x(i)
END DO
CLOSE(11)

name_config = 'ycoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,Ny
  WRITE(11,*) y(j)
END DO
CLOSE(11)

CALL dfftw_destroy_plan_(planfxy)
CALL dfftw_destroy_plan_(planbxy)
CALL dfftw_cleanup_()

DEALLOCATE(time,kx,kxx,ky,kyy,x,y,&
    u,uold,v,vold,u_y,v_x,omegold, omegcheck, omeg, &
    omegoldhat, omegoldhat_x, omegold_x,&
    omegoldhat_y, omegold_y, nlold, nloldhat,&
```

```
        omeghat, omeghat_x, omeghat_y, omeg_x, omeg_y,&
        nl, nlhat, psihat, psihat_x, psi_x, psihat_y, psi_y,&
        uexact_y,vexact_x,omegexact, &
        fftfx,fftbx,stat=AllocateStatus)
    IF (AllocateStatus .ne. 0) STOP
    PRINT *,'Program execution complete'
    END PROGRAM main
```

Listing 13.4: A Matlab program to plot the vorticity fields and error produced by listing 13.3.

```
% A program to create a plot of the computed results
% from the 2D Matlab Navier-Stokes solver

clear all; format compact, format short,
set(0,'defaultaxesfontsize',14,'defaultaxeslinewidth',.7,...
    'defaultlinelinewidth',2,'defaultpatchlinewidth',3.5);

% Load data
% Get coordinates
X=load('xcoord.dat');
Y=load('ycoord.dat');
% find number of grid points
Nx=length(X);
Ny=length(Y);

% reshape coordinates to allow easy plotting
[xx,yy]=ndgrid(X,Y);

%
% Open file and dataset using the default properties.
%
FILENUM=['omegafinal.datbin'];
FILEEXA=['omegaexactfinal.datbin'];
fidnum=fopen(FILENUM,'r');
[fnamenum,modenum,mformatnum]=fopen(fidnum);
fidexa=fopen(FILEEXA,'r');
[fnameexa,modeexa,mformatexa]=fopen(fidexa);
Num=fread(fidnum,Nx*Ny,'double',mformatnum);
Exa=fread(fidexa,Nx*Ny,'double',mformatexa);
Num=reshape(Num,Nx,Ny);
Exa=reshape(Exa,Nx,Ny);
% close files
fclose(fidnum);
fclose(fidexa);
%
% Plot data on the screen.
%
figure(2);clf;
subplot(3,1,1); contourf(xx,yy,Num);
```

```
title(['Numerical Solution ']);
colorbar; axis square;
subplot(3,1,2); contourf(xx,yy,Exa);
title(['Exact Solution ']);
colorbar; axis square;
subplot(3,1,3); contourf(xx,yy,Exa-Num);
title(['Error']);
colorbar; axis square;
drawnow;
```

Listing 13.5: A Fortran program to solve the 3D Navier-Stokes equations.

```
  PROGRAM main
  !
     ---------------------------------------------------------------------------------------------

  !
  !
  ! PURPOSE
  !
  ! This program numerically solves the 3D incompressible Navier-Stokes
  ! on a Cubic Domain [0,2pi]x[0,2pi]x[0,2pi] using pseudo-spectral
     methods and
  ! Implicit Midpoint rule timestepping. The numerical solution is
     compared to
  ! an exact solution reported by Shapiro
  !
  ! Analytical Solution:
  ! u(x,y,z,t)=-0.25*(cos(x)sin(y)sin(z)+sin(x)cos(y)cos(z))exp(-t/Re)
  ! v(x,y,z,t)= 0.25*(sin(x)cos(y)sin(z)-cos(x)sin(y)cos(z))exp(-t/Re)
  ! w(x,y,z,t)= 0.5*cos(x)cos(y)sin(z)exp(-t/Re)
  !
  ! .. Parameters ..
  !  Nx        = number of modes in x - power of 2 for FFT
  !  Ny        = number of modes in y - power of 2 for FFT
  !  Nz        = number of modes in z - power of 2 for FFT
  !  Nt        = number of timesteps to take
  !  Tmax        = maximum simulation time
  !  FFTW_IN_PLACE  = value for FFTW input
  !  FFTW_MEASURE   = value for FFTW input
  !  FFTW_EXHAUSTIVE  = value for FFTW input
  !  FFTW_PATIENT   = value for FFTW input
  !  FFTW_ESTIMATE  = value for FFTW input
  !  FFTW_FORWARD      = value for FFTW input
  !  FFTW_BACKWARD  = value for FFTW input
  !  pi = 3.14159265358979323846264338327950288419716939937510d0
  !  Re        = Reynolds number
  ! .. Scalars ..
  !  i         = loop counter in x direction
  !  j         = loop counter in y direction
```

```
!  k         = loop counter in z direction
!  n         = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  count      = keep track of information written to disk
!  iol        = size of array to write to disk
!  start      = variable to record start time of program
!  finish     = variable to record end time of program
!  count_rate   = variable for clock count rate
!  planfxyz     = Forward 3d fft plan
!  planbxyz     = Backward 3d fft plan
!  dt        = timestep
! .. Arrays ..
!  u         = velocity in x direction
!  v         = velocity in y direction
!  w         = velocity in z direction
!  uold        = velocity in x direction at previous timestep
!  vold        = velocity in y direction at previous timestep
!  wold        = velocity in z direction at previous timestep
!  ux        = x derivative of velocity in x direction
!  uy        = y derivative of velocity in x direction
!  uz        = z derivative of velocity in x direction
!  vx        = x derivative of velocity in y direction
!  vy        = y derivative of velocity in y direction
!  vz        = z derivative of velocity in y direction
!  wx        = x derivative of velocity in z direction
!  wy        = y derivative of velocity in z direction
!  wz        = z derivative of velocity in z direction
!  uxold       = x derivative of velocity in x direction
!  uyold       = y derivative of velocity in x direction
!  uzold       = z derivative of velocity in x direction
!  vxold       = x derivative of velocity in y direction
!  vyold       = y derivative of velocity in y direction
!  vzold       = z derivative of velocity in y direction
!  wxold       = x derivative of velocity in z direction
!  wyold       = y derivative of velocity in z direction
!  wzold       = z derivative of velocity in z direction
!  omeg        = vorticity in real space
!  omegold       = vorticity in real space at previous
!           iterate
!  omegcheck    = store of vorticity at previous iterate
!  omegoldhat    = 2D Fourier transform of vorticity at previous
!           iterate
!  omegoldhat_x    = x-derivative of vorticity in Fourier space
!           at previous iterate
!  omegold_x     = x-derivative of vorticity in real space
!           at previous iterate
!  omegoldhat_y    = y-derivative of vorticity in Fourier space
!           at previous iterate
!  omegold_y     = y-derivative of vorticity in real space
!            at previous iterate
!  nlold       = nonlinear term in real space
!             at previous iterate
!  nloldhat      = nonlinear term in Fourier space
!             at previous iterate
!  omeghat       = 2D Fourier transform of vorticity
!             at next iterate
!  omeghat_x     = x-derivative of vorticity in Fourier space
!             at next timestep
!  omeghat_y     = y-derivative of vorticity in Fourier space
!             at next timestep
!  omeg_x      = x-derivative of vorticity in real space
!             at next timestep
!  omeg_y      = y-derivative of vorticity in real space
!             at next timestep
! .. Vectors ..
!  kx        = fourier frequencies in x direction
!  ky        = fourier frequencies in y direction
!  kz        = fourier frequencies in z direction
!  x         = x locations
!  y         = y locations
!  z         = y locations
!  time        = times at which save data
!  name_config     = array to store filename for data to be saved
!
! REFERENCES
!
! A. Shapiro " The use of an exact solution of the Navier-Stokes
   equations
! in a validation test of a three-dimensional nonhydrostatic numerical
   model"
! Monthly Weather Review vol. 121, 2420-2425, (1993).
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!
! This program has not been optimized to use the least amount of memory
! but is intended as an example only for which all states can be saved
!
!
   --------------------------------------------------------------------------------
! External routines required
!
! External libraries required
! FFTW3  -- Fast Fourier Transform in the West Library
!     (http://www.fftw.org/)
IMPLICIT NONE
```

```
  !declare variables
  INTEGER(kind=4), PARAMETER    :: Nx=64
INTEGER(kind=4), PARAMETER    :: Ny=64
INTEGER(kind=4), PARAMETER    :: Nz=64
  INTEGER(kind=4), PARAMETER    :: Lx=1
INTEGER(kind=4), PARAMETER    :: Ly=1
INTEGER(kind=4), PARAMETER    :: Lz=1
INTEGER(kind=4), PARAMETER    :: Nt=20
REAL(kind=8), PARAMETER     :: dt=0.2d0/Nt
REAL(kind=8), PARAMETER     :: Re=1.0d0
REAL(kind=8), PARAMETER     :: tol=0.1d0**10
REAL(kind=8), PARAMETER     :: theta=0.0d0

REAL(kind=8), PARAMETER &
  ::  pi=3.14159265358979323846264338327950288419716939937510d0
REAL(kind=8), PARAMETER   ::  ReInv=1.0d0/REAL(Re,kind(0d0))
REAL(kind=8), PARAMETER   :: dtInv=1.0d0/REAL(dt,kind(0d0))
REAL(kind=8)                  :: scalemodes,chg,factor
REAL(kind=8), DIMENSION(:), ALLOCATABLE     :: x, y, z, time
COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE  :: u, v, w,&
                          ux, uy, uz,&
                          vx, vy, vz,&
                          wx, wy, wz,&
                          uold, uxold, uyold, uzold,&
                          vold, vxold, vyold, vzold,&
                          wold, wxold, wyold, wzold,&
                          utemp, vtemp, wtemp, temp_r

COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE    :: kx, ky, kz
COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE  :: uhat, vhat, what,&
                        rhsuhatfix, rhsvhatfix,&
                        rhswhatfix, nonlinuhat,&
                        nonlinvhat, nonlinwhat,&
                        phat,temp_c
REAL(kind=8), DIMENSION(:,:,:), ALLOCATABLE   :: realtemp
!FFTW variables
INTEGER(kind=4)             :: ierr
INTEGER(kind=4), PARAMETER          :: FFTW_IN_PLACE = 8,&
                      FFTW_MEASURE = 0,&
                        FFTW_EXHAUSTIVE = 8,&
                      FFTW_PATIENT = 32,&
                                  FFTW_ESTIMATE = 64
  INTEGER(kind=4),PARAMETER           :: FFTW_FORWARD = -1,&
                      FFTW_BACKWARD=1
INTEGER(kind=8)             :: planfxyz,planbxyz

!variables used for saving data and timing
INTEGER(kind=4)             :: count, iol
INTEGER(kind=4)             :: i,j,k,n,t,allocatestatus
INTEGER(kind=4)             :: ind, numberfile
CHARACTER*100             :: name_config
```

```
INTEGER(kind=4)             ::  start, finish, count_rate

  PRINT *,'Grid:',Nx,'X',Ny,'Y',Nz,'Z'
PRINT *,'dt:',dt
ALLOCATE(x(1:Nx),y(1:Ny),z(1:Nz),time(1:Nt+1),u(1:Nx,1:Ny,1:Nz),&
      v(1:Nx,1:Ny,1:Nz), w(1:Nx,1:Ny,1:Nz), ux(1:Nx,1:Ny,1:Nz),&
      uy(1:Nx,1:Ny,1:Nz), uz(1:Nx,1:Ny,1:Nz), vx(1:Nx,1:Ny,1:Nz),&
      vy(1:Nx,1:Ny,1:Nz), vz(1:Nx,1:Ny,1:Nz), wx(1:Nx,1:Ny,1:Nz),&
      wy(1:Nx,1:Ny,1:Nz), wz(1:Nx,1:Ny,1:Nz), uold(1:Nx,1:Ny,1:Nz),&
      uxold(1:Nx,1:Ny,1:Nz), uyold(1:Nx,1:Ny,1:Nz), uzold(1:Nx,1:Ny,1:Nz
         ),&
      vold(1:Nx,1:Ny,1:Nz), vxold(1:Nx,1:Ny,1:Nz), vyold(1:Nx,1:Ny,1:Nz)
         ,&
      vzold(1:Nx,1:Ny,1:Nz), wold(1:Nx,1:Ny,1:Nz), wxold(1:Nx,1:Ny,1:Nz)
         ,&
      wyold(1:Nx,1:Ny,1:Nz), wzold(1:Nx,1:Ny,1:Nz), utemp(1:Nx,1:Ny,1:Nz
         ),&
      vtemp(1:Nx,1:Ny,1:Nz), wtemp(1:Nx,1:Ny,1:Nz), temp_r(1:Nx,1:Ny,1:
         Nz),&
      kx(1:Nx),ky(1:Ny),kz(1:Nz),uhat(1:Nx,1:Ny,1:Nz), vhat(1:Nx,1:Ny,1:
         Nz),&
      what(1:Nx,1:Ny,1:Nz), rhsuhatfix(1:Nx,1:Ny,1:Nz),&
      rhsvhatfix(1:Nx,1:Ny,1:Nz), rhswhatfix(1:Nx,1:Ny,1:Nz),&
      nonlinuhat(1:Nx,1:Ny,1:Nz), nonlinvhat(1:Nx,1:Ny,1:Nz),&
      nonlinwhat(1:Nx,1:Ny,1:Nz), phat(1:Nx,1:Ny,1:Nz),temp_c(1:Nx,1:Ny
         ,1:Nz),&
      realtemp(1:Nx,1:Ny,1:Nz), stat=AllocateStatus)
IF (AllocateStatus .ne. 0) STOP
PRINT *,'allocated space'

CALL dfftw_plan_dft_3d_(planfxyz,Nx,Ny,Nz,temp_r(1:Nx,1:Ny,1:Nz),&
    temp_c(1:Nx,1:Ny,1:Nz),FFTW_FORWARD,FFTW_ESTIMATE)
CALL dfftw_plan_dft_3d_(planbxyz,Nx,Ny,Nz,temp_c(1:Nx,1:Ny,1:Nz),&
    temp_r(1:Nx,1:Ny,1:Nz),FFTW_BACKWARD,FFTW_ESTIMATE)
PRINT *,'Setup 3D FFTs'

! setup fourier frequencies in x-direction
DO i=1,Nx/2+1
  kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
END DO
kx(1+Nx/2)=0.0d0
DO i = 1,Nx/2 -1
  kx(i+1+Nx/2)=-kx(1-i+Nx/2)
END DO
ind=1
DO i=-Nx/2,Nx/2-1
  x(ind)=2.0d0*pi*REAL(i,kind(0d0))*Lx/REAL(Nx,kind(0d0))
  ind=ind+1
END DO
! setup fourier frequencies in y-direction
DO j=1,Ny/2+1
```

```
    ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
  END DO
  ky(1+Ny/2)=0.0d0
  DO j = 1,Ny/2 -1
    ky(j+1+Ny/2)=-ky(1-j+Ny/2)
  END DO
  ind=1
  DO j=-Ny/2,Ny/2-1
    y(ind)=2.0d0*pi*REAL(j,kind(0d0))*Ly/REAL(Ny,kind(0d0))
    ind=ind+1
  END DO
  ! setup fourier frequencies in z-direction
  DO k=1,Nz/2+1
    kz(k)= cmplx(0.0d0,1.0d0)*REAL(k-1,kind(0d0))/Lz
  END DO
  kz(1+Nz/2)=0.0d0
  DO k = 1,Nz/2 -1
    kz(k+1+Nz/2)=-kz(1-k+Nz/2)
  END DO
  ind=1
  DO k=-Nz/2,Nz/2-1
    z(ind)=2.0d0*pi*REAL(k,kind(0d0))*Lz/REAL(Nz,kind(0d0))
    ind=ind+1
  END DO
  scalemodes=1.0d0/REAL(Nx*Ny*Nz,kind(0d0))
  PRINT *,'Setup grid and fourier frequencies'

  !initial conditions for Taylor-Green vortex
! factor=2.0d0/sqrt(3.0d0)
! DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
!   u(i,j,k)=factor*sin(theta+2.0d0*pi/3.0d0)*sin(x(i))*cos(y(j))*cos(z(k)
    )
! END DO; END DO; END DO
! DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
!   v(i,j,k)=factor*sin(theta-2.0d0*pi/3.0d0)*cos(x(i))*sin(y(j))*cos(z(k)
    )
! END DO ; END DO ; END DO
! DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
!   w(i,j,k)=factor*sin(theta)*cos(x(i))*cos(y(j))*sin(z(k))
! END DO ; END DO ; END DO

  ! Initial conditions for exact solution
  time(1)=0.0d0
  factor=sqrt(3.0d0)
  DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
    u(i,j,k)=-0.5*( factor*cos(x(i))*sin(y(j))*sin(z(k))&
            +sin(x(i))*cos(y(j))*cos(z(k)) )*exp(-(factor**2)*time(1)/Re)
  END DO; END DO; END DO
  DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
    v(i,j,k)=0.5*(  factor*sin(x(i))*cos(y(j))*sin(z(k))&
            -cos(x(i))*sin(y(j))*cos(z(k)) )*exp(-(factor**2)*time(1)/Re)
```

```
END DO ; END DO ; END DO
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  w(i,j,k)=cos(x(i))*cos(y(j))*sin(z(k))*exp(-(factor**2)*time(1)/Re)
END DO ; END DO ; END DO

CALL dfftw_execute_dft_(planfxyz,u(1:Nx,1:Ny,1:Nz),uhat(1:Nx,1:Ny,1:Nz))
CALL dfftw_execute_dft_(planfxyz,v(1:Nx,1:Ny,1:Nz),vhat(1:Nx,1:Ny,1:Nz))
CALL dfftw_execute_dft_(planfxyz,w(1:Nx,1:Ny,1:Nz),what(1:Nx,1:Ny,1:Nz))

! derivative of u with respect to x, y, and z
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  temp_c(i,j,k)=uhat(i,j,k)*kx(i)*scalemodes
END DO ; END DO ; END DO
CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),ux(1:Nx,1:Ny,1:
   Nz))
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  temp_c(i,j,k)=uhat(i,j,k)*ky(j)*scalemodes
END DO ; END DO ; END DO
CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),uy(1:Nx,1:Ny,1:
   Nz))
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  temp_c(i,j,k)=uhat(i,j,k)*kz(k)*scalemodes
END DO ; END DO ; END DO
CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),uz(1:Nx,1:Ny,1:
   Nz))

! derivative of v with respect to x, y, and z
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  temp_c(i,j,k)=vhat(i,j,k)*kx(i)*scalemodes
END DO ; END DO ; END DO
CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),vx(1:Nx,1:Ny,1:
   Nz))
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  temp_c(i,j,k)=vhat(i,j,k)*ky(j)*scalemodes
END DO ; END DO ; END DO
CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),vy(1:Nx,1:Ny,1:
   Nz))
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  temp_c(i,j,k)=vhat(i,j,k)*kz(k)*scalemodes
END DO ; END DO ; END DO
CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),vz(1:Nx,1:Ny,1:
   Nz))

! derivative of w with respect to x, y, and z
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  temp_c(i,j,k)=what(i,j,k)*kx(i)*scalemodes
END DO ; END DO ; END DO
CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),wx(1:Nx,1:Ny,1:
   Nz))
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  temp_c(i,j,k)=what(i,j,k)*ky(j)*scalemodes
```

```
END DO ; END DO ; END DO
CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),wy(1:Nx,1:Ny,1:
   Nz))
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  temp_c(i,j,k)=what(i,j,k)*kz(k)*scalemodes
END DO ; END DO ; END DO
CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),wz(1:Nx,1:Ny,1:
   Nz))
! save initial data
time(1)=0.0
n=0
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  realtemp(i,j,k)=REAL(wy(i,j,k)-vz(i,j,k),KIND=8)
END DO ; END DO ; END DO
name_config='./data/omegax'
CALL savedata(Nx,Ny,Nz,n,name_config,realtemp)
!omegay
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  realtemp(i,j,k)=REAL(uz(i,j,k)-wx(i,j,k),KIND=8)
END DO ; END DO ; END DO
name_config='./data/omegay'
CALL savedata(Nx,Ny,Nz,n,name_config,realtemp)
!omegaz
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  realtemp(i,j,k)=REAL(vx(i,j,k)-uy(i,j,k),KIND=8)
END DO ; END DO ; END DO
name_config='./data/omegaz'
CALL savedata(Nx,Ny,Nz,n,name_config,realtemp)

DO n=1,Nt
  !fixed point
  DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
    uold(i,j,k)=u(i,j,k)
    uxold(i,j,k)=ux(i,j,k)
    uyold(i,j,k)=uy(i,j,k)
    uzold(i,j,k)=uz(i,j,k)
  END DO ; END DO ; END DO
  DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
    vold(i,j,k)=v(i,j,k)
    vxold(i,j,k)=vx(i,j,k)
    vyold(i,j,k)=vy(i,j,k)
    vzold(i,j,k)=vz(i,j,k)
  END DO ; END DO ; END DO
  DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
    wold(i,j,k)=w(i,j,k)
    wxold(i,j,k)=wx(i,j,k)
    wyold(i,j,k)=wy(i,j,k)
    wzold(i,j,k)=wz(i,j,k)
  END DO ; END DO ; END DO
  DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
    rhsuhatfix(i,j,k) = (dtInv+(0.5d0*ReInv)*&
    (kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))*uhat(i,j,k)
  END DO ; END DO ; END DO
  DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
    rhsvhatfix(i,j,k) = (dtInv+(0.5d0*ReInv)*&
    (kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))*vhat(i,j,k)
  END DO ; END DO ; END DO
  DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
    rhswhatfix(i,j,k) = (dtInv+(0.5d0*ReInv)*&
    (kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))*what(i,j,k)
  END DO ; END DO ; END DO

  chg=1
  DO WHILE (chg .gt. tol)
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      temp_r(i,j,k)=0.25d0*((u(i,j,k)+uold(i,j,k))*(ux(i,j,k)+uxold(i,j,
         k))&
                   +(v(i,j,k)+vold(i,j,k))*(uy(i,j,k)+uyold(i,j,k))&
                   +(w(i,j,k)+wold(i,j,k))*(uz(i,j,k)+uzold(i,j,k)))
    END DO ; END DO ; END DO
    CALL dfftw_execute_dft_(planfxyz,temp_r(1:Nx,1:Ny,1:Nz),nonlinuhat
       (1:Nx,1:Ny,1:Nz))
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      temp_r(i,j,k)=0.25d0*((u(i,j,k)+uold(i,j,k))*(vx(i,j,k)+vxold(i,j,
         k))&
                   +(v(i,j,k)+vold(i,j,k))*(vy(i,j,k)+vyold(i,j,k))&
                   +(w(i,j,k)+wold(i,j,k))*(vz(i,j,k)+vzold(i,j,k)))
    END DO ; END DO ; END DO
    CALL dfftw_execute_dft_(planfxyz,temp_r(1:Nx,1:Ny,1:Nz),nonlinvhat
       (1:Nx,1:Ny,1:Nz))
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      temp_r(i,j,k)=0.25d0*((u(i,j,k)+uold(i,j,k))*(wx(i,j,k)+wxold(i,j,
         k))&
                   +(v(i,j,k)+vold(i,j,k))*(wy(i,j,k)+wyold(i,j,k))&
                   +(w(i,j,k)+wold(i,j,k))*(wz(i,j,k)+wzold(i,j,k)))
    END DO ; END DO ; END DO
    CALL dfftw_execute_dft_(planfxyz,temp_r(1:Nx,1:Ny,1:Nz),nonlinwhat
       (1:Nx,1:Ny,1:Nz))
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      phat(i,j,k)=-1.0d0*( kx(i)*nonlinuhat(i,j,k)&
                +ky(j)*nonlinvhat(i,j,k)&
                +kz(k)*nonlinwhat(i,j,k))&
                /(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)+0.1d0**13)
    END DO ; END DO ; END DO

    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      uhat(i,j,k)=(rhsuhatfix(i,j,k)-nonlinuhat(i,j,k)-kx(i)*phat(i,j,k)
         )/&
            (dtInv-(0.5d0*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))
               !*scalemodes
    END DO ; END DO ; END DO
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      vhat(i,j,k)=(rhsvhatfix(i,j,k)-nonlinvhat(i,j,k)-ky(j)*phat(i,j,k))/&
            (dtInv-(0.5d0*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k))) !*scalemodes
    END DO ; END DO ; END DO
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      what(i,j,k)=(rhswhatfix(i,j,k)-nonlinwhat(i,j,k)-kz(k)*phat(i,j,k))/&
            (dtInv-(0.5d0*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k))) !*scalemodes
    END DO ; END DO ; END DO

    ! derivative of u with respect to x, y, and z
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      temp_c(i,j,k)=uhat(i,j,k)*kx(i)*scalemodes
    END DO ; END DO ; END DO
    CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),ux(1:Nx,1:Ny,1:Nz))
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      temp_c(i,j,k)=uhat(i,j,k)*ky(j)*scalemodes
    END DO ; END DO ; END DO
    CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),uy(1:Nx,1:Ny,1:Nz))
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      temp_c(i,j,k)=uhat(i,j,k)*kz(k)*scalemodes
    END DO ; END DO ; END DO
    CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),uz(1:Nx,1:Ny,1:Nz))

    ! derivative of v with respect to x, y, and z
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      temp_c(i,j,k)=vhat(i,j,k)*kx(i)*scalemodes
    END DO ; END DO ; END DO
    CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),vx(1:Nx,1:Ny,1:Nz))
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      temp_c(i,j,k)=vhat(i,j,k)*ky(j)*scalemodes
    END DO ; END DO ; END DO
    CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),vy(1:Nx,1:Ny,1:Nz))
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      temp_c(i,j,k)=vhat(i,j,k)*kz(k)*scalemodes
    END DO ; END DO ; END DO
    CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),vz(1:Nx,1:Ny,1:Nz))

    ! derivative of w with respect to x, y, and z
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      temp_c(i,j,k)=what(i,j,k)*kx(i)*scalemodes
    END DO ; END DO ; END DO
```

```
      CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),wx(1:Nx,1:Ny
         ,1:Nz))
      DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
        temp_c(i,j,k)=what(i,j,k)*ky(j)*scalemodes
      END DO ; END DO ; END DO
      CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),wy(1:Nx,1:Ny
         ,1:Nz))
      DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
        temp_c(i,j,k)=what(i,j,k)*kz(k)*scalemodes
      END DO ; END DO ; END DO
      CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),wz(1:Nx,1:Ny
         ,1:Nz))

      DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
        utemp(i,j,k)=u(i,j,k)
      END DO ; END DO ; END DO
      DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
        vtemp(i,j,k)=v(i,j,k)
      END DO ; END DO ; END DO
      DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
        wtemp(i,j,k)=w(i,j,k)
      END DO ; END DO ; END DO

      CALL dfftw_execute_dft_(planbxyz,uhat(1:Nx,1:Ny,1:Nz),u(1:Nx,1:Ny,1:
         Nz))
      CALL dfftw_execute_dft_(planbxyz,vhat(1:Nx,1:Ny,1:Nz),v(1:Nx,1:Ny,1:
         Nz))
      CALL dfftw_execute_dft_(planbxyz,what(1:Nx,1:Ny,1:Nz),w(1:Nx,1:Ny,1:
         Nz))

      DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
        u(i,j,k)=u(i,j,k)*scalemodes
      END DO ; END DO ; END DO
      DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
        v(i,j,k)=v(i,j,k)*scalemodes
      END DO ; END DO ; END DO
      DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
        w(i,j,k)=w(i,j,k)*scalemodes
      END DO ; END DO ; END DO

      chg =maxval(abs(utemp-u))+maxval(abs(vtemp-v))+maxval(abs(wtemp-w))
      PRINT *,'chg:',chg
    END DO
    time(n+1)=n*dt
    PRINT *,'time',n*dt
    !NOTE: utemp, vtemp, and wtemp are just temporary space that can be
       used
    !    instead of creating new arrays.
    !omegax
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      realtemp(i,j,k)=REAL(wy(i,j,k)-vz(i,j,k),KIND=8)
    END DO ; END DO ; END DO
    name_config='./data/omegax'
    CALL savedata(Nx,Ny,Nz,n,name_config,realtemp)
    !omegay
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      realtemp(i,j,k)=REAL(uz(i,j,k)-wx(i,j,k),KIND=8)
    END DO ; END DO ; END DO
    name_config='./data/omegay'
    CALL savedata(Nx,Ny,Nz,n,name_config,realtemp)
    !omegaz
    DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
      realtemp(i,j,k)=REAL(vx(i,j,k)-uy(i,j,k),KIND=8)
    END DO ; END DO ; END DO
    name_config='./data/omegaz'
    CALL savedata(Nx,Ny,Nz,n,name_config,realtemp)
  END DO

  name_config = './data/tdata.dat'
  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
  REWIND(11)
  DO n=1,1+Nt
    WRITE(11,*) time(n)
  END DO
  CLOSE(11)

  name_config = './data/xcoord.dat'
  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
  REWIND(11)
  DO i=1,Nx
    WRITE(11,*) x(i)
  END DO
  CLOSE(11)

  name_config = './data/ycoord.dat'
  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
  REWIND(11)
  DO j=1,Ny
    WRITE(11,*) y(j)
  END DO
  CLOSE(11)

  name_config = './data/zcoord.dat'
  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
  REWIND(11)
  DO k=1,Nz
    WRITE(11,*) z(k)
  END DO
  CLOSE(11)
  PRINT *,'Saved data'

  ! Calculate error in final numerical solution
```

```
DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
  utemp(i,j,k)=u(i,j,k) -&
          (-0.5*( factor*cos(x(i))*sin(y(j))*sin(z(k))&
          +sin(x(i))*cos(y(j))*cos(z(k)) )*exp(-(factor**2)*time(Nt+1)/
             Re))
END DO; END DO; END DO
DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
  vtemp(i,j,k)=v(i,j,k) -&
        (0.5*(  factor*sin(x(i))*cos(y(j))*sin(z(k))&
          -cos(x(i))*sin(y(j))*cos(z(k)) )*exp(-(factor**2)*time(Nt+1)/
             Re))
END DO ; END DO ; END DO
DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
  wtemp(i,j,k)=w(i,j,k)-&
        (cos(x(i))*cos(y(j))*sin(z(k))*exp(-(factor**2)*time(Nt+1)/Re))
END DO ; END DO ; END DO
chg=maxval(abs(utemp))+maxval(abs(vtemp))+maxval(abs(wtemp))
PRINT*,'The error at the final timestep is',chg

CALL dfftw_destroy_plan_(planfxyz)
CALL dfftw_destroy_plan_(planbxyz)
DEALLOCATE(x,y,z,time,u,v,w,ux,uy,uz,vx,vy,vz,wx,wy,wz,uold,uxold,uyold,
   uzold,&
        vold,vxold,vyold,vzold,wold,wxold,wyold,wzold,utemp,vtemp,wtemp
           ,&
        temp_r,kx,ky,kz,uhat,vhat,what,rhsuhatfix,rhsvhatfix,&
        rhswhatfix,phat,nonlinuhat,nonlinvhat,nonlinwhat,temp_c,&
        realtemp,stat=AllocateStatus)
IF (AllocateStatus .ne. 0) STOP
PRINT *,'Program execution complete'
END PROGRAM main
```

Listing 13.6: A Matlab program to plot the vorticity fields produced by listing 13.5.

```
% A program to create a plot of the computed results
% from the 3D Fortran Navier-Stokes solver
clear all; format compact; format short;
set(0,'defaultaxesfontsize',30,'defaultaxeslinewidth',.7,...
    'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,...
    'defaultaxesfontweight','bold')
% Load data
% Get coordinates
tdata=load('./data/tdata.dat');
x=load('./data/xcoord.dat');
y=load('./data/ycoord.dat');
z=load('./data/zcoord.dat');
nplots = length(tdata);

Nx = length(x); Nt = length(tdata);
Ny = length(y); Nz = length(z);
```

```

% reshape coordinates to allow easy plotting
[yy,xx,zz]=meshgrid(x,y,z);

for i =1:nplots
    %
    % Open file and dataset using the default properties.
    %
    FILEX=['./data/omegax',num2str(9999999+i),'.datbin'];
    FILEY=['./data/omegay',num2str(9999999+i),'.datbin'];
    FILEZ=['./data/omegaz',num2str(9999999+i),'.datbin'];
    FILEPIC=['./data/pic',num2str(9999999+i),'.jpg'];
    fid=fopen(FILEX,'r');
    [fname,mode,mformat]=fopen(fid);
    omegax=fread(fid,Nx*Ny*Nz,'real*8');
    omegax=reshape(omegax,Nx,Ny,Nz);
    fclose(fid);
    fid=fopen(FILEY,'r');
    [fname,mode,mformat]=fopen(fid);
    omegay=fread(fid,Nx*Ny*Nz,'real*8');
    omegay=reshape(omegay,Nx,Ny,Nz);
    fclose(fid);
    fid=fopen(FILEZ,'r');
    [fname,mode,mformat]=fopen(fid);
    omegaz=fread(fid,Nx*Ny*Nz,'real*8');
    omegaz=reshape(omegaz,Nx,Ny,Nz);
    fclose(fid);
    %
    % Plot data on the screen.
    %
    omegatot=omegax.^2+omegay.^2+omegaz.^2;
    figure(100); clf;
    subplot(2,2,1); title(['omega x ',num2str(tdata(i))]);
    p1 = patch(isosurface(xx,yy,zz,omegax,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
    p2 = patch(isocaps(xx,yy,zz,omegax,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
        isonormals(omegax,p1); lighting phong;
    xlabel('x'); ylabel('y'); zlabel('z');
    axis equal; axis square; view(3); colorbar;
    subplot(2,2,2); title(['omega y ',num2str(tdata(i))]);
    p1 = patch(isosurface(xx,yy,zz,omegay,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
    p2 = patch(isocaps(xx,yy,zz,omegay,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
        isonormals(omegay,p1); lighting phong;
    xlabel('x'); ylabel('y'); zlabel('z');
    axis equal; axis square; view(3); colorbar;
    subplot(2,2,3); title(['omega z ',num2str(tdata(i))]);
    p1 = patch(isosurface(xx,yy,zz,omegaz,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
    p2 = patch(isocaps(xx,yy,zz,omegaz,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
        isonormals(omegaz,p1); lighting phong;
    xlabel('x'); ylabel('y'); zlabel('z');
    axis equal; axis square; view(3); colorbar;
    subplot(2,2,4); title(['|omega|^2 ',num2str(tdata(i))]);
    p1 = patch(isosurface(xx,yy,zz,omegatot,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
    p2 = patch(isocaps(xx,yy,zz,omegatot,.0025),...
            'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
        isonormals(omegatot,p1); lighting phong;
    xlabel('x'); ylabel('y'); zlabel('z'); colorbar;
    axis equal; axis square; view(3);
  saveas(100,FILEPIC);

end
```

13.5.1 Exercises

1) Verify that the program in listing 13.3 is second order accurate in time.

2) Use OpenMP directives to parallelize the example Fortran code for the two-dimensional Navier Stokes equations. Try and make it as efficient as possible.

3) Write another code which uses threaded FFTW to do the Fast Fourier transforms. This code should have a similar structure to the program in listing 12.11.

4) Use OpenMP directives to parallelize the example Fortran code for the three-dimensional Navier-Stokes equations in listing 13.5. Try and make it as efficient as possible.

5) Write another code which uses threaded FFTW to do the Fast Fourier transforms for the three-dimensional Navier-Stokes equations. This code should have a similar structure to the program in listing 12.11.

13.6 Parallel Programs: MPI

The code for this is very similar to the serial code in listing 13.3. For completeness and to allow one to see how to parallelize other programs, we include it. The program uses the library 2DECOMP&FFT. One difference between this program and the serial program is that a subroutine is included to write out data. Since this portion of the calculation is repeated several times, the program becomes more readable when the repeated code is placed in a subroutine. The subroutine is also generic enough that it can be reused in other programs, saving program developers time.

Listing 13.7: A parallel MPI Fortran program to solve the 3D Navier-Stokes equations.

```
PROGRAM main
!
    ----------------------------------------------------------------------------------
!
!
! PURPOSE
!
! This program numerically solves the 3D incompressible Navier-Stokes
! on a Cubic Domain [0,2pi]x[0,2pi]x[0,2pi] using pseudo-spectral
   methods and
! Implicit Midpoint rule timestepping. The numerical solution is
   compared to
! an exact solution reported by Shapiro
!
! Analytical Solution:
! u(x,y,z,t)=-0.25*(cos(x)sin(y)sin(z)+sin(x)cos(y)cos(z))exp(-t/Re)
! v(x,y,z,t)= 0.25*(sin(x)cos(y)sin(z)-cos(x)sin(y)cos(z))exp(-t/Re)
! w(x,y,z,t)= 0.5*cos(x)cos(y)sin(z)exp(-t/Re)
!
! .. Parameters ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  Nz        = number of modes in z - power of 2 for FFT
!  Nt        = number of timesteps to take
!  Tmax        = maximum simulation time
!  FFTW_IN_PLACE  = value for FFTW input
!  FFTW_MEASURE   = value for FFTW input
!  FFTW_EXHAUSTIVE  = value for FFTW input
!  FFTW_PATIENT   = value for FFTW input
!  FFTW_ESTIMATE  = value for FFTW input
!  FFTW_FORWARD     = value for FFTW input
!  FFTW_BACKWARD  = value for FFTW input
!  pi = 3.14159265358979323846264338327950288419716939937510d0
!  Re        = Reynolds number
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!  k         = loop counter in z direction
!  n         = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  count       = keep track of information written to disk
!  iol         = size of array to write to disk
!  start       = variable to record start time of program
!  finish      = variable to record end time of program
!  count_rate    = variable for clock count rate
!  planfxyz      = Forward 3d fft plan
!  planbxyz      = Backward 3d fft plan
!  dt        = timestep
! .. Arrays ..
!  u         = velocity in x direction
!  v         = velocity in y direction
!  w         = velocity in z direction
!  uold        = velocity in x direction at previous timestep
!  vold        = velocity in y direction at previous timestep
!  wold        = velocity in z direction at previous timestep
!  ux        = x derivative of velocity in x direction
!  uy        = y derivative of velocity in x direction
!  uz        = z derivative of velocity in x direction
!  vx        = x derivative of velocity in y direction
!  vy        = y derivative of velocity in y direction
!  vz        = z derivative of velocity in y direction
!  wx        = x derivative of velocity in z direction
!  wy        = y derivative of velocity in z direction
!  wz        = z derivative of velocity in z direction
!  uxold       = x derivative of velocity in x direction
!  uyold       = y derivative of velocity in x direction
!  uzold       = z derivative of velocity in x direction
!  vxold       = x derivative of velocity in y direction
!  vyold       = y derivative of velocity in y direction
!  vzold       = z derivative of velocity in y direction
!  wxold       = x derivative of velocity in z direction
!  wyold       = y derivative of velocity in z direction
!  wzold       = z derivative of velocity in z direction
!  utemp       = temporary storage of u to check convergence
!  vtemp       = temporary storage of u to check convergence
!  wtemp       = temporary storage of u to check convergence
!  temp_r      = temporary storage for untransformed variables
!  uhat        = Fourier transform of u
!  vhat        = Fourier transform of v
!  what        = Fourier transform of w
!  rhsuhatfix    = Fourier transform of righthand side for u for
    timestepping
!  rhsvhatfix    = Fourier transform of righthand side for v for
    timestepping
!  rhswhatfix    = Fourier transform of righthand side for w for
    timestepping
!  nonlinuhat    = Fourier transform of nonlinear term for u
!  nonlinvhat      = Fourier transform of nonlinear term for u
!  nonlinwhat    = Fourier transform of nonlinear term for u
!  phat        = Fourier transform of nonlinear term for pressure, p
!  temp_c      = temporary storage for Fourier transforms
!  realtemp      = Real storage
!
! .. Vectors ..
!  kx        = fourier frequencies in x direction
!  ky        = fourier frequencies in y direction
!  kz        = fourier frequencies in z direction
!  x         = x locations
!  y         = y locations
!  z         = y locations
!  time        = times at which save data
!  name_config    = array to store filename for data to be saved
!
! REFERENCES
!
! A. Shapiro " The use of an exact solution of the Navier-Stokes
   equations
! in a validation test of a three-dimensional nonhydrostatic numerical
   model"
! Monthly Weather Review vol. 121, 2420-2425, (1993).
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!
! This program has not been optimized to use the least amount of memory
! but is intended as an example only for which all states can be saved
!
!
   --------------------------------------------------------------------------------
! External routines required
!
! External libraries required
! 2DECOMP&FFT -- Fast Fourier Transform in the West Library
!     (http://2decomp.org/)

USE decomp_2d
USE decomp_2d_fft
USE decomp_2d_io
USE MPI
IMPLICIT NONE
! declare variables
  INTEGER(kind=4), PARAMETER    :: Nx=256
INTEGER(kind=4), PARAMETER    :: Ny=256
INTEGER(kind=4), PARAMETER    :: Nz=256
  INTEGER(kind=4), PARAMETER    :: Lx=1
INTEGER(kind=4), PARAMETER    :: Ly=1
INTEGER(kind=4), PARAMETER    :: Lz=1
INTEGER(kind=4), PARAMETER    :: Nt=20
REAL(kind=8), PARAMETER     :: dt=0.05d0/Nt
REAL(kind=8), PARAMETER     :: Re=1.0d0
REAL(kind=8), PARAMETER     :: tol=0.1d0**10
REAL(kind=8), PARAMETER     :: theta=0.0d0

REAL(kind=8), PARAMETER &
  ::  pi=3.14159265358979323846264338327950288419716939937510d0
```

```
REAL(kind=8), PARAMETER   ::  ReInv=1.0d0/REAL(Re,kind(0d0))
REAL(kind=8), PARAMETER   ::  dtInv=1.0d0/REAL(dt,kind(0d0))
REAL(kind=8)          :: scalemodes,chg,factor
REAL(kind=8), DIMENSION(:), ALLOCATABLE     :: x, y, z, time,mychg,
   allchg
COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE  :: u, v, w,&
                          ux, uy, uz,&
                          vx, vy, vz,&
                          wx, wy, wz,&
                          uold, uxold, uyold, uzold,&
                          vold, vxold, vyold, vzold,&
                          wold, wxold, wyold, wzold,&
                          utemp, vtemp, wtemp, temp_r

COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE    ::  kx, ky, kz
COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE  ::  uhat, vhat, what,&
                            rhsuhatfix, rhsvhatfix,&
                          rhswhatfix, nonlinuhat,&
                          nonlinvhat, nonlinwhat,&
                          phat,temp_c
REAL(kind=8), DIMENSION(:,:,:), ALLOCATABLE   ::  realtemp
! MPI and 2DECOMP variables
TYPE(DECOMP_INFO)               ::  decomp
INTEGER(kind=MPI_OFFSET_KIND)           ::  filesize, disp
INTEGER(kind=4)                 ::  p_row=0, p_col=0, numprocs, myid,
   ierr

! variables used for saving data and timing
INTEGER(kind=4)             :: count, iol
INTEGER(kind=4)             :: i,j,k,n,t,allocatestatus
INTEGER(kind=4)             :: ind, numberfile
CHARACTER*100             :: name_config
INTEGER(kind=4)             :: start, finish, count_rate

! initialisation of 2DECOMP&FFT
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
! do automatic domain decomposition
CALL decomp_2d_init(Nx,Ny,Nz,p_row,p_col)
! get information about domain decomposition choosen
CALL decomp_info_init(Nx,Ny,Nz,decomp)
! initialise FFT library
CALL decomp_2d_fft_init
IF (myid.eq.0) THEN
    PRINT *,'Grid:',Nx,'X',Ny,'Y',Nz,'Z'
  PRINT *,'dt:',dt
END IF
ALLOCATE(x(1:Nx),y(1:Ny),z(1:Nz),time(1:Nt+1),mychg(1:3),allchg(1:3),&
      u(decomp%xst(1):decomp%xen(1),&
          decomp%xst(2):decomp%xen(2),&
```

```
             decomp%xst(3):decomp%xen(3)),&
         v(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         w(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         ux(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         uy(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         uz(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         vx(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         vy(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         vz(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         wx(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         wy(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         wz(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         uold(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         uxold(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         uyold(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         uzold(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         vold(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
             decomp%xst(3):decomp%xen(3)),&
         vxold(decomp%xst(1):decomp%xen(1),&
             decomp%xst(2):decomp%xen(2),&
```

```
                decomp%xst(3):decomp%xen(3)),&
            vyold(decomp%xst(1):decomp%xen(1),&
                decomp%xst(2):decomp%xen(2),&
                decomp%xst(3):decomp%xen(3)),&
            vzold(decomp%xst(1):decomp%xen(1),&
                decomp%xst(2):decomp%xen(2),&
                decomp%xst(3):decomp%xen(3)),&
            wold(decomp%xst(1):decomp%xen(1),&
                decomp%xst(2):decomp%xen(2),&
                decomp%xst(3):decomp%xen(3)),&
            wxold(decomp%xst(1):decomp%xen(1),&
                decomp%xst(2):decomp%xen(2),&
                decomp%xst(3):decomp%xen(3)),&
            wyold(decomp%xst(1):decomp%xen(1),&
                decomp%xst(2):decomp%xen(2),&
                decomp%xst(3):decomp%xen(3)),&
            wzold(decomp%xst(1):decomp%xen(1),&
                decomp%xst(2):decomp%xen(2),&
                decomp%xst(3):decomp%xen(3)),&
            utemp(decomp%xst(1):decomp%xen(1),&
                decomp%xst(2):decomp%xen(2),&
                decomp%xst(3):decomp%xen(3)),&
            vtemp(decomp%xst(1):decomp%xen(1),&
                decomp%xst(2):decomp%xen(2),&
                decomp%xst(3):decomp%xen(3)),&
            wtemp(decomp%xst(1):decomp%xen(1),&
                decomp%xst(2):decomp%xen(2),&
                decomp%xst(3):decomp%xen(3)),&
            temp_r(decomp%xst(1):decomp%xen(1),&
                decomp%xst(2):decomp%xen(2),&
                decomp%xst(3):decomp%xen(3)),&
            kx(1:Nx),ky(1:Ny),kz(1:Nz),&
            uhat(decomp%zst(1):decomp%zen(1),&
                decomp%zst(2):decomp%zen(2),&
                decomp%zst(3):decomp%zen(3)),&
            vhat(decomp%zst(1):decomp%zen(1),&
                decomp%zst(2):decomp%zen(2),&
                decomp%zst(3):decomp%zen(3)),&
            what(decomp%zst(1):decomp%zen(1),&
                decomp%zst(2):decomp%zen(2),&
                decomp%zst(3):decomp%zen(3)),&
            rhsuhatfix(decomp%zst(1):decomp%zen(1),&
                decomp%zst(2):decomp%zen(2),&
                decomp%zst(3):decomp%zen(3)),&
            rhsvhatfix(decomp%zst(1):decomp%zen(1),&
                decomp%zst(2):decomp%zen(2),&
                decomp%zst(3):decomp%zen(3)),&
            rhswhatfix(decomp%zst(1):decomp%zen(1),&
                decomp%zst(2):decomp%zen(2),&
                decomp%zst(3):decomp%zen(3)),&
            nonlinuhat(decomp%zst(1):decomp%zen(1),&
```

```fortran
            decomp%zst(2):decomp%zen(2),&
            decomp%zst(3):decomp%zen(3)),&
        nonlinvhat(decomp%zst(1):decomp%zen(1),&
            decomp%zst(2):decomp%zen(2),&
            decomp%zst(3):decomp%zen(3)),&
        nonlinwhat(decomp%zst(1):decomp%zen(1),&
            decomp%zst(2):decomp%zen(2),&
            decomp%zst(3):decomp%zen(3)),&
        phat(decomp%zst(1):decomp%zen(1),&
            decomp%zst(2):decomp%zen(2),&
            decomp%zst(3):decomp%zen(3)),&
        temp_c(decomp%zst(1):decomp%zen(1),&
            decomp%zst(2):decomp%zen(2),&
            decomp%zst(3):decomp%zen(3)),&
        realtemp(decomp%xst(1):decomp%xen(1),&
            decomp%xst(2):decomp%xen(2),&
            decomp%xst(3):decomp%xen(3)), stat=AllocateStatus)
  IF (AllocateStatus .ne. 0) STOP
  IF (myid.eq.0) THEN
    PRINT *,'allocated space'
  END IF

  ! setup fourier frequencies in x-direction
  DO i=1,Nx/2+1
    kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
  END DO
  kx(1+Nx/2)=0.0d0
  DO i = 1,Nx/2 -1
    kx(i+1+Nx/2)=-kx(1-i+Nx/2)
  END DO
  ind=1
  DO i=-Nx/2,Nx/2-1
    x(ind)=2.0d0*pi*REAL(i,kind(0d0))*Lx/REAL(Nx,kind(0d0))
    ind=ind+1
  END DO
  ! setup fourier frequencies in y-direction
  DO j=1,Ny/2+1
    ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
  END DO
  ky(1+Ny/2)=0.0d0
  DO j = 1,Ny/2 -1
    ky(j+1+Ny/2)=-ky(1-j+Ny/2)
  END DO
  ind=1
  DO j=-Ny/2,Ny/2-1
    y(ind)=2.0d0*pi*REAL(j,kind(0d0))*Ly/REAL(Ny,kind(0d0))
    ind=ind+1
  END DO
  ! setup fourier frequencies in z-direction
  DO k=1,Nz/2+1
    kz(k)= cmplx(0.0d0,1.0d0)*REAL(k-1,kind(0d0))/Lz
  END DO
  kz(1+Nz/2)=0.0d0
  DO k = 1,Nz/2 -1
    kz(k+1+Nz/2)=-kz(1-k+Nz/2)
  END DO
  ind=1
  DO k=-Nz/2,Nz/2-1
    z(ind)=2.0d0*pi*REAL(k,kind(0d0))*Lz/REAL(Nz,kind(0d0))
    ind=ind+1
  END DO
  scalemodes=1.0d0/REAL(Nx*Ny*Nz,kind(0d0))
  IF (myid.eq.0) THEN
    PRINT *,'Setup grid and fourier frequencies'
  END IF

  !initial conditions for Taylor-Green vortex
! factor=2.0d0/sqrt(3.0d0)
! DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
   =decomp%xst(1),decomp%xen(1)
!   u(i,j,k)=factor*sin(theta+2.0d0*pi/3.0d0)*sin(x(i))*cos(y(j))*cos(z(k)
   )
! END DO; END DO; END DO
! DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
   =decomp%xst(1),decomp%xen(1)
!   v(i,j,k)=factor*sin(theta-2.0d0*pi/3.0d0)*cos(x(i))*sin(y(j))*cos(z(k)
   )
! END DO ; END DO ; END DO
! DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
   =decomp%xst(1),decomp%xen(1)
!   w(i,j,k)=factor*sin(theta)*cos(x(i))*cos(y(j))*sin(z(k))
! END DO ; END DO ; END DO

  time(1)=0.0d0
  factor=sqrt(3.0d0)
  DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
     =decomp%xst(1),decomp%xen(1)
    u(i,j,k)=-0.5*( factor*cos(x(i))*sin(y(j))*sin(z(k))&
            +sin(x(i))*cos(y(j))*cos(z(k)) )*exp(-(factor**2)*time(1)/Re)
  END DO; END DO; END DO
  DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
     =decomp%xst(1),decomp%xen(1)
    v(i,j,k)=0.5*(  factor*sin(x(i))*cos(y(j))*sin(z(k))&
            -cos(x(i))*sin(y(j))*cos(z(k)) )*exp(-(factor**2)*time(1)/Re)
  END DO ; END DO ; END DO
  DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
     =decomp%xst(1),decomp%xen(1)
    w(i,j,k)=cos(x(i))*cos(y(j))*sin(z(k))*exp(-(factor**2)*time(1)/Re)
  END DO ; END DO ; END DO

  CALL decomp_2d_fft_3d(u,uhat,DECOMP_2D_FFT_FORWARD)
  CALL decomp_2d_fft_3d(v,vhat,DECOMP_2D_FFT_FORWARD)
  CALL decomp_2d_fft_3d(w,what,DECOMP_2D_FFT_FORWARD)

  ! derivative of u with respect to x, y, and z
  DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
      i=decomp%zst(1),decomp%zen(1)
    temp_c(i,j,k)=uhat(i,j,k)*kx(i)*scalemodes
  END DO ; END DO ; END DO
  CALL decomp_2d_fft_3d(temp_c,ux,DECOMP_2D_FFT_BACKWARD)
  DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
      i=decomp%zst(1),decomp%zen(1)
    temp_c(i,j,k)=uhat(i,j,k)*ky(j)*scalemodes
  END DO ; END DO ; END DO
  CALL decomp_2d_fft_3d(temp_c,uy,DECOMP_2D_FFT_BACKWARD)
  DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
      i=decomp%zst(1),decomp%zen(1)
    temp_c(i,j,k)=uhat(i,j,k)*kz(k)*scalemodes
  END DO ; END DO ; END DO
  CALL decomp_2d_fft_3d(temp_c,uz,DECOMP_2D_FFT_BACKWARD)

  ! derivative of v with respect to x, y, and z
  DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
      i=decomp%zst(1),decomp%zen(1)
    temp_c(i,j,k)=vhat(i,j,k)*kx(i)*scalemodes
  END DO ; END DO ; END DO
  CALL decomp_2d_fft_3d(temp_c,vx,DECOMP_2D_FFT_BACKWARD)
  DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
      i=decomp%zst(1),decomp%zen(1)
    temp_c(i,j,k)=vhat(i,j,k)*ky(j)*scalemodes
  END DO ; END DO ; END DO
  CALL decomp_2d_fft_3d(temp_c,vy,DECOMP_2D_FFT_BACKWARD)
  DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
      i=decomp%zst(1),decomp%zen(1)
    temp_c(i,j,k)=vhat(i,j,k)*kz(k)*scalemodes
  END DO ; END DO ; END DO
  CALL decomp_2d_fft_3d(temp_c,vz,DECOMP_2D_FFT_BACKWARD)

  ! derivative of w with respect to x, y, and z
  DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
      i=decomp%zst(1),decomp%zen(1)
    temp_c(i,j,k)=what(i,j,k)*kx(i)*scalemodes
  END DO ; END DO ; END DO
  CALL decomp_2d_fft_3d(temp_c,wx,DECOMP_2D_FFT_BACKWARD)
  DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
      i=decomp%zst(1),decomp%zen(1)
    temp_c(i,j,k)=what(i,j,k)*ky(j)*scalemodes
  END DO ; END DO ; END DO
  CALL decomp_2d_fft_3d(temp_c,wy,DECOMP_2D_FFT_BACKWARD)
  DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
      i=decomp%zst(1),decomp%zen(1)
    temp_c(i,j,k)=what(i,j,k)*kz(k)*scalemodes
  END DO ; END DO ; END DO
```

```
CALL decomp_2d_fft_3d(temp_c,wz,DECOMP_2D_FFT_BACKWARD)
! save initial data
n=0
DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
   =decomp%xst(1),decomp%xen(1)
  realtemp(i,j,k)=REAL(wy(i,j,k)-vz(i,j,k),KIND=8)
END DO ; END DO ; END DO
name_config='./data/omegax'
CALL savedata(Nx,Ny,Nz,n,name_config,realtemp,decomp)
!omegay
DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
   =decomp%xst(1),decomp%xen(1)
  realtemp(i,j,k)=REAL(uz(i,j,k)-wx(i,j,k),KIND=8)
END DO ; END DO ; END DO
name_config='./data/omegay'
CALL savedata(Nx,Ny,Nz,n,name_config,realtemp,decomp)
!omegaz
DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
   =decomp%xst(1),decomp%xen(1)
  realtemp(i,j,k)=REAL(vx(i,j,k)-uy(i,j,k),KIND=8)
END DO ; END DO ; END DO
name_config='./data/omegaz'
CALL savedata(Nx,Ny,Nz,n,name_config,realtemp,decomp)

      !start timer
      CALL system_clock(start,count_rate)
DO n=1,Nt
  !fixed point
  DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO
      i=decomp%xst(1),decomp%xen(1)
    uold(i,j,k)=u(i,j,k)
    uxold(i,j,k)=ux(i,j,k)
    uyold(i,j,k)=uy(i,j,k)
    uzold(i,j,k)=uz(i,j,k)
  END DO ; END DO ; END DO
  DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO
      i=decomp%xst(1),decomp%xen(1)
    vold(i,j,k)=v(i,j,k)
    vxold(i,j,k)=vx(i,j,k)
    vyold(i,j,k)=vy(i,j,k)
    vzold(i,j,k)=vz(i,j,k)
  END DO ; END DO ; END DO
  DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO
      i=decomp%xst(1),decomp%xen(1)
    wold(i,j,k)=w(i,j,k)
    wxold(i,j,k)=wx(i,j,k)
    wyold(i,j,k)=wy(i,j,k)
    wzold(i,j,k)=wz(i,j,k)
  END DO ; END DO ; END DO
  DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ;
     DO i=decomp%zst(1),decomp%zen(1)
```

```
                    rhsuhatfix(i,j,k) = (dtInv+(0.5*ReInv)*(kx(i)*kx(i
                        )+ky(j)*ky(j)+kz(k)*kz(k)))*uhat(i,j,k)
  END DO ; END DO ; END DO
  DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ;
     DO i=decomp%zst(1),decomp%zen(1)
    rhsvhatfix(i,j,k) = (dtInv+(0.5*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k
       )*kz(k)))*vhat(i,j,k)
  END DO ; END DO ; END DO
  DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ;
     DO i=decomp%zst(1),decomp%zen(1)
    rhswhatfix(i,j,k) = (dtInv+(0.5*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k
       )*kz(k)))*what(i,j,k)
  END DO ; END DO ; END DO

  chg=1
  DO WHILE (chg .gt. tol)
    DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
       DO i=decomp%xst(1),decomp%xen(1)
      temp_r(i,j,k)=0.25d0*((u(i,j,k)+uold(i,j,k))*(ux(i,j,k)+uxold(i,j,
         k))&
                    +(v(i,j,k)+vold(i,j,k))*(uy(i,j,k)+uyold(i,j,k))&
                    +(w(i,j,k)+wold(i,j,k))*(uz(i,j,k)+uzold(i,j,k)))
    END DO ; END DO ; END DO
    CALL decomp_2d_fft_3d(temp_r,nonlinuhat,DECOMP_2D_FFT_FORWARD)
    DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
       DO i=decomp%xst(1),decomp%xen(1)
      temp_r(i,j,k)=0.25d0*((u(i,j,k)+uold(i,j,k))*(vx(i,j,k)+vxold(i,j,
         k))&
                    +(v(i,j,k)+vold(i,j,k))*(vy(i,j,k)+vyold(i,j,k))&
                    +(w(i,j,k)+wold(i,j,k))*(vz(i,j,k)+vzold(i,j,k)))
    END DO ; END DO ; END DO
    CALL decomp_2d_fft_3d(temp_r,nonlinvhat,DECOMP_2D_FFT_FORWARD)
    DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
       DO i=decomp%xst(1),decomp%xen(1)
      temp_r(i,j,k)=0.25d0*((u(i,j,k)+uold(i,j,k))*(wx(i,j,k)+wxold(i,j,
         k))&
                    +(v(i,j,k)+vold(i,j,k))*(wy(i,j,k)+wyold(i,j,k))&
                    +(w(i,j,k)+wold(i,j,k))*(wz(i,j,k)+wzold(i,j,k)))
    END DO ; END DO ; END DO
    CALL decomp_2d_fft_3d(temp_r,nonlinwhat,DECOMP_2D_FFT_FORWARD)
    DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
       ; DO i=decomp%zst(1),decomp%zen(1)
      phat(i,j,k)=-1.0d0*( kx(i)*nonlinuhat(i,j,k)&
                  +ky(j)*nonlinvhat(i,j,k)&
                  +kz(k)*nonlinwhat(i,j,k))&
                  /(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)+0.1d0**13)
    END DO ; END DO ; END DO

    DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
       ; DO i=decomp%zst(1),decomp%zen(1)
      uhat(i,j,k)=(rhsuhatfix(i,j,k)-nonlinuhat(i,j,k)-kx(i)*phat(i,j,k)
         )/&
              (dtInv-(0.5d0*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))
                 !*scalemodes
    END DO ; END DO ; END DO
    DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
       ; DO i=decomp%zst(1),decomp%zen(1)
      vhat(i,j,k)=(rhsvhatfix(i,j,k)-nonlinvhat(i,j,k)-ky(j)*phat(i,j,k)
         )/&
              (dtInv-(0.5d0*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))
                 !*scalemodes
    END DO ; END DO ; END DO
    DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
       ; DO i=decomp%zst(1),decomp%zen(1)
      what(i,j,k)=(rhswhatfix(i,j,k)-nonlinwhat(i,j,k)-kz(k)*phat(i,j,k)
         )/&
              (dtInv-(0.5d0*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))
                 !*scalemodes
    END DO ; END DO ; END DO

    ! derivative of u with respect to x, y, and z
    DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
       ; DO i=decomp%zst(1),decomp%zen(1)
      temp_c(i,j,k)=uhat(i,j,k)*kx(i)*scalemodes
    END DO ; END DO ; END DO
    CALL decomp_2d_fft_3d(temp_c,ux,DECOMP_2D_FFT_BACKWARD)
    DO k=decomp%zst(3),decomp%zen(3); DO j=decomp%zst(2),decomp%zen(2) ;
        DO i=decomp%zst(1),decomp%zen(1)
      temp_c(i,j,k)=uhat(i,j,k)*ky(j)*scalemodes
    END DO ; END DO ; END DO
    CALL decomp_2d_fft_3d(temp_c,uy,DECOMP_2D_FFT_BACKWARD)
    DO k=decomp%zst(3),decomp%zen(3); DO j=decomp%zst(2),decomp%zen(2) ;
        DO i=decomp%zst(1),decomp%zen(1)
      temp_c(i,j,k)=uhat(i,j,k)*kz(k)*scalemodes
    END DO ; END DO ; END DO
    CALL decomp_2d_fft_3d(temp_c,uz,DECOMP_2D_FFT_BACKWARD)

    ! derivative of v with respect to x, y, and z
    DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
       ; DO i=decomp%zst(1),decomp%zen(1)
      temp_c(i,j,k)=vhat(i,j,k)*kx(i)*scalemodes
    END DO ; END DO ; END DO
    CALL decomp_2d_fft_3d(temp_c,vx,DECOMP_2D_FFT_BACKWARD)
    DO k=decomp%zst(3),decomp%zen(3); DO j=decomp%zst(2),decomp%zen(2) ;
        DO i=decomp%zst(1),decomp%zen(1)
      temp_c(i,j,k)=vhat(i,j,k)*ky(j)*scalemodes
    END DO ; END DO ; END DO
    CALL decomp_2d_fft_3d(temp_c,vy,DECOMP_2D_FFT_BACKWARD)
    DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
       ; DO i=decomp%zst(1),decomp%zen(1)
      temp_c(i,j,k)=vhat(i,j,k)*kz(k)*scalemodes
    END DO ; END DO ; END DO
```

```fortran
    CALL decomp_2d_fft_3d(temp_c,vz,DECOMP_2D_FFT_BACKWARD)

    ! derivative of w with respect to x, y, and z
    DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
       ; DO i=decomp%zst(1),decomp%zen(1)
      temp_c(i,j,k)=what(i,j,k)*kx(i)*scalemodes
    END DO ; END DO ; END DO
    CALL decomp_2d_fft_3d(temp_c,wx,DECOMP_2D_FFT_BACKWARD)
    DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
       ; DO i=decomp%zst(1),decomp%zen(1)
      temp_c(i,j,k)=what(i,j,k)*ky(j)*scalemodes
    END DO ; END DO ; END DO
    CALL decomp_2d_fft_3d(temp_c,wy,DECOMP_2D_FFT_BACKWARD)
    DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
       ; DO i=decomp%zst(1),decomp%zen(1)
      temp_c(i,j,k)=what(i,j,k)*kz(k)*scalemodes
    END DO ; END DO ; END DO
    CALL decomp_2d_fft_3d(temp_c,wz,DECOMP_2D_FFT_BACKWARD)

    DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
       DO i=decomp%xst(1),decomp%xen(1)
      utemp(i,j,k)=u(i,j,k)
    END DO ; END DO ; END DO
    DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
       DO i=decomp%xst(1),decomp%xen(1)
      vtemp(i,j,k)=v(i,j,k)
    END DO ; END DO ; END DO
    DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
       DO i=decomp%xst(1),decomp%xen(1)
      wtemp(i,j,k)=w(i,j,k)
    END DO ; END DO ; END DO

    CALL decomp_2d_fft_3d(uhat,u,DECOMP_2D_FFT_BACKWARD)
    CALL decomp_2d_fft_3d(vhat,v,DECOMP_2D_FFT_BACKWARD)
    CALL decomp_2d_fft_3d(what,w,DECOMP_2D_FFT_BACKWARD)

    DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
       DO i=decomp%xst(1),decomp%xen(1)
      u(i,j,k)=u(i,j,k)*scalemodes
    END DO ; END DO ; END DO
    DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
       DO i=decomp%xst(1),decomp%xen(1)
      v(i,j,k)=v(i,j,k)*scalemodes
    END DO ; END DO ; END DO
    DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
       DO i=decomp%xst(1),decomp%xen(1)
      w(i,j,k)=w(i,j,k)*scalemodes
    END DO ; END DO ; END DO

    mychg(1) =maxval(abs(utemp-u))
    mychg(2) =maxval(abs(vtemp-v))
```

```
      mychg(3) =maxval(abs(wtemp-w))
      CALL MPI_ALLREDUCE(mychg,allchg,3,MPI_DOUBLE_PRECISION,MPI_MAX,
         MPI_COMM_WORLD,ierr)
      chg=allchg(1)+allchg(2)+allchg(3)
      IF (myid.eq.0) THEN
        PRINT *,'chg:',chg
      END IF
    END DO
    time(n+1)=n*dt

                !goto 5100
    IF (myid.eq.0) THEN
      PRINT *,'time',n*dt
    END IF

                !save omegax, omegay, and omegaz
    !omegax
    DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO
        i=decomp%xst(1),decomp%xen(1)
      realtemp(i,j,k)=REAL(wy(i,j,k)-vz(i,j,k),KIND=8)
    END DO ; END DO ; END DO
    name_config='./data/omegax'
    CALL savedata(Nx,Ny,Nz,n,name_config,realtemp,decomp)
    !omegay
    DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO
        i=decomp%xst(1),decomp%xen(1)
      realtemp(i,j,k)=REAL(uz(i,j,k)-wx(i,j,k),KIND=8)
    END DO ; END DO ; END DO
    name_config='./data/omegay'
    CALL savedata(Nx,Ny,Nz,n,name_config,realtemp,decomp)
    !omegaz
    DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO
        i=decomp%xst(1),decomp%xen(1)
      realtemp(i,j,k)=REAL(vx(i,j,k)-uy(i,j,k),KIND=8)
    END DO ; END DO ; END DO
    name_config='./data/omegaz'
    CALL savedata(Nx,Ny,Nz,n,name_config,realtemp,decomp)
                !5100 continue
  END DO

        CALL system_clock(finish,count_rate)

        IF (myid.eq.0) then
           PRINT *, 'Program took', REAL(finish-start)/REAL(count_rate), '
              for main timestepping loop'
        END IF

  IF (myid.eq.0) THEN
    name_config = './data/tdata.dat'
    OPEN(unit=11,FILE=name_config,status="UNKNOWN")
    REWIND(11)
    DO n=1,1+Nt
      WRITE(11,*) time(n)
    END DO
    CLOSE(11)

    name_config = './data/xcoord.dat'
    OPEN(unit=11,FILE=name_config,status="UNKNOWN")
    REWIND(11)
    DO i=1,Nx
      WRITE(11,*) x(i)
    END DO
    CLOSE(11)

    name_config = './data/ycoord.dat'
    OPEN(unit=11,FILE=name_config,status="UNKNOWN")
    REWIND(11)
    DO j=1,Ny
      WRITE(11,*) y(j)
    END DO
    CLOSE(11)

    name_config = './data/zcoord.dat'
    OPEN(unit=11,FILE=name_config,status="UNKNOWN")
    REWIND(11)
    DO k=1,Nz
      WRITE(11,*) z(k)
    END DO
    CLOSE(11)
    PRINT *,'Saved data'
  END IF

    ! Calculate error in final numerical solution
  DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
     =decomp%xst(1),decomp%xen(1)
    utemp(i,j,k)=u(i,j,k) -&
            (-0.5*( factor*cos(x(i))*sin(y(j))*sin(z(k))&
            +sin(x(i))*cos(y(j))*cos(z(k)) )*exp(-(factor**2)*time(Nt+1)/
               Re))
  END DO; END DO; END DO
  DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
     =decomp%xst(1),decomp%xen(1)
    vtemp(i,j,k)=v(i,j,k) -&
          (0.5*(  factor*sin(x(i))*cos(y(j))*sin(z(k))&
            -cos(x(i))*sin(y(j))*cos(z(k)) )*exp(-(factor**2)*time(Nt+1)/
               Re))
  END DO ; END DO ; END DO
  DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
     =decomp%xst(1),decomp%xen(1)
    wtemp(i,j,k)=w(i,j,k)-&
          (cos(x(i))*cos(y(j))*sin(z(k))*exp(-(factor**2)*time(Nt+1)/Re))
  END DO ; END DO ; END DO
```

```fortran
mychg(1) = maxval(abs(utemp))
mychg(2) = maxval(abs(vtemp))
mychg(3) = maxval(abs(wtemp))
CALL MPI_ALLREDUCE(mychg,allchg,3,MPI_DOUBLE_PRECISION,MPI_MAX,
    MPI_COMM_WORLD,ierr)
chg=allchg(1)+allchg(2)+allchg(3)
IF (myid.eq.0) THEN
  PRINT*,'The error at the final timestep is',chg
END IF

      ! clean up
  CALL decomp_2d_fft_finalize
  CALL decomp_2d_finalize

DEALLOCATE(x,y,z,time,mychg,allchg,u,v,w,ux,uy,uz,vx,vy,vz,wx,wy,wz,uold
    ,uxold,uyold,uzold,&
        vold,vxold,vyold,vzold,wold,wxold,wyold,wzold,utemp,vtemp,wtemp
            ,&
        temp_r,kx,ky,kz,uhat,vhat,what,rhsuhatfix,rhsvhatfix,&
        rhswhatfix,phat,nonlinuhat,nonlinvhat,nonlinwhat,temp_c,&
        realtemp,stat=AllocateStatus)
IF (AllocateStatus .ne. 0) STOP
IF (myid.eq.0) THEN
  PRINT *,'Program execution complete'
END IF
CALL MPI_FINALIZE(ierr)

END PROGRAM main
```

Listing 13.8: A subroutine to save real array data for the parallel MPI Fortran program to solve the 3D Navier-Stokes equations in listing 13.7.

```fortran
SUBROUTINE savedata(Nx,Ny,Nz,plotnum,name_config,field,decomp)
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This subroutine saves a three dimensional real array in binary
! format
!
! INPUT
!
! .. Scalars ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  Nz        = number of modes in z - power of 2 for FFT
!  plotnum       = number of plot to be made
! .. Arrays ..
!  field      = real data to be saved
!  name_config    = root of filename to save to
!
! .. Output ..
! plotnum      = number of plot to be saved
! .. Special Structures ..
!  decomp      = contains information on domain decomposition
!          see http://www.2decomp.org/ for more information
! LOCAL VARIABLES
!
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!  k         = loop counter in z direction
!  count       = counter
!  iol         = size of file
! .. Arrays ..
!   number_file   = array to hold the number of the plot
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
! 2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
!     (http://www.2decomp.org/index.html)
! MPI library
USE decomp_2d
USE decomp_2d_fft
USE decomp_2d_io
IMPLICIT NONE
INCLUDE 'mpif.h'
! Declare variables
INTEGER(KIND=4), INTENT(IN)           :: Nx,Ny,Nz
INTEGER(KIND=4), INTENT(IN)           :: plotnum
TYPE(DECOMP_INFO), INTENT(IN)         ::  decomp
REAL(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
              decomp%xst(2):decomp%xen(2),&
              decomp%xst(3):decomp%xen(3)), &
                    INTENT(IN) :: field
CHARACTER*100, INTENT(IN)            :: name_config
INTEGER(kind=4)                  :: i,j,k,iol,count,ind
CHARACTER*100                   :: number_file

```

```
  ! create character array with full filename
  ind = index(name_config,' ') - 1
  WRITE(number_file,'(i0)') 10000000+plotnum
  number_file = name_config(1:ind)//number_file
  ind = index(number_file,' ') - 1
  number_file = number_file(1:ind)//'.datbin'
  CALL decomp_2d_write_one(1,field,number_file)

  END SUBROUTINE savedata
```

Listing 13.9: A makefile to compile the parallel MPI Fortran program to solve the 3D Navier-Stokes equations.

```
COMPILER =  mpif90
decompdir= ../2decomp_fft
FLAGS = -O0

DECOMPLIB = -I${decompdir}/include -L${decompdir}/lib -l2decomp_fft
LIBS = #-L${FFTW_LINK}  -lfftw3 -lm
SOURCES = NavierStokes3DfftIMR.f90 savedata.f90

ns3d: $(SOURCES)
    ${COMPILER} -o ns3d $(FLAGS) $(SOURCES) $(LIBS) $(DECOMPLIB)

clean:
  rm -f *.o
  rm -f *.mod
clobber:
  rm -f ns3d
```

13.6.1 Exercises

1) Use 2DECOMP&FFT to write a two dimensional Navier-Stokes solver. The library is built to do three dimensional FFTs, however by choosing one of the arrays to have only one entry, the library can then do two dimensional FFTs on a distributed memory machine.

2) Uecker [59] describes the expected power law scaling for the power spectrum of the enstrophy[4] in two dimensional isotropic turbulence. Look up Uecker [59] and then try to produce numerical data which verifies the power scaling law over as many decades of wavenumber space as are feasible on the computational resources you have access to. A recent overview of research work in this area can be found in Boffetta and Ecke [5]. Fornberg [18] discusses how to calculate power spectra.

[4]The enstrophy is the square of the vorticity.

3) If we set $\mu = 0$ the Navier Stokes equations become the Euler equations. Try to use the implicit midpoint rule and/or the Crank-Nicolson methods to simulate the Euler equations in either two or three dimensions. See if you can find good iterative schemes to do this, you may need to use Newton iteration. An introduction to the Euler equations is in Majda and Bertozzi [42].

4) The Taylor-Green vortex flow initial conditions have been studied as a possible flow that could have a blow up in the maximum value of the absolute value of the gradient of the velocity at a point for the Euler and Navier-Stokes equations. In many of these simulations, symmetries have been used to get higher effective resolutions, see for example Cichowlas and Brachet [10]. Consider using the Kida-Pelz and/or Taylor-Green vortex as initial conditions for the Euler equations and adding non-symmetric perturbations. If you are unable to get an implicit time-stepping scheme to work, consider using an explicit scheme such as a Runge-Kutta method. How does the flow evolve in comparison to previous studies in the literature? An introduction to blow up for the Euler equations is in Majda and Bertozzi [42].

5) The three dimensional program we have written is not the most efficient since one can use a real to complex transform to halve the work done. Implement a real to complex transform in one of the Navier-Stokes programs.

6) The programs we have written can also introduce some aliasing errors. By reading a book on spectral methods, such as Canuto et al. [9], find out what aliasing errors are. Explain why the strategy explained in Johnstone [30] can reduce aliasing errors.

Chapter 14

The Klein-Gordon Equation

14.1 Background

[1]The focusing/defocusing nonlinear Klein-Gordon equation describes the evolution of a possible complex scalar field u according to,

$$\frac{\partial^2 u}{\partial t^2} - \Delta u + u = \pm |u|^2 u, \tag{14.1}$$

where $+$ is the focusing case and $-$ the defocusing case in a similar manner to the nonlinear Schrödinger equation. Blow up of three dimensional radially symmetric real solutions to this equation have recently been numerically studied by Donninger and Schlag [14]. Two dimensional simulations of the Klein-Gordon equation can be found in Yang [62]. The linear Klein-Gordon equation occurs as a modification of the linear Schrödinger equation that is consistent with special relativity, see for example Landau [36] or Grenier [21]. At the present time, there have been no numerical studies of blow up of solutions to this equation without the assumption of radial symmetry. This equation has generated a large mathematical literature and is still poorly understood. Most of this mathematical literature has concentrated on analyzing the equation on an infinite three dimensional space with initial data that either decays exponentially as one tends to infinity or is nonzero on a finite set of the domain. Here, we will simulate this equation in a periodic setting. Since this equation is a wave equation, it has a finite speed of propagation of information, much as a sound wave in air takes time to move from one point to another. Consequently for short time simulations, a simulation of a solution that is only nonzero on a finite part of the domain is similar to a simulation on an infinite domain. However, over long times, the solution can spread out and interact with itself on a periodic domain, whereas on an infinite domain, the interaction over long times is significantly reduced and the solution primarily spreads out. Understanding the interactions in a periodic setting is an interesting mathematical problem. The Klein-Gordon equation

[1]An incomplete but easily accessible mathematical introduction to this equation can be found at `http://wiki.math.toronto.edu/DispersiveWiki/index.php/Semilinear_NLW`.

has a conserved energy given by

$$\int \frac{1}{2}\left(\frac{\partial u}{\partial t}\right)^2 + \frac{u^2}{2} + \frac{1}{2}|\nabla u|^2 \mp \frac{|u|^4}{4}\mathrm{d}\boldsymbol{x}. \tag{14.2}$$

The equation is also time reversible. For long time simulations, one wants to construct numerical methods that approximately conserve this energy and are also time reversible. When using Fourier spectral methods, we primarily need to ensure that the time discretization preserves these properties, since the spectral spatial discretization will typically automatically satisfy these properties. Following Donninger and Schlag [14], we use two schemes. First, an implicit-explicit time stepping scheme which is time reversible but only conserves the energy approximately and is given by

$$\frac{u^{n+1}-2u^n+u^{n-1}}{(\delta t)^2} - \Delta\frac{u^{n+1}+2u^n+u^{n-1}}{4} + \frac{u^{n+1}+2u^n+u^{n-1}}{4} = \pm|u^n|^2 u^n \tag{14.3}$$

and second, a fully implicit time stepping scheme with fixed point iteration

$$\begin{aligned}&\frac{u^{n+1,k+1}-2u^n+u^{n-1}}{(\delta t)^2} - \Delta\frac{u^{n+1,k+1}+2u^n+u^{n-1}}{4} + \frac{u^{n+1,k+1}+2u^n+u^{n-1}}{4}\\ &= \pm\frac{\left|u^{n+1,k}\right|^4-|u^{n-1}|^4}{u^{n+1,k}-u^{n-1}}\end{aligned} \tag{14.4}$$

which conserves a discrete energy exactly

$$\int \frac{1}{2}\left(\frac{u^{n+1}-u^n}{\delta t}\right)^2 + \frac{1}{2}\left(\frac{u^{n+1}+u^n}{2}\right)^2 + \frac{1}{2}\left|\nabla\frac{u^{n+1}+u^n}{2}\right|^2 \mp \frac{|u^{n+1}|^4+|u^n|^4}{8}. \tag{14.5}$$

As before, the superscript n denotes the time step and k denotes the iterate in the fixed point iteration scheme. Iterations are stopped once the difference between two successive iterates falls below a certain tolerance.

14.1.1 Matlab Programs

Listings 14.1, 14.2, 14.3 and 14.4 demonstrate Matlab implementations of these time stepping schemes. In one dimension, the Klein-Gordon equation has easily computable exact solutions, (see for example Nakanishi and Schlag [45, p.6]) which can be used to test the accuracy of the numerical schemes. These equations seem to display three possibilities for the behavior of solutions which are dependent on the initial conditions:

- the solutions could *disperse* or *thermalize*, that is a given localized initial condition spreads out over the entire space
- the solutions blow up or become infinite

- a portion of the solution travels around as a localized particle while the rest of the solution disperses.

Since the equations are reversible, there is also the possibility that a solution which is initially distributed over the spatial domain localizes itself.

Listing 14.1: A Matlab program to solve the 1-dimensional Klein Gordon equation (14.1) using the time discretization in eq. (14.3).

```
% A program to solve the 1D cubic Klein Gordon equation using a
% second order semi-explicit method
% u_{tt}-u_{xx}+u=u^3
clear all; format compact; format short;
set(0,'defaultaxesfontsize',30,'defaultaxeslinewidth',.7,...
    'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,...
    'defaultaxesfontweight','bold')

% set up grid
tic
Lx = 64;            % period  2*pi*L
Nx = 4096;          % number of harmonics
Nt = 500;           % number of time slices
plotgap=10;         % time steps to take between plots
c=0.5;              % wave speed
dt = 5.00/Nt;       % time step

Es = 1.0;           % focusing (+1) or defocusing (-1) parameter
t=0; tdata(1)=t;

% initialise variables
x = (2*pi/Nx)*(-Nx/2:Nx/2 -1)'*Lx;          % x coordinate
kx = 1i*[0:Nx/2-1 0 -Nx/2+1:-1]'/Lx;        % wave vector

% initial conditions
u = sqrt(2)*sech((x-c*t)/sqrt(1-c^2));
uexact= sqrt(2)*sech((x-c*t)/sqrt(1-c^2));
uold=sqrt(2)*sech((x+c*dt)/sqrt(1-c^2));
v=fft(u,[],1);
vold=fft(uold,[],1);
figure(1); clf;
% Plot data on
plot(x,u,'r+',x,uexact,'b-'); legend('numerical','exact');
title(num2str(t));  xlabel x; ylabel u;  drawnow;


% initial energy
vx=0.5*kx.*(v+vold);
ux=ifft(vx,[],1);
Kineticenergy=0.5*abs( (u-uold)/dt).^2;
Strainenergy=0.5*abs(ux).^2;
```

```
Potentialenergy=0.5*abs(0.5*(u+uold)).^2 ...
                    -Es*0.25*((u+uold)*0.5).^4;
Kineticenergy=fft(Kineticenergy,[],1);
Potentialenergy=fft(Potentialenergy,[],1);
Strainenergy=fft(Strainenergy,[],1);
EnKin(1)=Kineticenergy(1);
EnPot(1)=Potentialenergy(1);
EnStr(1)=Strainenergy(1);
En(1)=EnStr(1)+EnKin(1)+EnPot(1);
En0=En(1)

plotnum=1;
% solve pde and plot results

for n =1:Nt+1
    nonlin=u.^3;
    nonlinhat=fft(nonlin,[],1);
    vnew=(0.25*(kx.*kx -1).*(2*v+vold)...
                +(2*v-vold)/(dt*dt) +Es*nonlinhat)./...
            (1/(dt*dt) - (kx.*kx-1)*0.25 );
    unew=ifft(vnew,[],1);
    t=n*dt;
    if (mod(n,plotgap)==0)
        uexact=sqrt(2)*sech((x-c*t)/sqrt(1-c^2));
        figure(1); clf;
        plot(x,u,'r+',x,uexact,'b-'); legend('numerical','exact');
        title(num2str(t)); xlim([-6,6]); xlabel x; ylabel u;  drawnow;
        tdata(plotnum+1)=t;
        vx=0.5*kx.*(v+vold);
        ux=ifft(vx,[],1);
        Kineticenergy=0.5*abs( (u-uold)/dt).^2;
        Strainenergy=0.5*abs(ux).^2;
        Potentialenergy=0.5*abs(0.5*(u+uold)).^2 ...
                    -Es*0.25*((u+uold)*0.5).^4;
        Kineticenergy=fft(Kineticenergy,[],1);
        Potentialenergy=fft(Potentialenergy,[],1);
        Strainenergy=fft(Strainenergy,[],1);
        EnKin(plotnum+1)=Kineticenergy(1);
        EnPot(plotnum+1)=Potentialenergy(1);
        EnStr(plotnum+1)=Strainenergy(1);
        En(plotnum+1)=EnStr(plotnum+1)+EnKin(plotnum+1)+EnPot(plotnum+1);
        Enchange(plotnum)=log(abs(1-En(1+plotnum)/En0));
        plotnum=plotnum+1;
    end
    % update old terms
    vold=v;
    v=vnew;
    uold=u;
    u=unew;
end
figure(4); clf;
```

```
uexact=sqrt(2)*sech((x-c*t)/sqrt(1-c^2));
plot(x,u,'r+',x,uexact,'b-'); legend('numerical','exact');
title(num2str(t)); xlabel x; ylabel u;  drawnow;
max(abs(u-uexact))
figure(5); clf; plot(tdata,En,'r-',tdata,EnKin,'b:',tdata,EnPot,'g-.',
    tdata,EnStr,'y--');
xlabel time; ylabel Energy; legend('Total','Kinetic','Potential','Strain')
    ;
figure(6); clf; plot(tdata(2:end),Enchange,'r-'); xlabel time; ylabel('
    Energy change');

toc
```

Listing 14.2: A Matlab program to solve the 1-dimensional Klein Gordon equation (14.1) using the time discretization in eq. (14.4).

```
% A program to solve the 1D cubic Klein Gordon equation using a
% second order implicit method
% u_{tt}-u_{xx}+u=u^3
clear all; format compact; format short;
set(0,'defaultaxesfontsize',30,'defaultaxeslinewidth',.7,...
    'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,...
    'defaultaxesfontweight','bold')

% set up grid
tic
Lx = 64;          % period  2*pi*L
Nx = 4096;         % number of harmonics
Nt = 400;        % number of time slices
plotgap=10;      % timesteps between plots
tol=0.1^(15);    % tolerance for fixed point iterations
dt = 0.500/Nt;     % time step
c=0.5;               % wave speed

Es = 1.0;  % focusing (+1) or defocusing (-1) parameter
t=0; tdata(1)=t;

% initialise variables
x = (2*pi/Nx)*(-Nx/2:Nx/2 -1)'*Lx;           % x coordinate
kx = 1i*[0:Nx/2-1 0 -Nx/2+1:-1]'/Lx;         % wave vector

% initial conditions
u = sqrt(2)*sech((x-c*t)/sqrt(1-c^2));
uexact= sqrt(2)*sech((x-c*t)/sqrt(1-c^2));
uold=sqrt(2)*sech((x+c*dt)/sqrt(1-c^2));
v=fft(u,[],1);
vold=fft(uold,[],1);
figure(1); clf;
% Plot data on
plot(x,u,'r+',x,uexact,'b-'); legend('numerical','exact');
```

```
title(num2str(0)); xlim([-6,6]); xlabel x; ylabel u;  drawnow;


% initial energy
vx=0.5*kx.*(v+vold);
ux=ifft(vx,[],1);
Kineticenergy=0.5*abs( (u-uold)/dt).^2;
Strainenergy=0.5*abs(ux).^2;
Potentialenergy=0.5*abs(0.5*(u+uold)).^2 ...
                    -Es*0.25*((u+uold)*0.5).^4;
Kineticenergy=fft(Kineticenergy,[],1);
Potentialenergy=fft(Potentialenergy,[],1);
Strainenergy=fft(Strainenergy,[],1);
EnKin(1)=Kineticenergy(1);
EnPot(1)=Potentialenergy(1);
EnStr(1)=Strainenergy(1);
En(1)=EnStr(1)+EnKin(1)+EnPot(1);
En0=En(1)

plotnum=1;
% solve pde and plot results

for n =1:Nt+1
   nonlin=(u.^2 +uold.^2).*(u+uold)/4;
    nonlinhat=fft(nonlin,[],1);
    chg=1;
    unew=u;
    while (chg>tol)
        utemp=unew;
        vnew=(0.25*(kx.*kx -1).*(2*v+vold)...
                +(2*v-vold)/(dt*dt) +Es*nonlinhat)./...
            (1/(dt*dt) - (kx.*kx -1)*0.25 );
        unew=ifft(vnew,[],1);
        nonlin=(unew.^2 +uold.^2).*(unew+uold)/4;
        nonlinhat=fft(nonlin,[],1);
        chg=max(abs(unew-utemp));
    end
    t=n*dt;
    if (mod(n,plotgap)==0)
        uexact=sqrt(2)*sech((x-c*t)/sqrt(1-c^2));
        figure(1); clf;
        plot(x,u,'r+',x,uexact,'b-'); legend('numerical','exact');
        title(num2str(t)); xlim([-6,6]); xlabel x; ylabel u;  drawnow;
        tdata(plotnum+1)=t;
        vx=0.5*kx.*(v+vold);
        ux=ifft(vx,[],1);
        Kineticenergy=0.5*abs( (u-uold)/dt).^2;
        Strainenergy=0.5*abs(ux).^2;
        Potentialenergy=0.5*abs(0.5*(u+uold)).^2 ...
                    -Es*0.25*((u+uold)*0.5).^4;
        Kineticenergy=fft(Kineticenergy,[],1);
        Potentialenergy=fft(Potentialenergy,[],1);
        Strainenergy=fft(Strainenergy,[],1);
        EnKin(plotnum+1)=Kineticenergy(1);
        EnPot(plotnum+1)=Potentialenergy(1);
        EnStr(plotnum+1)=Strainenergy(1);
        En(plotnum+1)=EnStr(plotnum+1)+EnKin(plotnum+1)+EnPot(plotnum+1);
        Enchange(plotnum)=log(abs(1-En(1+plotnum)/En0));
        plotnum=plotnum+1;
    end
    % update old terms
    vold=v;
    v=vnew;
    uold=u;
    u=unew;
end
figure(4); clf;
uexact=sqrt(2)*sech((x-c*t)/sqrt(1-c^2));
plot(x,u,'r+',x,uexact,'b-'); legend('numerical','exact');
title(num2str(t)); xlim([-6,6]); xlabel x; ylabel u;  drawnow;
max(abs(u-uexact))
figure(5); clf; plot(tdata,En,'r-',tdata,EnKin,'b:',tdata,EnPot,'g-.',
    tdata,EnStr,'y--');
xlabel time; ylabel Energy; legend('Total','Kinetic','Potential','Strain')
    ;
figure(6); clf; plot(tdata(2:end),Enchange,'r-'); xlabel time; ylabel('
    Energy change');

toc
```

Listing 14.3: A Matlab program to solve the 2-dimensional Klein Gordon equation (14.1) using the time discretization in eq. (14.4).

```
% A program to solve the 2D Klein Gordon equation using a
% second order implicit method

clear all; format compact; format short;
set(0,'defaultaxesfontsize',30,'defaultaxeslinewidth',.7,...
    'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,...
    'defaultaxesfontweight','bold')

% set up grid
tic
Lx = 3;          % period  2*pi*L
Ly = 3;          % period  2*pi*L
Nx = 2*256;      % number of harmonics
Ny = 2*256;      % number of harmonics
Nt = 2000;         % number of time slices
dt = 50.0/Nt;   % time step
tol=0.1^(10);    % tolerance for fixed point iterations
plotgap=10;      % timesteps between plots
```

```

Es = 1.0; % focusing (+1) or defocusing (-1) parameter


% initialise variables
x = (2*pi/Nx)*(-Nx/2:Nx/2 -1)'*Lx;           % x coordinate
kx = 1i*[0:Nx/2-1 0 -Nx/2+1:-1]'/Lx;         % wave vector
y = (2*pi/Ny)*(-Ny/2:Ny/2 -1)'*Ly;           % y coordinate
ky = 1i*[0:Ny/2-1 0 -Ny/2+1:-1]'/Ly;         % wave vector
[xx,yy]=meshgrid(x,y);
[kxm,kym]=meshgrid(kx,ky);

% initial conditions
u = (0.5*exp(-(xx.^2+yy.^2))).*sin(10*xx+12*yy);
uold=u;
v=fft2(u);
vold=fft2(uold);
figure(1); clf; mesh(xx,yy,u); drawnow;
t=0; tdata(1)=t;

% initial energy
vx=0.5*kxm.*(v+vold);
vy=0.5*kym.*(v+vold);
ux=ifft2(vx);
uy=ifft2(vy);
ux=ifft2(vx);
uy=ifft2(vy);
Kineticenergy=0.5*abs( (u-uold)/dt).^2;
Strainenergy=0.5*abs(ux).^2 +0.5*abs(uy).^2;
Potentialenergy=0.5*abs(0.5*(u+uold)).^2 ...
                    -Es*0.25*((u+uold)*0.5).^4;
Kineticenergy=fft2(Kineticenergy);
Potentialenergy=fft2(Potentialenergy);
Strainenergy=fft2(Strainenergy);
EnKin(1)=Kineticenergy(1,1);
EnPot(1)=Potentialenergy(1,1);
EnStr(1)=Strainenergy(1,1);
En(1)=EnStr(1)+EnKin(1)+EnPot(1);
En0=En(1)
plotnum=1;

% solve pde and plot results

for n =1:Nt+1
    nonlin=(u.^4 -uold.^4)./(u-uold+0.1^14);
    nonlinhat=fft2(nonlin);
    chg=1;
    unew=u;
    while (chg>tol)
        utemp=unew;
        vnew=(0.25*(kxm.^2 + kym.^2 -1).*(2*v+vold)...
             +(2*v-vold)/(dt*dt) +Es*nonlinhat)./...
            (1/(dt*dt) - (kxm.^2 +  kym.^2-1)*0.25 );
        unew=ifft2(vnew);
        nonlin=(unew.^4 -uold.^4)./(unew-uold+0.1^14);
        nonlinhat=fft2(nonlin);
        chg=max(abs(unew-utemp));
    end
    t=n*dt;
    if (mod(n,plotgap)==0)
        figure(1); clf; mesh(xx,yy,abs(u).^2);
        t
        tdata(plotnum+1)=t;
        vx=0.5*kxm.*(v+vold);
        vy=0.5*kym.*(v+vold);
        ux=ifft2(vx);
        uy=ifft2(vy);
        Kineticenergy=0.5*abs( (unew-u)/dt).^2;
        Strainenergy=0.5*abs(ux).^2 +0.5*abs(uy).^2;
        Potentialenergy=0.5*abs(0.5*(unew+u)).^2 ...
                    -Es*0.25*((unew+u)*0.5).^4;
        Kineticenergy=fft2(Kineticenergy);
        Potentialenergy=fft2(Potentialenergy);
        Strainenergy=fft2(Strainenergy);
        EnKin(1+plotnum)=Kineticenergy(1,1);
        EnPot(1+plotnum)=Potentialenergy(1,1);
        EnStr(1+plotnum)=Strainenergy(1,1);
        En(1+plotnum)=EnStr(1+plotnum)+EnKin(1+plotnum)+EnPot(1+plotnum);

        Enchange(plotnum)=log(abs(1-En(1+plotnum)/En0));
        plotnum=plotnum+1;
    end
    % update old terms
    vold=v;
    v=vnew;
    uold=u;
    u=unew;
end
figure(5); clf; plot(tdata,En,'r-',tdata,EnKin,'b:',tdata,EnPot,'g-.',
    tdata,EnStr,'y--');
xlabel time; ylabel Energy; legend('Total','Kinetic','Potential','Strain')
    ;
figure(6); clf; plot(tdata(2:end),Enchange,'r-'); xlabel time; ylabel('
    Energy change');

figure(4); clf; mesh(xx,yy,abs(u).^2);
toc
```

Listing 14.4: A Matlab program to solve the 3-dimensional Klein Gordon equation (14.1) using the time discretization in eq. (14.3).

```
% A program to solve the 3D Klein Gordon equation using a
% second order semi-explicit method

clear all; format compact; format short;
set(0,'defaultaxesfontsize',30,'defaultaxeslinewidth',.7,...
    'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,...
    'defaultaxesfontweight','bold')

% set up grid
tic
Lx = 2;          % period  2*pi*L
Ly = 2;          % period  2*pi*L
Lz = 2;          % period  2*pi*L
Nx = 64;         % number of harmonics
Ny = 64;         % number of harmonics
Nz = 64;         % number of harmonics
Nt = 2000;        % number of time slices
plotgap=10;
dt = 10.0/Nt;     % time step

Es = -1.0;  % focusing (+1) or defocusing (-1) parameter

% initialise variables
x = (2*pi/Nx)*(-Nx/2:Nx/2 -1)'*Lx;           % x coordinate
kx = 1i*[0:Nx/2-1 0 -Nx/2+1:-1]'/Lx;         % wave vector
y = (2*pi/Ny)*(-Ny/2:Ny/2 -1)'*Ly;           % y coordinate
ky = 1i*[0:Ny/2-1 0 -Ny/2+1:-1]'/Ly;         % wave vector
z = (2*pi/Nz)*(-Nz/2:Nz/2 -1)'*Lz;           % y coordinate
kz = 1i*[0:Nz/2-1 0 -Nz/2+1:-1]'/Lz;         % wave vector
[xx,yy,zz]=meshgrid(x,y,z);
[kxm,kym,kzm]=meshgrid(kx,ky,kz);

% initial conditions
u = 0.1*exp(-(xx.^2+(yy).^2+zz.^2));
uold=u;
v=fftn(u);
vold=v;
figure(1); clf;
% coordinate slice to show plots on
sx=[0]; sy=[0]; sz=[-Lx*2*pi];
slice(xx,yy,zz,u,sx,sy,sz); colormap jet;
title(num2str(0)); colorbar('location','EastOutside'); drawnow;

xlabel('x'); ylabel('y'); zlabel('z');
axis equal; axis square;  view(3); drawnow;
t=0; tdata(1)=t;

% initial energy
vx=0.5*kxm.*(v+vold);
vy=0.5*kym.*(v+vold);
vz=0.5*kzm.*(v+vold);
```

```
ux=ifftn(vx);
uy=ifftn(vy);
uz=ifftn(vz);
Kineticenergy=0.5*abs( (u-uold)/dt).^2;
Strainenergy=0.5*abs(ux).^2 +0.5*abs(uy).^2+0.5*abs(uz).^2;
Potentialenergy=0.5*abs(0.5*(u+uold)).^2 ...
                    -Es*0.25*((u+uold)*0.5).^4;
Kineticenergy=fftn(Kineticenergy);
Potentialenergy=fftn(Potentialenergy);
Strainenergy=fftn(Strainenergy);
EnKin(1)=Kineticenergy(1,1);
EnPot(1)=Potentialenergy(1,1);
EnStr(1)=Strainenergy(1,1);
En(1)=EnStr(1)+EnKin(1)+EnPot(1);
En0=En(1)

plotnum=1;
% solve pde and plot results

for n =1:Nt+1
    nonlin=u.^3;
    nonlinhat=fftn(nonlin);
    vnew=(0.25*(kxm.^2 + kym.^2 + kzm.^2 -1).*(2*v+vold)...
                +(2*v-vold)/(dt*dt) +Es*nonlinhat)./...
            (1/(dt*dt) - (kxm.^2 + kzm.^2 + kym.^2 - 1)*0.25 );
    unew=ifftn(vnew);
    t=n*dt;
    if (mod(n,plotgap)==0)
        figure(1); clf; sx=[0]; sy=[0]; sz=[0];
        slice(xx,yy,zz,u,sx,sy,sz); colormap jet;
        title(num2str(t)); colorbar('location','EastOutside'); drawnow;
        xlabel('x'); ylabel('y'); zlabel('z');
        axis equal; axis square;  view(3); drawnow;
        tdata(plotnum+1)=t;
        t
        vx=0.5*kxm.*(v+vold);
        vy=0.5*kym.*(v+vold);
        vz=0.5*kzm.*(v+vold);
        ux=ifftn(vx);
        uy=ifftn(vy);
        uz=ifftn(vz);
        Kineticenergy=0.5*abs( (u-uold)/dt).^2;
        Strainenergy=0.5*abs(ux).^2 +0.5*abs(uy).^2+0.5*abs(uz).^2;
        Potentialenergy=0.5*abs(0.5*(u+uold)).^2 ...
                    -Es*0.25*((u+uold)*0.5).^4;
        Kineticenergy=fftn(Kineticenergy);
        Potentialenergy=fftn(Potentialenergy);
        Strainenergy=fftn(Strainenergy);
        EnKin(plotnum+1)=Kineticenergy(1,1,1);
        EnPot(plotnum+1)=Potentialenergy(1,1,1);
        EnStr(plotnum+1)=Strainenergy(1,1,1);
        En(plotnum+1)=EnStr(plotnum+1)+EnKin(plotnum+1)+EnPot(plotnum+1);
        Enchange(plotnum)=log(abs(1-En(1+plotnum)/En0));
        plotnum=plotnum+1;
    end
    % update old terms
    vold=v;
    v=vnew;
    uold=u;
    u=unew;
end
figure(4); clf;
% coordinate slice to show plots on
sx=[0]; sy=[0]; sz=[0];
slice(xx,yy,zz,u,sx,sy,sz); colormap jet;
title(num2str(t)); colorbar('location','EastOutside'); drawnow;

xlabel('x'); ylabel('y'); zlabel('z');
axis equal; axis square;  view(3); drawnow;

figure(5); clf; plot(tdata,En,'r-',tdata,EnKin,'b:',tdata,EnPot,'g-.',
    tdata,EnStr,'y--');
xlabel time; ylabel Energy; legend('Total','Kinetic','Potential','Strain')
    ;
figure(6); clf; plot(tdata(2:end),Enchange,'r-'); xlabel time; ylabel('
    Energy change');

toc
```

14.1.2 A Two-Dimensional OpenMP Fortran Program

The programs that we have developed in Fortran have become rather long. Here we add subroutines to make the programs shorter and easier to maintain. Listing 14.5 is the main Fortran program which uses OpenMP to solve the 2D Klein-Gordon equation. Notice that by using subroutines, we have made the main program significantly shorter and easier to read. It is still not as simple to read as the Matlab program, but is significantly better than some of the previous Fortran programs. It is also much easier to maintain, and once the subroutines have been written and debugged, they may be reused in other programs. The only drawback in using too many subroutines is that one may encounter a slight decrease in performance due to the overhead of calling a subroutine and passing data to it. The subroutines are in listings 14.6, 14.7, 14.8, 14.9, 14.10, 14.11 and an example makefile is in listing 14.12. Finally listing 14.13 contains a Matlab program which produces pictures from the binary files that have been computed. One can then use another program to take the images and create a video[2].

[2]At the present time, Matlab's video commands cannot reliably produce a single video from a very long simulation, so it is better to use Matlab to create still images.

Listing 14.5: A Fortran program to solve the 2D Klein-Gordon equation.

```
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This program solves nonlinear Klein-Gordon equation in 2 dimensions
! u_{tt}-u_{xx}+u_{yy}+u=Es*|u|^2u
! using a second order implicit-explicit time stepping scheme.
!
! The boundary conditions are u(x=0,y)=u(2*Lx*\pi,y),
! u(x,y=0)=u(x,y=2*Ly*\pi)
! The initial condition is u=0.5*exp(-x^2-y^2)*sin(10*x+12*y)
!
! .. Parameters ..
!  Nx       = number of modes in x - power of 2 for FFT
!  Ny       = number of modes in y - power of 2 for FFT
!  Nt       = number of timesteps to take
!  Tmax       = maximum simulation time
!  plotgap      = number of timesteps between plots
!  FFTW_IN_PLACE  = value for FFTW input
!  FFTW_MEASURE   = value for FFTW input
!  FFTW_EXHAUSTIVE  = value for FFTW input
!  FFTW_PATIENT   = value for FFTW input
!  FFTW_ESTIMATE  = value for FFTW input
!  FFTW_FORWARD     = value for FFTW input
!  FFTW_BACKWARD  = value for FFTW input
!  pi = 3.14159265358979323846264338327950288419716939937510d0
!  Lx       = width of box in x direction
!  Ly       = width of box in y direction
!  ES       = +1 for focusing and -1 for defocusing
! .. Scalars ..
!  i        = loop counter in x direction
!  j        = loop counter in y direction
!  n        = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  start      = variable to record start time of program
!  finish     = variable to record end time of program
!  count_rate   = variable for clock count rate
!  planfxy      = Forward 2d fft plan
!  planbxy      = Backward 2d fft plan
!  dt       = timestep
!  ierr       = error code
!  plotnum      = number of plot
! .. Arrays ..
!  unew       = approximate solution
!  vnew       = Fourier transform of approximate solution
!  u      = approximate solution
!  v      = Fourier transform of approximate solution
!  uold       = approximate solution
!  vold       = Fourier transform of approximate solution
!  nonlin       = nonlinear term, u^3
!  nonlinhat    = Fourier transform of nonlinear term, u^3
! .. Vectors ..
!  kx       = fourier frequencies in x direction
!  ky       = fourier frequencies in y direction
!  x        = x locations
!  y        = y locations
!  time       = times at which save data
!  en       = total energy
!  enstr      = strain energy
!  enpot      = potential energy
!  enkin      = kinetic energy
!  name_config    = array to store filename for data to be saved
!  fftfxy     = array to setup 2D Fourier transform
!  fftbxy     = array to setup 2D Fourier transform
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!---------------------------------------------------------------------
! External routines required
! getgrid.f90 -- Get initial grid of points
!   initialdata.f90 -- Get initial data
! enercalc.f90 -- Subroutine to calculate the energy
! savedata.f90 -- Save initial data
! storeold.f90 -- Store old data
! External libraries required
!   FFTW3  -- Fast Fourier Transform in the West Library
!     (http://www.fftw.org/)
!   OpenMP library

PROGRAM Kg
USE omp_lib
! Declare variables
IMPLICIT NONE
INTEGER(kind=4), PARAMETER  :: Nx=128
INTEGER(kind=4), PARAMETER  :: Ny=128
INTEGER(kind=4), PARAMETER  :: Nt=20
INTEGER(kind=4), PARAMETER  :: plotgap=5
REAL(kind=8), PARAMETER   :: &
  pi=3.14159265358979323846264338327950288419716939937510d0
REAL(kind=8), PARAMETER   ::  Lx=3.0d0
REAL(kind=8), PARAMETER   ::  Ly=3.0d0
REAL(kind=8), PARAMETER   ::  Es=1.0d0
REAL(kind=8)         ::  dt=0.10d0/REAL(Nt,kind(0d0))
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  kx,ky
REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  x,y
COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  u,nonlin
COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  v,nonlinhat
COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  uold
COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  vold
COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  unew
COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  vnew
REAL(kind=8), DIMENSION(:,:), ALLOCATABLE :: savearray
REAL(kind=8), DIMENSION(:), ALLOCATABLE ::  time,enkin,enstr,enpot,en
INTEGER(kind=4) ::  ierr,i,j,n,allocatestatus,numthreads
INTEGER(kind=4) ::  start, finish, count_rate, plotnum
INTEGER(kind=4), PARAMETER  ::  FFTW_IN_PLACE = 8, FFTW_MEASURE = 0, &
  FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32, FFTW_ESTIMATE = 64
INTEGER(kind=4),PARAMETER ::  FFTW_FORWARD = -1, FFTW_BACKWARD=1
INTEGER(kind=8) ::  planfxy,planbxy
CHARACTER*100 ::  name_config
! Start short parallel region to count threads
numthreads=omp_get_max_threads()
PRINT *,'There are ',numthreads,' threads.'
ALLOCATE(kx(1:Nx),ky(1:Ny),x(1:Nx),y(1:Ny),u(1:Nx,1:Ny),&
    v(1:Nx,1:Ny),nonlin(1:Nx,1:Ny),nonlinhat(1:Nx,1:Ny),&
    uold(1:Nx,1:Ny),vold(1:Nx,1:Ny),&
    unew(1:Nx,1:Ny),vnew(1:Nx,1:Ny),savearray(1:Nx,1:Ny),&
    time(1:1+Nt/plotgap),enkin(1:1+Nt/plotgap),&
    enstr(1:1+Nt/plotgap),enpot(1:1+Nt/plotgap),&
    en(1:1+Nt/plotgap),stat=allocatestatus)
IF (allocatestatus .ne. 0) stop
PRINT *,'allocated arrays'

! set up multithreaded ffts
CALL dfftw_init_threads_(ierr)
PRINT *,'Initiated threaded FFTW'
CALL dfftw_plan_with_nthreads_(numthreads)
PRINT *,'Inidicated number of threads to be used in planning'
CALL dfftw_plan_dft_2d_(planfxy,Nx,Ny,u,v,&
            FFTW_FORWARD,FFTW_ESTIMATE)
CALL dfftw_plan_dft_2d_(planbxy,Nx,Ny,v,u,&
            FFTW_BACKWARD,FFTW_ESTIMATE)
PRINT *,'Setup FFTs'
! setup fourier frequencies
CALL getgrid(Nx,Ny,Lx,Ly,pi,name_config,x,y,kx,ky)
PRINT *,'Setup grid and fourier frequencies'
CALL initialdata(Nx,Ny,x,y,u,uold)
plotnum=1
name_config = 'data/u'
savearray=REAL(u)
CALL savedata(Nx,Ny,plotnum,name_config,savearray)

CALL dfftw_execute_dft_(planfxy,u,v)
CALL dfftw_execute_dft_(planfxy,uold,vold)

CALL enercalc(Nx,Ny,planfxy,planbxy,dt,Es,&
      enkin(plotnum),enstr(plotnum),&
      enpot(plotnum),en(plotnum),&
      kx,ky,nonlin,nonlinhat,&
      v,vold,u,uold)

PRINT *,'Got initial data, starting timestepping'
time(plotnum)=0.0d0
  CALL system_clock(start,count_rate)
DO n=1,Nt
  !$OMP PARALLEL DO PRIVATE(i,j) SCHEDULE(static)
  DO j=1,Ny
    DO i=1,Nx
      nonlin(i,j)=(abs(u(i,j))*2)*u(i,j)
    END DO
  END DO
  !$OMP END PARALLEL DO
  CALL dfftw_execute_dft_(planfxy,nonlin,nonlinhat)
  !$OMP PARALLEL DO PRIVATE(i,j) SCHEDULE(static)
  DO j=1,Ny
    DO i=1,Nx
      vnew(i,j)=( 0.25*(kx(i)*kx(i) + ky(j)*ky(j)-1.0d0)&
        *(2.0d0*v(i,j)+vold(i,j))&
        +(2.0d0*v(i,j)-vold(i,j))/(dt*dt)&
        +Es*nonlinhat(i,j) )&
        /(1/(dt*dt)-0.25*(kx(i)*kx(i) + ky(j)*ky(j)-1.0d0))
    END DO
  END DO
  !$OMP END PARALLEL DO
  CALL dfftw_execute_dft_(planbxy,vnew,unew)
  ! normalize result
  !$OMP PARALLEL DO PRIVATE(i,j) SCHEDULE(static)
  DO j=1,Ny
    DO i=1,Nx
      unew(i,j)=unew(i,j)/REAL(Nx*Ny,kind(0d0))
    END DO
  END DO
  !$OMP END PARALLEL DO
  IF (mod(n,plotgap)==0) then
    plotnum=plotnum+1
    time(plotnum)=n*dt
    PRINT *,'time',n*dt
    CALL enercalc(Nx,Ny,planfxy,planbxy,dt,Es,&
      enkin(plotnum),enstr(plotnum),&
      enpot(plotnum),en(plotnum),&
      kx,ky,&
      nonlin,nonlinhat,&
      vnew,v,unew,u)
    savearray=REAL(unew,kind(0d0))
    CALL savedata(Nx,Ny,plotnum,name_config,savearray)
  END IF
    ! .. Update old values ..
  CALL storeold(Nx,Ny,&
      unew,u,uold,&
      vnew,v,vold)
END DO
PRINT *,'Finished time stepping'
CALL system_clock(finish,count_rate)
PRINT*,'Program took ',&
  REAL(finish-start,kind(0d0))/REAL(count_rate,kind(0d0)),&
  'for Time stepping'
CALL saveresults(Nt,plotgap,time(1:1+n/plotgap),en(1:1+n/plotgap),&
    enstr(1:1+n/plotgap),enkin(1:1+n/plotgap),enpot(1:1+n/plotgap))

! Save times at which output was made in text format
PRINT *,'Saved data'

CALL dfftw_destroy_plan_(planbxy)
CALL dfftw_destroy_plan_(planfxy)
CALL dfftw_cleanup_threads_()
DEALLOCATE(kx,ky,x,y,u,v,nonlin,nonlinhat,savearray,&
  uold,vold,unew,vnew,time,enkin,enstr,enpot,en,&
  stat=allocatestatus)
IF (allocatestatus .ne. 0) STOP
PRINT *,'Deallocated arrays'
PRINT *,'Program execution complete'
END PROGRAM Kg
```

Listing 14.6: A Fortran subroutine to get the grid to solve the 2D Klein-Gordon equation on.

```
  SUBROUTINE getgrid(Nx,Ny,Lx,Ly,pi,name_config,x,y,kx,ky)
  !--------------------------------------------------------------------
  !
  !
  ! PURPOSE
  !
  ! This subroutine gets grid points and fourier frequencies for a
  ! pseudospectral simulation of the 2D nonlinear Klein-Gordon equation
  !
  ! u_{tt}-u_{xx}+u_{yy}+u=Es*u^3
  !
  ! The boundary conditions are u(x=0,y)=u(2*Lx*\pi,y),
  ! u(x,y=0)=u(x,y=2*Ly*\pi)
  !
  ! INPUT
  !
  ! .. Scalars ..
```

```
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  pi        = 3.142....
!  Lx        = width of box in x direction
!  Ly        = width of box in y direction
! OUTPUT
!
! .. Vectors ..
!  kx        = fourier frequencies in x direction
!  ky        = fourier frequencies in y direction
!  x         = x locations
!  y         = y locations
!
! LOCAL VARIABLES
!
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
! OpenMP library
IMPLICIT NONE
USE omp_lib
! Declare variables
INTEGER(KIND=4), INTENT(IN)              :: Nx,Ny
REAL(kind=8), INTENT(IN)               :: Lx,Ly,pi
REAL(KIND=8), DIMENSION(1:NX), INTENT(OUT)      :: x
REAL(KIND=8), DIMENSION(1:NY), INTENT(OUT)      :: y
COMPLEX(KIND=8), DIMENSION(1:NX), INTENT(OUT)    :: kx
COMPLEX(KIND=8), DIMENSION(1:NY), INTENT(OUT)    :: ky
CHARACTER*100, INTENT(OUT)               :: name_config
INTEGER(kind=4)                      :: i,j

!$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
DO i=1,1+Nx/2
  kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
END DO
```

```
!$OMP END PARALLEL DO
kx(1+Nx/2)=0.0d0
!$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
DO i = 1,Nx/2 -1
  kx(i+1+Nx/2)=-kx(1-i+Nx/2)
END DO
!$OMP END PARALLEL DO

!$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
DO i=1,Nx
  x(i)=(-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)))*pi*Lx
END DO
!$OMP END PARALLEL DO
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,1+Ny/2
  ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
END DO
!$OMP END PARALLEL DO
ky(1+Ny/2)=0.0d0
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j = 1,Ny/2 -1
  ky(j+1+Ny/2)=-ky(1-j+Ny/2)
END DO
!$OMP END PARALLEL DO
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,Ny
  y(j)=(-1.0d0 + 2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)))*pi*Ly
END DO
!$OMP END PARALLEL DO
! Save x grid points in text format
name_config = 'xcoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO i=1,Nx
  WRITE(11,*) x(i)
END DO
CLOSE(11)
! Save y grid points in text format
name_config = 'ycoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,Ny
  WRITE(11,*) y(j)
END DO
CLOSE(11)


END SUBROUTINE getgrid
```

Listing 14.7: A Fortran subroutine to get the initial data to solve the 2D Klein-Gordon equation for.

```fortran
SUBROUTINE initialdata(Nx,Ny,x,y,u,uold)
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This subroutine gets initial data for nonlinear Klein-Gordon equation
! in 2 dimensions
! u_{tt}-u_{xx}+u_{yy}+u=Es*u^3+
!
! The boundary conditions are u(x=-Lx*\pi,y)=u(x=Lx*\pi,y),
! u(x,y=-Ly*\pi)=u(x,y=Ly*\pi)
! The initial condition is u=0.5*exp(-x^2-y^2)*sin(10*x+12*y)
!
! INPUT
!
! .. Parameters ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
! .. Vectors ..
!  x         = x locations
!  y         = y locations
!
! OUTPUT
!
! .. Arrays ..
!  u         = initial solution
!  uold        = approximate solution based on time derivative of
!          initial solution
!
! LOCAL VARIABLES
!
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
```

```
!
! External libraries required
! OpenMP library
USE omp_lib
IMPLICIT NONE
! Declare variables
INTEGER(KIND=4), INTENT(IN)                  :: Nx,Ny
REAL(KIND=8), DIMENSION(1:NX), INTENT(IN)        :: x
REAL(KIND=8), DIMENSION(1:NY), INTENT(IN)        :: y
COMPLEX(KIND=8), DIMENSION(1:NX,1:NY), INTENT(OUT)  :: u,uold
INTEGER(kind=4)                     :: i,j
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,Ny
  u(1:Nx,j)=0.5d0*exp(-1.0d0*(x(1:Nx)**2 +y(j)**2))*&
        sin(10.0d0*x(1:Nx)+12.0d0*y(j))
END DO
!$OMP END PARALLEL DO
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,Ny
  uold(1:Nx,j)=0.5d0*exp(-1.0d0*(x(1:Nx)**2 +y(j)**2))*&
        sin(10.0d0*x(1:Nx)+12.0d0*y(j))
END DO
!$OMP END PARALLEL DO

END SUBROUTINE initialdata
```

Listing 14.8: A Fortran program to save a field from the solution of the 2D Klein-Gordon equation.

```
SUBROUTINE savedata(Nx,Ny,plotnum,name_config,field)
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This subroutine saves a two dimensional real array in binary
! format
!
! INPUT
!
! .. Scalars ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  plotnum       = number of plot to be made
! .. Arrays ..
!  field       = real data to be saved
!  name_config     = root of filename to save to
!
! .. Output ..
! plotnum      = number of plot to be saved
```

```fortran
!
! LOCAL VARIABLES
!
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!  count       = counter
!  iol         = size of file
! .. Arrays ..
!   number_file    = array to hold the number of the plot
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
IMPLICIT NONE
! Declare variables
INTEGER(KIND=4), INTENT(IN)            :: Nx,Ny
INTEGER(KIND=4), INTENT(IN)            :: plotnum
REAL(KIND=8), DIMENSION(1:NX,1:NY), INTENT(IN)  :: field
CHARACTER*100, INTENT(IN)            :: name_config
INTEGER(kind=4)                   :: i,j,iol,count,ind
CHARACTER*100                   :: number_file

! create character array with full filename
ind = index(name_config,' ') - 1
WRITE(number_file,'(i0)') 10000000+plotnum
number_file = name_config(1:ind)//number_file
ind = index(number_file,' ') - 1
number_file = number_file(1:ind)//'.datbin'
INQUIRE( iolength=iol ) field(1,1)
OPEN(unit=11,FILE=number_file,form="unformatted",&
  access="direct",recl=iol)
count=1
DO j=1,Ny
  DO i=1,Nx
    WRITE(11,rec=count) field(i,j)
    count=count+1
  END DO
END DO
CLOSE(11)

```

```
END SUBROUTINE savedata
```

Listing 14.9: A Fortran subroutine to update arrays when solving the 2D Klein-Gordon equation.

```
SUBROUTINE storeold(Nx,Ny,unew,u,uold,vnew,v,vold)
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This subroutine copies arrays for a
! pseudospectral simulation of the 2D nonlinear Klein-Gordon equation
!
! u_{tt}-u_{xx}+u_{yy}+u=Es*u^3
!
! INPUT
!
! .. Parameters ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  .. Arrays ..
!  unew        = approximate solution
!  vnew        = Fourier transform of approximate solution
!  u         = approximate solution
!  v         = Fourier transform of approximate solution
!  uold        = approximate solution
!  vold        = Fourier transform of approximate solution
!
! OUTPUT
!
!  u         = approximate solution
!  v         = Fourier transform of approximate solution
!  uold        = approximate solution
!  vold        = Fourier transform of approximate solution
!
! LOCAL VARIABLES
!
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
```

```
! FURTHER COMMENTS
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
! OpenMP library
USE omp_lib
IMPLICIT NONE
! Declare variables
INTEGER(KIND=4), INTENT(IN)              :: Nx,Ny
COMPLEX(KIND=8), DIMENSION(1:NX,1:NY), INTENT(OUT)  :: vold,uold
COMPLEX(KIND=8), DIMENSION(1:NX,1:NY), INTENT(INOUT):: u,v
COMPLEX(KIND=8), DIMENSION(1:NX,1:NY), INTENT(IN) :: unew,vnew
INTEGER(kind=4)                    :: i,j

!$OMP PARALLEL PRIVATE(i,j)

!$OMP DO SCHEDULE(static)
DO j=1,Ny
  DO i=1,Nx
    vold(i,j)=v(i,j)
  END DO
END DO
!$OMP END DO NOWAIT

!$OMP DO SCHEDULE(static)
DO j=1,Ny
  DO i=1,Nx
    uold(i,j)=u(i,j)
  END DO
END DO
!$OMP END DO NOWAIT

!$OMP DO SCHEDULE(static)
DO j=1,Ny
  DO i=1,Nx
    u(i,j)=unew(i,j)
  END DO
END DO
!$OMP END DO NOWAIT

!$OMP DO SCHEDULE(static)
DO j=1,Ny
  DO i=1,Nx
    v(i,j)=vnew(i,j)
  END DO
END DO
!$OMP END DO NOWAIT

!$OMP END PARALLEL

```

```
END SUBROUTINE storeold
```

Listing 14.10: A Fortran subroutine to calculate the energy when solving the 2D Klein-Gordon equation.

```
SUBROUTINE enercalc(Nx,Ny,planfxy,planbxy,dt,Es,enkin,enstr,&
      enpot,en,kx,ky,temp1,temp2,v,vold,u,uold)
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This subroutine program calculates the energy for the nonlinear
! Klein-Gordon equation in 2 dimensions
! u_{tt}-u_{xx}+u_{yy}+u=Es*|u|^2u
!
! The energy density is given by
! 0.5u_t^2+0.5u_x^2+0.5u_y^2+0.5u^2+Es*0.25u^4
!
! INPUT
!
! .. Scalars ..
!  Nx       = number of modes in x - power of 2 for FFT
!  Ny       = number of modes in y - power of 2 for FFT
!  planfxy      = Forward 2d fft plan
!  planbxy      = Backward 2d fft plan
!  dt       = timestep
!  Es       = +1 for focusing, -1 for defocusing
! .. Arrays ..
!  u        = approximate solution
!  v        = Fourier transform of approximate solution
!  uold       = approximate solution
!  vold       = Fourier transform of approximate solution
!  temp1      = array to hold temporary values
!  temp2      = array to hold temporary values
! .. Vectors ..
!  kx       = fourier frequencies in x direction
!  ky       = fourier frequencies in y direction
!
! OUTPUT
!
! .. Scalars ..
!  enkin      = Kinetic energy
!  enstr      = Strain energy
!  enpot      = Potential energy
!  en       = Total energy
!
! LOCAL VARIABLES
!
! .. Scalars ..
!  j         = loop counter in y direction
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
!   FFTW3  -- Fast Fourier Transform in the West Library
!     (http://www.fftw.org/)
!   OpenMP library
USE omp_lib
IMPLICIT NONE
! Declare variables
INTEGER(KIND=4), INTENT(IN)              :: Nx,Ny
REAL(KIND=8), INTENT(IN)               :: dt,Es
INTEGER(KIND=8), INTENT(IN)              :: planfxy
INTEGER(KIND=8), INTENT(IN)              :: planbxy
COMPLEX(KIND=8), DIMENSION(1:Nx),INTENT(IN)     :: kx
COMPLEX(KIND=8), DIMENSION(1:Ny),INTENT(IN)     :: ky
COMPLEX(KIND=8), DIMENSION(1:Nx,1:Ny),INTENT(IN)  :: u,v,uold,vold
COMPLEX(KIND=8), DIMENSION(1:Nx,1:Ny),INTENT(INOUT) :: temp1,temp2
REAL(KIND=8), INTENT(OUT)              :: enkin,enstr
REAL(KIND=8), INTENT(OUT)              :: enpot,en
INTEGER(KIND=4)                    :: j

!.. Strain energy ..
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,Ny
  temp1(1:Nx,j)=0.5d0*kx(1:Nx)*(vold(1:Nx,j)+v(1:Nx,j))
END DO
!$OMP END PARALLEL DO
CALL dfftw_execute_dft_(planbxy,temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny))
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,Ny
  temp1(1:Nx,j)=abs(temp2(1:Nx,j)/REAL(Nx*Ny,kind(0d0)))**2
END DO
!$OMP END PARALLEL DO
CALL dfftw_execute_dft_(planfxy,temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny))
enstr=0.5d0*REAL(abs(temp2(1,1)),kind(0d0))/REAL(Nx*Ny,kind(0d0))
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,Ny
```

```
  temp1(1:Nx,j)=0.5d0*ky(j)*(vold(1:Nx,j)+v(1:Nx,j))
END DO
!$OMP END PARALLEL DO
CALL dfftw_execute_dft_(planbxy,temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny))
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,Ny
  temp1(1:Nx,j)=abs(temp2(1:Nx,j)/REAL(Nx*Ny,kind(0d0)))**2
END DO
!$OMP END PARALLEL DO
CALL dfftw_execute_dft_(planfxy,temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny))
enstr=enstr+0.5d0*REAL(abs(temp2(1,1)),kind(0d0))/REAL(Nx*Ny,kind(0d0))

! .. Kinetic Energy ..
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,Ny
  temp1(1:Nx,j)=( abs(u(1:Nx,j)-uold(1:Nx,j))/dt )**2
END DO
!$OMP END PARALLEL DO
CALL dfftw_execute_dft_(planfxy,temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny))
enkin=0.5d0*REAL(abs(temp2(1,1)),kind(0d0))/REAL(Nx*Ny,kind(0d0))

! .. Potential Energy ..
!$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
DO j=1,Ny
  temp1(1:Nx,j)=0.5d0*(abs((u(1:Nx,j)+uold(1:Nx,j))*0.50d0))**2&
        -0.125d0*Es*(abs(u(1:Nx,j))**4+abs(uold(1:Nx,j))**4)
END DO
!$OMP END PARALLEL DO
CALL dfftw_execute_dft_(planfxy,temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny))
enpot=REAL(abs(temp2(1,1)),kind(0d0))/REAL(Nx*Ny,kind(0d0))

en=enpot+enkin+enstr

END SUBROUTINE enercalc
```

Listing 14.11: A Fortran subroutine to save final results after solving the 2D Klein-Gordon equation.

```
SUBROUTINE saveresults(Nt,plotgap,time,en,enstr,enkin,enpot)
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This subroutine saves the energy and times stored during the
! computation for the nonlinear Klein-Gordon equation
!
! INPUT
!
! .. Parameters ..
```

```fortran
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
! .. Vectors ..
!  time        = times at which save data
!  en        = total energy
!  enstr       = strain energy
!  enpot       = potential energy
!  enkin       = kinetic energy
!
! OUTPUT
!
!
! LOCAL VARIABLES
!
! .. Scalars ..
!  n         = loop counter
! .. Arrays ..
!   name_config   = array to hold the filename
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
IMPLICIT NONE
! Declare variables
INTEGER(kind=4), INTENT(IN)              :: plotgap,Nt
REAL(KIND=8), DIMENSION(1+Nt/plotgap), INTENT(IN) :: enpot, enkin
REAL(KIND=8), DIMENSION(1+Nt/plotgap), INTENT(IN) :: en,enstr,time
INTEGER(kind=4)                        :: j
CHARACTER*100                        :: name_config

name_config = 'tdata.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,1+Nt/plotgap
  WRITE(11,*) time(j)
END DO
CLOSE(11)

name_config = 'en.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
```

```
  DO j=1,1+Nt/plotgap
    WRITE(11,*) en(j)
  END DO
  CLOSE(11)

  name_config = 'enkin.dat'
  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
  REWIND(11)
  DO j=1,1+Nt/plotgap
    WRITE(11,*) enkin(j)
  END DO
  CLOSE(11)

  name_config = 'enpot.dat'
  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
  REWIND(11)
  DO j=1,1+Nt/plotgap
    WRITE(11,*) enpot(j)
  END DO
  CLOSE(11)

  name_config = 'enstr.dat'
  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
  REWIND(11)
  DO j=1,1+Nt/plotgap
    WRITE(11,*) enstr(j)
  END DO
  CLOSE(11)

  END SUBROUTINE saveresults
```

Listing 14.12: An example makefile for compiling the OpenMP program in listing 14.5.

```
#define the complier
COMPILER = mpif90
# compilation settings, optimization, precision, parallelization
  FLAGS = -O0 -mp

# libraries
LIBS = -L${FFTW_LINK} -lfftw3_threads -lfftw3 -lm
# source list for main program
SOURCES =  KgSemiImp2d.f90 initialdata.f90 savedata.f90 getgrid.f90 \
      storeold.f90 saveresults.f90 enercalc.f90

test: $(SOURCES)
    ${COMPILER} -o kg $(FLAGS) $(SOURCES) $(LIBS)

clean:
  rm *.o
```

Listing 14.13: A Matlab program to plot the fields produced by listing 14.5.

```
% A program to create a video of the computed results

clear all; format compact, format short,
set(0,'defaultaxesfontsize',14,'defaultaxeslinewidth',.7,...
    'defaultlinelinewidth',2,'defaultpatchlinewidth',3.5);

% Load data
% Get coordinates
X=load('./xcoord.dat');
Y=load('./ycoord.dat');
TIME=load('./tdata.dat');
% find number of grid points
Nx=length(X);
Ny=length(Y);

% reshape coordinates to allow easy plotting
[xx,yy]=ndgrid(X,Y);

nplots=length(TIME);

for i =1:nplots
    %
    % Open file and dataset using the default properties.
    %
    FILE=['./data/u',num2str(9999999+i),'.datbin'];
    FILEPIC=['./data/pic',num2str(9999999+i),'.jpg'];
    fid=fopen(FILE,'r');
    [fname,mode,mformat]=fopen(fid);
    u=fread(fid,Nx*Ny,'real*8');
    u=reshape(u,Nx,Ny);
    % close files
    fclose(fid);
    %
    % Plot data on the screen.
    %
    figure(2);clf; mesh(xx,yy,real(u)); xlabel x; ylabel y;
    title(['Time ',num2str(TIME(i))]); colorbar; axis square;
    drawnow; frame=getframe(2); saveas(2,FILEPIC,'jpg');
end
```

14.1.3 A Three-Dimensional MPI Fortran Program using 2DECOMP&FFT

We now give a program for the three-dimensional nonlinear Klein-Gordon equation. The program uses the same subroutine structure as the two-dimensional code. To make the program easy to reuse, the subroutine listed in listing 14.21 has been created to read an INPUTFILE which specifies the parameters to use for the program and so the program does not need to be recompiled every time it is run. To enable the program to scale better, the arrays which hold the Fourier frequencies and grid points have also been decomposed so that only the portions of the arrays used on each processor are created and stored on the processor. A further addition is a short postprocessing program to create header files to allow one to use the bov (brick of values) format that allows one to use the parallel visualization software VisIt. The program is listed in listing 14.23, to use this program simply compile it using gfortran, no special flags are required, and then run it in the directory from which the INPUTFILE and data are stored. The program VisIt can be downloaded from `https://wci.llnl.gov/codes/visit/home.html`. This program also run on laptops, desktops as well as parallel computer clusters. Documentation on using VisIt is available here `https://wci.llnl.gov/codes/visit/manuals.html` and here `http://www.visitusers.org/index.php?title=Main_Page`. A short video tutorial on how to use VisIt remotely is available at `http://cac.engin.umich.edu/resources/software/visit.html`.

Listing 14.14: A Fortran program to solve the 3D Klein-Gordon equation.

```
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This program solves nonlinear Klein-Gordon equation in 3 dimensions
! u_{tt}-(u_{xx}+u_{yy}+u_{zz})+u=Es*|u|^2u
! using a second order implicit-explicit time stepping scheme.
!
! The boundary conditions are u(x=-Lx*pi,y,z)=u(x=Lx*\pi,y,z),
! u(x,y=-Ly*pi,z)=u(x,y=Ly*pi,z),u(x,y,z=-Ly*pi)=u(x,y,z=Ly*pi),
! The initial condition is u=0.5*exp(-x^2-y^2-z^2)*sin(10*x+12*y)
!
! .. Parameters ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  Nz        = number of modes in z - power of 2 for FFT
!  Nt        = number of timesteps to take
!  Tmax        = maximum simulation time
!  plotgap       = number of timesteps between plots
!  pi = 3.14159265358979323846264338327950288419716939937510d0
!  Lx        = width of box in x direction
!  Ly        = width of box in y direction
!  Lz        = width of box in z direction
!  ES        = +1 for focusing and -1 for defocusing
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!  k         = loop counter in z direction
!  n         = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  start       = variable to record start time of program
!  finish      = variable to record end time of program
!  count_rate    = variable for clock count rate
!  dt       = timestep
!  modescalereal  = Number to scale after backward FFT
!  ierr       = error code
!  plotnum      = number of plot
!  myid       = Process id
!  p_row      = number of rows for domain decomposition
!  p_col      = number of columns for domain decomposition
!  filesize     = total filesize
!  disp       = displacement to start writing data from
! .. Arrays ..
!  unew       = approximate solution
!  vnew       = Fourier transform of approximate solution
!  u        = approximate solution
!  v        = Fourier transform of approximate solution
!  uold       = approximate solution
!  vold       = Fourier transform of approximate solution
!  nonlin       = nonlinear term, u^3
!  nonlinhat    = Fourier transform of nonlinear term, u^3
! .. Vectors ..
!  kx       = fourier frequencies in x direction
!  ky       = fourier frequencies in y direction
!  kz       = fourier frequencies in z direction
!  x        = x locations
!  y        = y locations
!  z        = z locations
!  time       = times at which save data
!  en       = total energy
!  enstr      = strain energy
!  enpot      = potential energy
!  enkin      = kinetic energy
!  name_config    = array to store filename for data to be saved
!  fftfxyz      = array to setup 2D Fourier transform
!  fftbxyz      = array to setup 2D Fourier transform
! .. Special Structures ..
!  decomp     = contains information on domain decomposition
!         see http://www.2decomp.org/ for more information
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
! getgrid.f90 -- Get initial grid of points
!   initialdata.f90 -- Get initial data
! enercalc.f90 -- Subroutine to calculate the energy
! savedata.f90 -- Save initial data
! storeold.f90 -- Store old data
! External libraries required
!   2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
!     (http://www.2decomp.org/index.html)
! MPI library
PROGRAM Kg
IMPLICIT NONE
USE decomp_2d
USE decomp_2d_fft
USE decomp_2d_io
INCLUDE 'mpif.h'
! Declare variables
INTEGER(kind=4)       :: Nx, Ny, Nz, Nt, plotgap
REAL(kind=8), PARAMETER   :: &
  pi=3.14159265358979323846264338327950288419716939937510d0
REAL(kind=8)        ::  Lx,Ly,Lz,Es,dt,starttime,modescalereal
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  kx,ky,kz
REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  x,y,z
COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE::  u,nonlin
COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE::  v,nonlinhat
COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE::  uold
COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE::  vold
COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE::  unew
COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE::  vnew
REAL(kind=8), DIMENSION(:,:,:), ALLOCATABLE :: savearray
REAL(kind=8), DIMENSION(:), ALLOCATABLE ::  time,enkin,enstr,enpot,en
INTEGER(kind=4)   ::  ierr,i,j,k,n,allocatestatus,myid,numprocs
INTEGER(kind=4)   ::  start, finish, count_rate, plotnum
TYPE(DECOMP_INFO) ::  decomp
INTEGER(kind=MPI_OFFSET_KIND) :: filesize, disp
INTEGER(kind=4) ::  p_row=0, p_col=0
CHARACTER*100 ::  name_config
    ! initialisation of 2DECOMP&FFT
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

CALL readinputfile(Nx,Ny,Nz,Nt,plotgap,Lx,Ly,Lz, &
           Es,DT,starttime,myid,ierr)
! do automatic domain decomposition
CALL decomp_2d_init(Nx,Ny,Nz,p_row,p_col)
! get information about domain decomposition choosen
CALL decomp_info_init(Nx,Ny,Nz,decomp)
! initialise FFT library
CALL decomp_2d_fft_init
ALLOCATE(kx(decomp%zst(1):decomp%zen(1)),&
    ky(decomp%zst(2):decomp%zen(2)),&
```

```
      kz(decomp%zst(3):decomp%zen(3)),&
       x(decomp%xst(1):decomp%xen(1)),&
       y(decomp%xst(2):decomp%xen(2)),&
       z(decomp%xst(3):decomp%xen(3)),&
      u(decomp%xst(1):decomp%xen(1),&
          decomp%xst(2):decomp%xen(2),&
          decomp%xst(3):decomp%xen(3)),&
      v(decomp%zst(1):decomp%zen(1),&
              decomp%zst(2):decomp%zen(2),&
              decomp%zst(3):decomp%zen(3)),&
      nonlin(decomp%xst(1):decomp%xen(1),&
          decomp%xst(2):decomp%xen(2),&
          decomp%xst(3):decomp%xen(3)),&
      nonlinhat(decomp%zst(1):decomp%zen(1),&
              decomp%zst(2):decomp%zen(2),&
              decomp%zst(3):decomp%zen(3)),&
      uold(decomp%xst(1):decomp%xen(1),&
          decomp%xst(2):decomp%xen(2),&
          decomp%xst(3):decomp%xen(3)),&
      vold(decomp%zst(1):decomp%zen(1),&
              decomp%zst(2):decomp%zen(2),&
              decomp%zst(3):decomp%zen(3)),&
      unew(decomp%xst(1):decomp%xen(1),&
          decomp%xst(2):decomp%xen(2),&
          decomp%xst(3):decomp%xen(3)),&
      vnew(decomp%zst(1):decomp%zen(1),&
              decomp%zst(2):decomp%zen(2),&
              decomp%zst(3):decomp%zen(3)),&
      savearray(decomp%xst(1):decomp%xen(1),&
          decomp%xst(2):decomp%xen(2),&
          decomp%xst(3):decomp%xen(3)),&
      time(1:1+Nt/plotgap),enkin(1:1+Nt/plotgap),&
      enstr(1:1+Nt/plotgap),enpot(1:1+Nt/plotgap),&
      en(1:1+Nt/plotgap),stat=allocatestatus)
  IF (allocatestatus .ne. 0) stop
  IF (myid.eq.0) THEN
    PRINT *,'allocated arrays'
  END IF
  ! setup fourier frequencies
  CALL getgrid(myid,Nx,Ny,Nz,Lx,Ly,Lz,pi,name_config,x,y,z,kx,ky,kz,decomp
     )
  IF (myid.eq.0) THEN
    PRINT *,'Setup grid and fourier frequencies'
  END IF
  CALL initialdata(Nx,Ny,Nz,x,y,z,u,uold,decomp)
  plotnum=1
  name_config = 'data/u'
  savearray=REAL(u)
  CALL savedata(Nx,Ny,Nz,plotnum,name_config,savearray,decomp)

  CALL decomp_2d_fft_3d(u,v,DECOMP_2D_FFT_FORWARD)
```

```fortran
CALL decomp_2d_fft_3d(uold,vold,DECOMP_2D_FFT_FORWARD)

modescalereal=1.0d0/REAL(Nx,KIND(0d0))
modescalereal=modescalereal/REAL(Ny,KIND(0d0))
modescalereal=modescalereal/REAL(Nz,KIND(0d0))

CALL enercalc(myid,Nx,Ny,Nz,dt,Es,modescalereal,&
      enkin(plotnum),enstr(plotnum),&
      enpot(plotnum),en(plotnum),&
      kx,ky,kz,nonlin,nonlinhat,&
      v,vold,u,uold,decomp)

IF (myid.eq.0) THEN
  PRINT *,'Got initial data, starting timestepping'
END IF
time(plotnum)=0.0d0+starttime
  CALL system_clock(start,count_rate)
DO n=1,Nt
  DO k=decomp%xst(3),decomp%xen(3)
    DO j=decomp%xst(2),decomp%xen(2)
      DO i=decomp%xst(1),decomp%xen(1)
        nonlin(i,j,k)=(abs(u(i,j,k))*2)*u(i,j,k)
      END DO
    END DO
  END DO
  CALL decomp_2d_fft_3d(nonlin,nonlinhat,DECOMP_2D_FFT_FORWARD)
  DO k=decomp%zst(3),decomp%zen(3)
    DO j=decomp%zst(2),decomp%zen(2)
      DO i=decomp%zst(1),decomp%zen(1)
        vnew(i,j,k)=&
        ( 0.25*(kx(i)*kx(i) + ky(j)*ky(j)+ kz(k)*kz(k)-1.0d0)&
        *(2.0d0*v(i,j,k)+vold(i,j,k))&
        +(2.0d0*v(i,j,k)-vold(i,j,k))/(dt*dt)&
        +Es*nonlinhat(i,j,k) )&
        /(1/(dt*dt)-0.25*(kx(i)*kx(i)+ ky(j)*ky(j)+ kz(k)*kz(k)-1.0d0))
      END DO
    END DO
  END DO
  CALL decomp_2d_fft_3d(vnew,unew,DECOMP_2D_FFT_BACKWARD)
  ! normalize result
  DO k=decomp%xst(3),decomp%xen(3)
    DO j=decomp%xst(2),decomp%xen(2)
      DO i=decomp%xst(1),decomp%xen(1)
        unew(i,j,k)=unew(i,j,k)*modescalereal
      END DO
    END DO
  END DO
  IF (mod(n,plotgap)==0) THEN
    plotnum=plotnum+1
    time(plotnum)=n*dt+starttime
    IF (myid.eq.0) THEN
```

```
      PRINT *,'time',n*dt+starttime
    END IF
    CALL enercalc(myid,Nx,Ny,Nz,dt,Es,modescalereal,&
        enkin(plotnum),enstr(plotnum),&
        enpot(plotnum),en(plotnum),&
        kx,ky,kz,nonlin,nonlinhat,&
        vnew,v,unew,u,decomp)
    savearray=REAL(unew,kind(0d0))
    CALL savedata(Nx,Ny,Nz,plotnum,name_config,savearray,decomp)
  END IF
    ! .. Update old values ..
  CALL storeold(Nx,Ny,Nz,unew,u,uold,vnew,v,vold,decomp)
END DO
CALL system_clock(finish,count_rate)
IF (myid.eq.0) THEN
  PRINT *,'Finished time stepping'
  PRINT*,'Program took ',&
    REAL(finish-start,kind(0d0))/REAL(count_rate,kind(0d0)),&
    'for Time stepping'
  CALL saveresults(Nt,plotgap,time,en,enstr,enkin,enpot)
  ! Save times at which output was made in text format
  PRINT *,'Saved data'
END IF
CALL decomp_2d_fft_finalize
  CALL decomp_2d_finalize

DEALLOCATE(kx,ky,kz,x,y,z,u,v,nonlin,nonlinhat,savearray,&
      uold,vold,unew,vnew,time,enkin,enstr,enpot,en,&
      stat=allocatestatus)
IF (allocatestatus .ne. 0) STOP
IF (myid.eq.0) THEN
  PRINT *,'Deallocated arrays'
  PRINT *,'Program execution complete'
END IF
CALL MPI_FINALIZE(ierr)

END PROGRAM Kg
```

Listing 14.15: A Fortran subroutine to get the grid to solve the 3D Klein-Gordon equation on.

```
SUBROUTINE getgrid(myid,Nx,Ny,Nz,Lx,Ly,Lz,pi,name_config,x,y,z,kx,ky,kz,
    decomp)
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This subroutine gets grid points and fourier frequencies for a
! pseudospectral simulation of the 2D nonlinear Klein-Gordon equation
```

```
!
! u_{tt}-(u_{xx}+u_{yy}+u_{zz})+u=Es*u^3
!
! The boundary conditions are u(x=-Lx*pi,y,z)=u(x=Lx*\pi,y,z),
! u(x,y=-Ly*pi,z)=u(x,y=Ly*pi,z),u(x,y,z=-Ly*pi)=u(x,y,z=Ly*pi),
!
! INPUT
!
! .. Scalars ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  Ny        = number of modes in z - power of 2 for FFT
!  pi        = 3.142....
!  Lx        = width of box in x direction
!  Ly        = width of box in y direction
!  Lz        = width of box in z direction
!  myid        = processor id
! .. Special Structures ..
!  decomp      = contains information on domain decomposition
!          see http://www.2decomp.org/ for more information
!
! OUTPUT
!
! .. Vectors ..
!  kx        = fourier frequencies in x direction
!  ky        = fourier frequencies in y direction
!  kz        = fourier frequencies in z direction
!  x         = x locations
!  y         = y locations
!  z         = z locations
!
! LOCAL VARIABLES
!
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!  k         = loop counter in z direction
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
```

```
!
! External libraries required
! 2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
!     (http://www.2decomp.org/index.html)
! MPI library
IMPLICIT NONE
USE decomp_2d
INCLUDE 'mpif.h'
! Declare variables
INTEGER(KIND=4), INTENT(IN)                :: myid,Nx,Ny,Nz
REAL(kind=8), INTENT(IN)                :: Lx,Ly,Lz,pi
TYPE(DECOMP_INFO), INTENT(IN)             ::  decomp
REAL(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1)), INTENT(OUT)   :: x
REAL(KIND=8), DIMENSION(decomp%xst(2):decomp%xen(2)), INTENT(OUT)   :: y
REAL(KIND=8), DIMENSION(decomp%xst(3):decomp%xen(3)), INTENT(OUT)   :: z
COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1)), INTENT(OUT)::
   kx
COMPLEX(KIND=8), DIMENSION(decomp%zst(2):decomp%zen(2)), INTENT(OUT)::
   ky
COMPLEX(KIND=8), DIMENSION(decomp%zst(3):decomp%zen(3)), INTENT(OUT)::
   kz
CHARACTER*100, INTENT(OUT)                :: name_config
INTEGER(kind=4)                      :: i,j,k


DO i = 1,1+ Nx/2
  IF ((i.GE.decomp%zst(1)).AND.(i.LE.decomp%zen(1))) THEN
    kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
  END IF
END DO
IF ((Nx/2 + 1 .GE.decomp%zst(1)).AND.(Nx/2 + 1 .LE.decomp%zen(1))) THEN
  kx( Nx/2 + 1 ) = 0.0d0
ENDIF
DO i = Nx/2+2, Nx
  IF ((i.GE.decomp%zst(1)).AND.(i.LE.decomp%zen(1))) THEN
    Kx( i) = cmplx(0.0d0,-1.0d0)*REAL(1-i+Nx,KIND(0d0))/Lx
  ENDIF
END DO
DO i=decomp%xst(1),decomp%xen(1)
  x(i)=(-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)))*pi*Lx
END DO

DO j = 1,1+ Ny/2
  IF ((j.GE.decomp%zst(2)).AND.(j.LE.decomp%zen(2))) THEN
    ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
  END IF
END DO
IF ((Ny/2 + 1 .GE.decomp%zst(2)).AND.(Ny/2 + 1 .LE.decomp%zen(2))) THEN
  ky( Ny/2 + 1 ) = 0.0d0
ENDIF
DO j = Ny/2+2, Ny
```

```fortran
    IF ((j.GE.decomp%zst(2)).AND.(j.LE.decomp%zen(2))) THEN
      ky(j) = cmplx(0.0d0,-1.0d0)*REAL(1-j+Ny,KIND(0d0))/Ly
    ENDIF
  END DO
  DO j=decomp%xst(2),decomp%xen(2)
    y(j)=(-1.0d0 + 2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)))*pi*Ly
  END DO

  DO k = 1,1+ Nz/2
    IF ((k.GE.decomp%zst(3)).AND.(k.LE.decomp%zen(3))) THEN
      kz(k)= cmplx(0.0d0,1.0d0)*REAL(k-1,kind(0d0))/Lz
    END IF
  END DO
  IF ((Nz/2 + 1 .GE.decomp%zst(3)).AND.(Nz/2 + 1 .LE.decomp%zen(3))) THEN
    kz( Nz/2 + 1 ) = 0.0d0
  ENDIF
  DO k = Nz/2+2, Nz
    IF ((k.GE.decomp%zst(3)).AND.(k.LE.decomp%zen(3))) THEN
      kz(k) = cmplx(0.0d0,-1.0d0)*REAL(1-k+Nz,KIND(0d0))/Lz
    ENDIF
  END DO
  DO k=decomp%xst(3),decomp%xen(3)
    z(k)=(-1.0d0 + 2.0d0*REAL(k-1,kind(0d0))/REAL(Nz,kind(0d0)))*pi*Lz
  END DO

  IF (myid.eq.0) THEN
    ! Save x grid points in text format
    name_config = 'xcoord.dat'
    OPEN(unit=11,FILE=name_config,status="UNKNOWN")
    REWIND(11)
    DO i=1,Nx
      WRITE(11,*) (-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)))*
          pi*Lx
    END DO
    CLOSE(11)
    ! Save y grid points in text format
    name_config = 'ycoord.dat'
    OPEN(unit=11,FILE=name_config,status="UNKNOWN")
    REWIND(11)
    DO j=1,Ny
      WRITE(11,*) (-1.0d0 + 2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)))*
          pi*Ly
    END DO
    CLOSE(11)
    ! Save z grid points in text format
    name_config = 'zcoord.dat'
    OPEN(unit=11,FILE=name_config,status="UNKNOWN")
    REWIND(11)
    DO k=1,Nz
      WRITE(11,*) (-1.0d0 + 2.0d0*REAL(k-1,kind(0d0))/REAL(Nz,kind(0d0)))*
          pi*Lz
    END DO
    CLOSE(11)
  END IF

  END SUBROUTINE getgrid
```

Listing 14.16: A Fortran subroutine to get the initial data to solve the 3D Klein-Gordon equation for.

```
  SUBROUTINE initialdata(Nx,Ny,Nz,x,y,z,u,uold,decomp)
  !--------------------------------------------------------------------
  !
  !
  ! PURPOSE
  !
  ! This subroutine gets initial data for nonlinear Klein-Gordon equation
  ! in 3 dimensions
  ! u_{tt}-(u_{xx}+u_{yy}+u_{zz})+u=Es*u^3+
  !
  ! The boundary conditions are u(x=-Lx*\pi,y,z)=u(x=Lx*\pi,y,z),
  ! u(x,y=-Ly*\pi,z)=u(x,y=Ly*\pi,z),u(x,y,z=-Ly*\pi)=u(x,y,z=Ly*\pi)
  ! The initial condition is u=0.5*exp(-x^2-y^2-z^2)*sin(10*x+12*y)
  !
  ! INPUT
  !
  ! .. Parameters ..
  !  Nx        = number of modes in x - power of 2 for FFT
  !  Ny        = number of modes in y - power of 2 for FFT
  !  Nz        = number of modes in z - power of 2 for FFT
  ! .. Vectors ..
  !  x         = x locations
  !  y         = y locations
  !  z         = z locations
  ! .. Special Structures ..
  !  decomp      = contains information on domain decomposition
  !          see http://www.2decomp.org/ for more information
  ! OUTPUT
  !
  ! .. Arrays ..
  !  u         = initial solution
  !  uold        = approximate solution based on time derivative of
  !          initial solution
  !
  ! LOCAL VARIABLES
  !
  ! .. Scalars ..
  !  i         = loop counter in x direction
  !  j         = loop counter in y direction
  !  k         = loop counter in z direction
  !
```

```
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
! 2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
!     (http://www.2decomp.org/index.html)
! MPI library
IMPLICIT NONE
USE decomp_2d
INCLUDE 'mpif.h'
! Declare variables
INTEGER(KIND=4), INTENT(IN)              :: Nx,Ny,Nz
TYPE(DECOMP_INFO), INTENT(IN)            ::  decomp
REAL(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1)), INTENT(IN) :: x
REAL(KIND=8), DIMENSION(decomp%xst(2):decomp%xen(2)), INTENT(IN) :: y
REAL(KIND=8), DIMENSION(decomp%xst(3):decomp%xen(3)), INTENT(IN) :: z
COMPLEX(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
              decomp%xst(2):decomp%xen(2),&
              decomp%xst(3):decomp%xen(3)),&
                      INTENT(OUT) :: u,uold
INTEGER(kind=4)                    :: i,j,k

DO k=decomp%xst(3),decomp%xen(3)
  DO j=decomp%xst(2),decomp%xen(2)
    DO i=decomp%xst(1),decomp%xen(1)
      u(i,j,k)=0.5d0*exp(-1.0d0*(x(i)**2 +y(j)**2+z(k)**2))!*&
        !sin(10.0d0*x(i)+12.0d0*y(j))
    END DO
  END DO
END DO
DO k=decomp%xst(3),decomp%xen(3)
  DO j=decomp%xst(2),decomp%xen(2)
    DO i=decomp%xst(1),decomp%xen(1)
      uold(i,j,k)=0.5d0*exp(-1.0d0*(x(i)**2 +y(j)**2+z(k)**2))!*&
        !sin(10.0d0*x(i)+12.0d0*y(j))
      END DO
  END DO
END DO

END SUBROUTINE initialdata
```

Listing 14.17: A Fortran program to save a field from the solution of the 3D Klein-Gordon equation.

```
SUBROUTINE savedata(Nx,Ny,Nz,plotnum,name_config,field,decomp)
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This subroutine saves a three dimensional real array in binary
! format
!
! INPUT
!
! .. Scalars ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  Nz        = number of modes in z - power of 2 for FFT
!  plotnum       = number of plot to be made
! .. Arrays ..
!  field      = real data to be saved
!  name_config     = root of filename to save to
!
! .. Output ..
! plotnum     = number of plot to be saved
! .. Special Structures ..
!  decomp     = contains information on domain decomposition
!         see http://www.2decomp.org/ for more information
! LOCAL VARIABLES
!
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!  k         = loop counter in z direction
!  count      = counter
!  iol        = size of file
! .. Arrays ..
!   number_file   = array to hold the number of the plot
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
! 2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
!     (http://www.2decomp.org/index.html)
! MPI library
IMPLICIT NONE
USE decomp_2d
USE decomp_2d_fft
USE decomp_2d_io
INCLUDE 'mpif.h'
! Declare variables
INTEGER(KIND=4), INTENT(IN)           :: Nx,Ny,Nz
INTEGER(KIND=4), INTENT(IN)           :: plotnum
TYPE(DECOMP_INFO), INTENT(IN)         ::  decomp
REAL(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
              decomp%xst(2):decomp%xen(2),&
              decomp%xst(3):decomp%xen(3)), &
                    INTENT(IN) :: field
CHARACTER*100, INTENT(IN)           :: name_config
INTEGER(kind=4)                   :: i,j,k,iol,count,ind
CHARACTER*100                     :: number_file

! create character array with full filename
ind = index(name_config,' ') - 1
WRITE(number_file,'(i0)') 10000000+plotnum
number_file = name_config(1:ind)//number_file
ind = index(number_file,' ') - 1
number_file = number_file(1:ind)//'.datbin'
CALL decomp_2d_write_one(1,field,number_file)

END SUBROUTINE savedata
```

Listing 14.18: A Fortran subroutine to update arrays when solving the 3D Klein-Gordon equation.

```
SUBROUTINE storeold(Nx,Ny,Nz,unew,u,uold,vnew,v,vold,decomp)
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This subroutine copies arrays for a
! pseudospectral simulation of the 2D nonlinear Klein-Gordon equation
!
! u_{tt}-(u_{xx}+u_{yy}+u_{zz})+u=Es*u^3
!
! INPUT
!
! .. Parameters ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  Nz        = number of modes in z - power of 2 for FFT
!  .. Arrays ..
!  unew       = approximate solution
!  vnew       = Fourier transform of approximate solution
!  u         = approximate solution
!  v         = Fourier transform of approximate solution
!  uold       = approximate solution
!  vold       = Fourier transform of approximate solution
! .. Special Structures ..
!  decomp     = contains information on domain decomposition
!          see http://www.2decomp.org/ for more information
! OUTPUT
!
!  u         = approximate solution
!  v         = Fourier transform of approximate solution
!  uold       = approximate solution
!  vold       = Fourier transform of approximate solution
!
! LOCAL VARIABLES
!
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!  k         = loop counter in z direction
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!---------------------------------------------------------------------
! External routines required
!
! External libraries required
! 2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
!     (http://www.2decomp.org/index.html)
! MPI library
IMPLICIT NONE
USE decomp_2d
USE decomp_2d_fft
USE decomp_2d_io
INCLUDE 'mpif.h'
! Declare variables
INTEGER(KIND=4), INTENT(IN)                :: Nx,Ny,Nz
TYPE(DECOMP_INFO), INTENT(IN)              ::  decomp
COMPLEX(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
                decomp%xst(2):decomp%xen(2),&
                decomp%xst(3):decomp%xen(3)), INTENT(OUT):: uold
```

```
COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1),&
                    decomp%zst(2):decomp%zen(2),&
                    decomp%zst(3):decomp%zen(3)), INTENT(OUT):: vold
COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1),&
                    decomp%zst(2):decomp%zen(2),&
                    decomp%zst(3):decomp%zen(3)), INTENT(INOUT):: v
COMPLEX(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
                  decomp%xst(2):decomp%xen(2),&
                  decomp%xst(3):decomp%xen(3)), INTENT(INOUT):: u
COMPLEX(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
                  decomp%xst(2):decomp%xen(2),&
                  decomp%xst(3):decomp%xen(3)), INTENT(IN):: unew
COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1),&
                    decomp%zst(2):decomp%zen(2),&
                    decomp%zst(3):decomp%zen(3)), INTENT(IN):: vnew
INTEGER(kind=4)                             :: i,j,k

DO k=decomp%zst(3),decomp%zen(3)
  DO j=decomp%zst(2),decomp%zen(2)
    DO i=decomp%zst(1),decomp%zen(1)
      vold(i,j,k)=v(i,j,k)
    END DO
  END DO
END DO
DO k=decomp%xst(3),decomp%xen(3)
  DO j=decomp%xst(2),decomp%xen(2)
    DO i=decomp%xst(1),decomp%xen(1)
      uold(i,j,k)=u(i,j,k)
    END DO
  END DO
END DO
DO k=decomp%xst(3),decomp%xen(3)
  DO j=decomp%xst(2),decomp%xen(2)
    DO i=decomp%xst(1),decomp%xen(1)
      u(i,j,k)=unew(i,j,k)
    END DO
  END DO
END DO
DO k=decomp%zst(3),decomp%zen(3)
  DO j=decomp%zst(2),decomp%zen(2)
    DO i=decomp%zst(1),decomp%zen(1)
      v(i,j,k)=vnew(i,j,k)
    END DO
  END DO
END DO

END SUBROUTINE storeold
```

Listing 14.19: A Fortran subroutine to calculate the energy when solving the 3D Klein-Gordon equation.

```fortran
SUBROUTINE enercalc(myid,Nx,Ny,Nz,dt,Es,modescalereal,enkin,enstr,&
      enpot,en,kx,ky,kz,tempu,tempv,v,vold,u,uold,decomp)
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This subroutine program calculates the energy for the nonlinear
! Klein-Gordon equation in 3 dimensions
! u_{tt}-(u_{xx}+u_{yy}+u_{zz})+u=Es*|u|^2u
!
! The energy density is given by
! 0.5u_t^2+0.5u_x^2+0.5u_y^2+0.5u_z^2+0.5u^2+Es*0.25u^4
!
! INPUT
!
! .. Scalars ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  Nz        = number of modes in z - power of 2 for FFT
!  dt        = timestep
!  Es        = +1 for focusing, -1 for defocusing
!  modescalereal  = parameter to scale after doing backward FFT
!  myid        = Process id
! .. Arrays ..
!  u         = approximate solution
!  v         = Fourier transform of approximate solution
!  uold        = approximate solution
!  vold        = Fourier transform of approximate solution
!  tempu       = array to hold temporary values - real space
!  tempv       = array to hold temporary values - fourier space
! .. Vectors ..
!  kx        = fourier frequencies in x direction
!  ky        = fourier frequencies in y direction
!  kz        = fourier frequencies in z direction
! .. Special Structures ..
!  decomp      = contains information on domain decomposition
!          see http://www.2decomp.org/ for more information
! OUTPUT
!
! .. Scalars ..
!  enkin       = Kinetic energy
!  enstr       = Strain energy
!  enpot       = Potential energy
!  en        = Total energy
!
! LOCAL VARIABLES
!
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!  k         = loop counter in z direction
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
! 2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
!     (http://www.2decomp.org/index.html)
! MPI library
IMPLICIT NONE
USE decomp_2d
USE decomp_2d_fft
USE decomp_2d_io
INCLUDE 'mpif.h'
! Declare variables
INTEGER(KIND=4), INTENT(IN)                 :: Nx,Ny,Nz,myid
REAL(KIND=8), INTENT(IN)                  :: dt,Es,modescalereal
TYPE(DECOMP_INFO), INTENT(IN)               :: decomp
COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1)),INTENT(IN)  ::
   kx
COMPLEX(KIND=8), DIMENSION(decomp%zst(2):decomp%zen(2)),INTENT(IN)  ::
   ky
COMPLEX(KIND=8), DIMENSION(decomp%zst(3):decomp%zen(3)),INTENT(IN)  ::
   kz
COMPLEX(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
                decomp%xst(2):decomp%xen(2),&
                decomp%xst(3):decomp%xen(3)),&
                          INTENT(IN)  :: u,uold
COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1),&
                    decomp%zst(2):decomp%zen(2),&
                    decomp%zst(3):decomp%zen(3)),&
                              INTENT(IN)  :: v,vold
COMPLEX(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
                decomp%xst(2):decomp%xen(2),&
                decomp%xst(3):decomp%xen(3)),&
                          INTENT(INOUT) :: tempu
COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1),&
                    decomp%zst(2):decomp%zen(2),&
```

```fortran
                      decomp%zst(3):decomp%zen(3)),&
                                 INTENT(INOUT):: tempv
REAL(KIND=8), INTENT(OUT)              :: enkin,enstr
REAL(KIND=8), INTENT(OUT)              :: enpot,en
INTEGER(KIND=4)                    :: i,j,k

!.. Strain energy ..
DO k=decomp%zst(3),decomp%zen(3)
  DO j=decomp%zst(2),decomp%zen(2)
    DO i=decomp%zst(1),decomp%zen(1)
      tempv(i,j,k)=0.5d0*kx(i)*(vold(i,j,k)+v(i,j,k))
    END DO
  END DO
END DO
CALL decomp_2d_fft_3d(tempv,tempu,DECOMP_2D_FFT_BACKWARD)

DO k=decomp%xst(3),decomp%xen(3)
  DO j=decomp%xst(2),decomp%xen(2)
    DO i=decomp%xst(1),decomp%xen(1)
      tempu(i,j,k)=abs(tempu(i,j,k)*modescalereal)**2
    END DO
  END DO
END DO
CALL decomp_2d_fft_3d(tempu,tempv,DECOMP_2D_FFT_FORWARD)
IF(myid.eq.0) THEN
  enstr=0.5d0*REAL(abs(tempv(1,1,1)),kind(0d0))
END IF
DO k=decomp%zst(3),decomp%zen(3)
  DO j=decomp%zst(2),decomp%zen(2)
    DO i=decomp%zst(1),decomp%zen(1)
      tempv(i,j,k)=0.5d0*ky(j)*(vold(i,j,k)+v(i,j,k))
    END DO
  END DO
END DO
CALL decomp_2d_fft_3d(tempv,tempu,DECOMP_2D_FFT_BACKWARD)
DO k=decomp%xst(3),decomp%xen(3)
  DO j=decomp%xst(2),decomp%xen(2)
    DO i=decomp%xst(1),decomp%xen(1)
      tempu(i,j,k)=abs(tempu(i,j,k)*modescalereal)**2
    END DO
  END DO
END DO
CALL decomp_2d_fft_3d(tempu,tempv,DECOMP_2D_FFT_FORWARD)
IF(myid.eq.0) THEN
  enstr=enstr+0.5d0*REAL(abs(tempv(1,1,1)),kind(0d0))
END IF
DO k=decomp%zst(3),decomp%zen(3)
  DO j=decomp%zst(2),decomp%zen(2)
    DO i=decomp%zst(1),decomp%zen(1)
      tempv(i,j,k)=0.5d0*kz(k)*(vold(i,j,k)+v(i,j,k))
    END DO
  END DO
END DO
CALL decomp_2d_fft_3d(tempv,tempu,DECOMP_2D_FFT_BACKWARD)
DO k=decomp%xst(3),decomp%xen(3)
  DO j=decomp%xst(2),decomp%xen(2)
    DO i=decomp%xst(1),decomp%xen(1)
      tempu(i,j,k)=abs(tempu(i,j,k)*modescalereal)**2
    END DO
  END DO
END DO
CALL decomp_2d_fft_3d(tempu,tempv,DECOMP_2D_FFT_FORWARD)
IF(myid.eq.0) THEN
  enstr=enstr+0.5d0*REAL(abs(tempv(1,1,1)),kind(0d0))
END IF
! .. Kinetic Energy ..
DO k=decomp%xst(3),decomp%xen(3)
  DO j=decomp%xst(2),decomp%xen(2)
    DO i=decomp%xst(1),decomp%xen(1)
      tempu(i,j,k)=( abs(u(i,j,k)-uold(i,j,k))/dt )**2
    END DO
  END DO
END DO
CALL decomp_2d_fft_3d(tempu,tempv,DECOMP_2D_FFT_FORWARD)
IF(myid.eq.0) THEN
  enkin=0.5d0*REAL(abs(tempv(1,1,1)),kind(0d0))
END IF
! .. Potential Energy ..
DO k=decomp%xst(3),decomp%xen(3)
  DO j=decomp%xst(2),decomp%xen(2)
    DO i=decomp%xst(1),decomp%xen(1)
      tempu(i,j,k)=0.5d0*(abs((u(i,j,k)+uold(i,j,k))*0.50d0))**2&
        -0.125d0*Es*(abs(u(i,j,k))**4+abs(uold(i,j,k))**4)
    END DO
  END DO
END DO
CALL decomp_2d_fft_3d(tempu,tempv,DECOMP_2D_FFT_FORWARD)
IF(myid.eq.0) THEN
  enpot=REAL(abs(tempv(1,1,1)),kind(0d0))
  en=enpot+enkin+enstr
END IF
END SUBROUTINE enercalc
```

Listing 14.20: A Fortran subroutine to save final results after solving the 3D Klein-Gordon equation.

```





```

```
SUBROUTINE saveresults(Nt,plotgap,time,en,enstr,enkin,enpot)
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This subroutine saves the energy and times stored during the
! computation for the nonlinear Klein-Gordon equation
!
! INPUT
!
! .. Parameters ..
!  Nt        = number of timesteps
!  plotgap       = number of timesteps between plots
! .. Vectors ..
!  time        = times at which save data
!  en        = total energy
!  enstr       = strain energy
!  enpot       = potential energy
!  enkin       = kinetic energy
!
! OUTPUT
!
!
! LOCAL VARIABLES
!
! .. Scalars ..
!  n         = loop counter
! .. Arrays ..
!   name_config   = array to hold the filename
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!--------------------------------------------------------------------
! External routines required
!
! External libraries required

! Declare variables
IMPLICIT NONE
INTEGER(kind=4), INTENT(IN)             :: plotgap,Nt
REAL(KIND=8), DIMENSION(1:1+Nt/plotgap), INTENT(IN) :: enpot, enkin
REAL(KIND=8), DIMENSION(1:1+Nt/plotgap), INTENT(IN) :: en,enstr,time
INTEGER(kind=4)                   :: n
```

```
CHARACTER*100                    :: name_config

name_config = 'tdata.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO n=1,1+Nt/plotgap
  WRITE(11,*) time(n)
END DO
CLOSE(11)

name_config = 'en.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO n=1,1+Nt/plotgap
  WRITE(11,*) en(n)
END DO
CLOSE(11)

name_config = 'enkin.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO n=1,1+Nt/plotgap
  WRITE(11,*) enkin(n)
END DO
CLOSE(11)

name_config = 'enpot.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO n=1,1+Nt/plotgap
  WRITE(11,*) enpot(n)
END DO
CLOSE(11)

name_config = 'enstr.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO n=1,1+Nt/plotgap
  WRITE(11,*) enstr(n)
END DO
CLOSE(11)

END SUBROUTINE saveresults
```

Listing 14.21: A Fortran subroutine to read in the parameters to use when solving the 3D Klein-Gordon equation.

```
SUBROUTINE readinputfile(Nx,Ny,Nz,Nt,plotgap,Lx,Ly,Lz, &
               Es,DT,starttime,myid,ierr)
```

```
!
  ----------------------------------------------------------------------------
!
!
!  PURPOSE
!
! Read inputfile intialize parameters, which are stocked in the Input
  File
!
! .. INPUT ..
! Nx         = number of modes in the x direction
! Ny         = number of modes in the y direction
! Nz         = number of modes in the z direction
! Nt         = the number of timesteps
! plotgap      = the number of timesteps to take before plotting
! myid       = number of MPI process
! ierr       = MPI error output variable
! Lx         = size of the periodic domain of computation in x direction
! Ly         = size of the periodic domain of computation in y direction
! Lz         = size of the periodic domain of computation in z direction
! DT         = the time step
! starttime    = initial time of computation
! InputFileName = name of the Input File
! REFERENCES
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!
  -----------------------------------------------------------------------------
! EXTERNAL ROUTINES REQUIRED
IMPLICIT NONE
INCLUDE 'mpif.h'
! .. Scalar Arguments ..
INTEGER(KIND=4), INTENT(IN)    ::  myid
INTEGER(KIND=4), INTENT(OUT)   ::  Nx,Ny,Nz,Nt
INTEGER(KIND=4), INTENT(OUT)   ::  plotgap, ierr
REAL(KIND=8), INTENT(OUT)    ::  Lx, Ly, Lz, DT, starttime, Es
! .. Local scalars ..
INTEGER(KIND=4)          ::  stat
! .. Local Arrays ..
CHARACTER*40           ::  InputFileName
INTEGER(KIND=4), DIMENSION(1:5) ::  intcomm
REAL(KIND=8), DIMENSION(1:6)   ::  dpcomm

IF(myid.eq.0) THEN
```

```
    CALL GET_ENVIRONMENT_VARIABLE(NAME="inputfile",VALUE=InputFileName,
       STATUS=stat)
    IF(stat.NE.0) THEN
      PRINT*,"Set environment variable inputfile to the name of the"
      PRINT*,"file where the simulation parameters are set"
      STOP
    END IF
    OPEN(unit=11,FILE=trim(InputFileName),status="OLD")
    REWIND(11)
    READ(11,*) intcomm(1), intcomm(2), intcomm(3), intcomm(4), intcomm(5),
        &
        dpcomm(1), dpcomm(2), dpcomm(3), dpcomm(4), dpcomm(5), dpcomm(6)
    CLOSE(11)
    PRINT *,"NX ",intcomm(1)
    PRINT *,"NY ",intcomm(2)
    PRINT *,"NZ ",intcomm(3)
    PRINT *,"NT ",intcomm(4)
    PRINT *,"plotgap ",intcomm(5)
    PRINT *,"Lx ",dpcomm(1)
    PRINT *,"Ly ",dpcomm(2)
    PRINT *,"Lz ",dpcomm(3)
    PRINT *,"Es ",dpcomm(4)
    PRINT *,"Dt ",dpcomm(5)
    PRINT *,"strart time ",dpcomm(6)
    PRINT *,"Read inputfile"
  END IF
  CALL MPI_BCAST(dpcomm,6,MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD,ierr)
  CALL MPI_BCAST(intcomm,5,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

  Nx=intcomm(1)
  Ny=intcomm(2)
  Nz=intcomm(3)
  Nt=intcomm(4)
  plotgap=intcomm(5)
  Lx=dpcomm(1)
  Ly=dpcomm(2)
  Lz=dpcomm(3)
  Es=dpcomm(4)
  DT=dpcomm(5)
  starttime=dpcomm(6)

  END SUBROUTINE readinputfile
```

Listing 14.22: An example makefile for compiling the MPI program in listing 14.14.

```
# All settings here for use on FLUX, a cluster at the University of
    Michigan
# Center for Advanced Computing (CAC), using INTEL nehalem hardware,
# Need to load fftw module

```

```
  COMPILER =  mpif90
  decompdir=../2decomp_fft
  # compilation settings, optimization, precision, parallelization
  FLAGS =  -O0 -fltconsistency
  LIBS = -L${FFTW_LINK} -lfftw3

  DECOMPLIB = -I${decompdir}/include -L${decompdir}/lib -l2decomp_fft


# libraries
# source list for main program
SOURCES = KgSemiImp3d.f90 initialdata.f90 savedata.f90 getgrid.f90 \
      storeold.f90 saveresults.f90 enercalc.f90 readinputfile.f90

Kg: $(SOURCES)
    ${COMPILER} -o Kg $(FLAGS) $(SOURCES) $(LIBS) $(DECOMPLIB)


clean:
  rm -f *.o
clobber:
  rm -f Kg
```

Listing 14.23: A Fortran subroutine to create BOV (Brick of Values) header files after solving the 3D Klein-Gordon equation.

```
  PROGRAM BovCreate
  !
     --------------------------------------------------------------------------------

  ! .. Purpose ..
  !   BovCreate is a postprocessing program which creates header files for
      VisIt
  ! It uses the INPUTFILE and assumes that the filenames in the program
     are
  ! consistent with those in the current file.
  !
  ! .. PARAMETERS .. INITIALIZED IN INPUTFILE
  ! time     = start time of the simulation
  ! Nx       = power of two, number of modes in the x direction
  ! Ny       = power of two, number of modes in the y direction
  ! Nz       = power of two, number of modes in the z direction
  ! Nt       = the number of timesteps
  ! plotgap    = the number of timesteps to take before plotting
  ! Lx       = definition of the periodic domain of computation in x
     direction
  ! Ly       = definition of the periodic domain of computation in y
     direction
  ! Lz       = definition of the periodic domain of computation in z
     direction
```

```
! Es       = focusing or defocusing
! Dt       = the time step
!
! REFERENCES
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
!
   --------------------------------------------------------------------------------
!   EXTERNAL ROUTINES REQUIRED
IMPLICIT NONE
! .. Scalar Arguments ..
INTEGER(KIND=4)      ::  Nx, Ny, Nz, Nt, plotgap
REAL(KIND=8)       ::  Lx, Ly, Lz,  DT, time, Es
! .. Local scalars ..
INTEGER(KIND=4)      ::  stat,plotnum,ind,n,numplots
! .. Local Arrays ..
CHARACTER*50       ::  InputFileName, OutputFileName, OutputFileName2
CHARACTER*10       ::  number_file
InputFileName='INPUTFILE'
OPEN(unit=11,FILE=trim(InputFileName),status="OLD")
REWIND(11)
READ(11,*) Nx, Ny, Nz, Nt, plotgap, Lx, Ly, Lz, Es, DT, time
CLOSE(11)

plotnum=1
numplots=1+Nt/plotgap
DO n=1,numplots
  OutputFileName = 'data/u'
  ind = index(OutputFileName,' ') - 1
  WRITE(number_file,'(i0)') 10000000+plotnum
  OutputFileName = OutputFileName(1:ind)//number_file
  ind = index(OutputFileName,' ') - 1
  OutputFileName = OutputFileName(1:ind)//'.bov'
  OutputFileName2='u'
  ind = index(OutputFileName2,' ') - 1
  OutputFileName2 = OutputFileName2(1:ind)//number_file
  ind = index(OutputFileName2,' ') - 1
  OutputFileName2 = OutputFileName2(1:ind)//'.datbin'
  OPEN(unit=11,FILE=trim(OutputFileName),status="UNKNOWN")
  REWIND(11)
  WRITE(11,*) 'TIME: ',time
  WRITE(11,*) 'DATA_FILE: ',trim(OutputFileName2)
  WRITE(11,*) 'DATA_SIZE: ', Nx, Ny, Nz
  WRITE(11,*) 'DATA_FORMAT: DOUBLE'
  WRITE(11,*) 'VARIABLE: u'
  WRITE(11,*) 'DATA_ENDIAN: LITTLE'
```

```
    WRITE(11,*) 'CENTERING: ZONAL'
    WRITE(11,*) 'BRICK_ORIGIN:', -Nx/2, -Ny/2, -Nz/2
    WRITE(11,*) 'BRICK_SIZE:', Nx, Ny, Nz
    WRITE(11,*) 'DIVIDE_BRICK: true'
    WRITE(11,*) 'DATA_BRICKLETS:', Nx/2, Ny/2, Nz/2
    CLOSE(11)

    time=time+plotgap*DT
    plotnum=plotnum+1
  END DO
  END PROGRAM BovCreate
```

14.1.4 Exercises

1) Compare the accuracy of the implicit and semi-implicit time stepping schemes in eqs. (14.3) and (14.4). Which scheme produces the most accurate results in the least amount of real time?

2) Write serial Fortran programs to solve the two- and three-dimensional Klein-Gordon equations using the fully implicit time stepping scheme in eq. (14.4).

3) Write OpenMP parallel Fortran programs to solve the two- and three-dimensional Klein-Gordon equations using the fully implicit time stepping scheme in eq. (14.4).

4) The MPI command MPI_BCAST is used in the subroutine readinputfile, listed in list 14.21. Look up this command (possibly using one of the references listed in the introduction to programming section) and explain what it does.

5) Write an MPI parallel Fortran program to solve the two- and three-dimensional Klein-Gordon equations using the fully implicit time stepping scheme in eq. (14.4).

6) Compare the results of fully three-dimensional simulations with periodic boundary conditions ($\mathbb{T}^3$) with analytical predictions for blow up on the entire real space ($\mathbb{R}^3$) summarized in Donninger and Schlag [14].

7) Grenier [21, p. 18] explains that the linear Klein-Gordon equation can be written as two coupled Schrödinger equations. One can extend this formulation to the nonlinear Klein-Gordon equation. If we let

$$u = \phi + \xi \quad \text{and} \quad \frac{\partial u}{\partial t} = \phi - \xi \tag{14.6}$$

then the two coupled equations

$$i\frac{\partial}{\partial t}\begin{bmatrix}\phi\\ \xi\end{bmatrix} = \begin{bmatrix}-\Delta - 1 & -\Delta \\ \Delta & \Delta + 1\end{bmatrix}\begin{bmatrix}\phi\\ \xi\end{bmatrix} \pm \begin{bmatrix}1\\ -1\end{bmatrix}\frac{|\phi+\xi|^2(\phi+\xi)}{2} \tag{14.7}$$

are equivalent to the nonlinear Klein-Gordon equation

$$\frac{\partial^2 u}{\partial t^2} - \Delta u + u = \pm u^3. \tag{14.8}$$

a) Fill in the details to explain why eqs. (14.6) and (14.7) are equivalent to eq. (14.8). In particular show that by adding and subtracting the two equations in eqs. (14.6) and (14.7), we get

$$\begin{aligned} i\frac{\partial}{\partial t}\left(\phi+\xi\right) &= -\left(\phi-\xi\right) \\ i\frac{\partial}{\partial t}\left(\phi-\xi\right) &= -\Delta\left(\phi+\xi\right) - \left(\phi+\xi\right) \pm \left|\phi+\xi\right|^2\left(\phi+\xi\right). \end{aligned}$$

Differentiating the first of these equations and substituting it into the second, then recalling that we defined $u = \phi + \xi$ in eq. (14.6) gives us the Klein-Gordon equation in eq. (14.8).

b) Solve these two equations using either the implicit midpoint rule or the Crank Nicolson method.

Bibliography

[1] S.M. Allen, and J.W. Cahn, A microscopic theory for antiphase boundary motion and its applications to antiphase domain coarsening, Acta Metallurgica **27**, 1085-1095, (1979).

[2] G. Birkhoff, and G.–C., Rota, **Ordinary Differential Equations** (4th ed.), Wiley, (1989).

[3] S. Blanes, F. Casas, P. Chartier and A. Murua, Splitting methods with complex coefficients for some classes of evolution equations, Mathematics of Computation (forthcoming) `http://arxiv.org/abs/1102.1622`

[4] B. Bradie, **A Friendly Introduction to Numerical Analysis**, Pearson, (2006).

[5] G. Boffetta and R.E. Ecke, Two-Dimensional Turbulence, Annual Review of Fluid Mechanics **44**, 427-451, (2012).

[6] W.E. Boyce and R.C. DiPrima, **Elementary Differential Equations and Boundary Value Problems**, Wiley, (2010).

[7] J. P. Boyd, **Chebyshev and Fourier Spectral Methods**, Dover, (2001). `http://www-personal.umich.edu/~jpboyd/`

[8] C. Canuto, M.Y. Hussaini, A. Quarteroni and T.A. Zang, **Spectral Methods: Fundamentals in Single Domains**, Springer, (2006).

[9] C. Canuto, M.Y. Hussaini, A. Quarteroni and T.A. Zang, **Spectral Methods: Evolution to Complex Geometries and Applications to Fluid Dynamics**, Springer, (2007).

[10] C. Cichowlas and M.-E. Brachet, Evolution of complex singularities in Kida-Pelz and Taylor-Green inviscid flows, Fluid Dynamics Research **36**, 239-248, (2005).

[11] B. Cloutier, B.K. Muite and P. Rigge, Performance of FORTRAN and C GPU Extensions for a Benchmark Suite of Fourier Pseudospectral Algorithms Forthcoming Proceedings of the Symposium on Application Accelerators in High Performancs computing (2012) `http://arxiv.org/abs/1206.3215`

[12] J.W. Cooley and J.W. Tukey, An algorithm for the machine calculation of complex Fourier series, Mathematics of Computation **19**, 297-301, (1965).

[13] R. Courant and F. John, **Introduction to Calculus and Analysis I, II** Springer (1998,1999)

[14] R. Donninger and W. Schlag, Numerical study of the blowup/global existence dichotomy for the focusing cubic nonlinear Klein-Gordon equation, Nonlinearity **24**, 2547-2562, (2011).

[15] C.R. Doering and J.D. Gibbon, **Applied Analysis of the Navier-Stokes Equations**, Cambridge University Press, (1995).

[16] B. Eliasson and P. K. Shukla Nonlinear aspects of quantum plasma physics: Nanoplasmonics and nanostructures in dense plasmas Plasma and Fusion Research: Review Articles, **4**, 32 (2009).

[17] L.C. Evans, **Partial Differential Equations**, American Mathematical Society, (2010).

[18] B. Fornberg, A numerical study of 2-D turbulence, Journal of Computational Physics **25**, 1-31, (1977).

[19] G. Gallavotti, **Foundations of Fluid Dynamics**, Springer, (2002). `http://www.math.rutgers.edu/~giovanni/glib.html#E`

[20] D. Gottlieb and S.A. Orszag, **Numerical Analysis of Spectral Methods: Theory and Applications**, SIAM, (1977).

[21] W. Grenier, **Relativistic Quantum Mechanics**, Springer, (1994)

[22] W. Gropp, E. Lusk and A. Skjellum, **Using MPI**, MIT Press, (1999).

[23] W. Gropp, E. Lusk and R. Thakur, **Using MPI-2**, MIT Press, (1999).

[24] M.T. Heideman, D.H. Johnson and C.S. Burrus, Gauss and the History of the Fast Fourier Transform, IEEE ASSP Magazine **1**(4), 1421, (1984).

[25] J.S. Hesthaven, S. Gottlieb and D. Gottlieb, **Spectral Methods for Time-Dependent Problems**, Cambridge University Press, (2007).

[26] D. Hughes-Hallett, A.M. Gleason, D.E. Flath, P.F. Lock, D.O. Lomen, D. Lovelock, W.G. MacCallum, D. Mumford, B. G. Osgood, D. Quinney, K. Rhea, J. Tecosky-Feldman, T.W. Tucker, and O.K. Bretscher, A. Iovita, W. Raskind, S.P. Gordon, A. Pasquale, J.B. Thrash, **Calculus, Single and Multivariable**, 5th ed. Wiley, (2008)

[27] H. Holden, K.H. Karlsen, K.-A. Lie and N.H. Risebro, **Splitting Methods for Partial Differential Equations with Rough Solutions**, European Mathematical Society Publishing House, Zurich, (2010).

[28] H. Holden, K.H. Karlsen, N.H. Risebro and T. Tao, Operator splitting for the KdV equation, Mathematics of Computation **80**, 821-846, (2011).

[29] A. Iserles, **A First Course in the Numerical Analysis of Differential Equations**, Cambridge University Press, (2009).

[30] R. Johnstone, Improved Scaling for Direct Numerical Simulations of Turbulence, HECTOR distributed Computational Science and Engineering Report, `http://www.hector.ac.uk/cse/distributedcse/reports/ss3f-swt/`

[31] C. Klein, Fourth order time-stepping for low dispersion Korteweg-De Vries and nonlinear Schrödinger equations, Electronic Transactions on Numerical Analysis **29**, 116-135, (2008).

[32] C. Klein, B.K. Muite and K. Roidot, Numerical Study of Blowup in the Davey-Stewartson System, `http://arxiv.org/abs/1112.4043`

[33] C. Klein and K. Roidot, Fourth order time-stepping for Kadomstev-Petviashvili and Davey-Stewartson Equations, SIAM Journal on Scientific Computation **33**, 3333-3356, (2011).
`http://arxiv.org/abs/1108.3345`

[34] S. Laizet and E. Lamballais, High-order compact schemes for incompressible flows: A simple and efficient method with quasi-spectral accuracy, Journal of Computational Physics **228**, 5989-6015, (2009).

[35] S. Laizet and N. Li, Incompact3d: A powerful tool to tackle turbulence problems with up to $O(10^5)$ computational cores, International Journal of Numerical Methods in Fluids **67**, 1735-1757, (2011).

[36] R. H. Landau, **Quantum Mechanics II**, Wiley, (1996).

[37] P. Lax, S. Burstein and A. Lax, **Calculus with Applications and Computing, Vol. 1**, Springer, (1976).

[38] N. Li and S. Laizet, 2DECOMP&FFT - A highly scalable 2D decomposition library and FFT interface, Proc. Cray User Group 2010 Conference.
`http://www.2decomp.org/pdf/17B-CUG2010-paper-Ning_LI.pdf`

[39] E.H. Lieb and M. Loss, **Analysis**, American Mathematical Society, (2003).

[40] F. Linares and G. Ponce, **Introduction to Nonlinear Dispersive Equations**, Springer, (2009).

[41] J. Levesque and G. Wagenbreth, **High Performance Computing: Programming and Applications**, CRC Press, (2011).

[42] A.J. Majda and A.L. Bertozzi, **Vorticity and Incompressible Flow**, Cambridge University Press, (2002).

[43] R.I. McLachlan and G.R.W. Quispel, Splitting Methods, Acta Numerica **11**, 341-434, (2002).

[44] M. Metcalf, J. Reid and M. Cohen, **Modern Fortran Explained**, Oxford University Press, (2011).

[45] K. Nakanishi and W. Schlag, **Invariant Manifolds and Dispersive Hamiltonian Evolution Equations**, European Mathematical Society, (2011).

[46] P.J. Olver, Dispersive Quantization, American Mathematical Monthly, **117**, 599-610, (2010).

[47] P.J. Olver and C. Shakiban, **Applied Linear Algebra**, Prentice Hall, (2006).

[48] S.A. Orszag and G.S. Patterson Jr., Numerical simulation of three-dimensional homogeneous isotropic turbulence, Physical Review Letters **28(2)**, 76-79, (1972).

[49] R. Peyret, **Spectral Methods for Incompressible Viscous Flow**, Springer, (2002).

[50] R. Renardy and R.C. Rogers, **An Introduction to Partial Differential Equations**, Springer, (2004).

[51] A. Shapiro The use of an exact solution of the Navier-Stokes equations in a validation test of a three-dimensional nonhydrostatic numerical model, Monthly Weather Review **121**, 2420-2425, (1993).

[52] J. Shen, T. Tang and L.-L. Wang, **Spectral Methods: Algorithms, Analysis and Applications**, Springer, (2011).

[53] C. Sulem and P.L. Sulem, **The Nonlinear Schrödinger equation: Self-Focusing and Wave Collapse**, Springer, (1999).

[54] R. Temam, **Navier-Stokes Equations**, Third revised edition, AMS, (2001).

[55] M. Thalhammer, Time-Splitting Spectral Methods for Nonlinear Schrödinger Equations, Unpublished manuscript, (2008).
`http://techmath.uibk.ac.at/mecht/research/SpringSchool/manuscript_Thalhammer.pdf`

[56] L. N. Trefethen, **Spectral Methods in Matlab**, SIAM, (2000).

[57] L. N. Trefethen and K. Embree (Ed.), **The (Unfninished) PDE coffee table book.** Unpublished notes available online
`http://people.maths.ox.ac.uk/trefethen/pdectb.html`

[58] D.J. Tritton, **Physical Fluid Dynamics**, Clarendon Press, (1988).

[59] H. Uecker, A short ad hoc introduction to spectral for parabolic PDE and the Navier-Stokes equations, Lecture notes from the 2009 International Summer School on Modern Computational Science
`http://www.staff.uni-oldenburg.de/hannes.uecker/hfweb-e.html`

[60] J.A.C. Weideman and B.M. Herbst, Split-step methods for the solution of the nonlinear Schrödinger equation, SIAM Journal on Numerical Analysis **23(3)**, 485-507, (1986).

[61] J. Yang, **Nonlinear Waves in Integrable and Nonintegrable Systems**, SIAM, (2010).

[62] L. Yang, Numerical studies of the Klein-Gordon-Schrödinger equations, Masters Thesis, National University of Singapore
`http://scholarbank.nus.edu.sg/handle/10635/15515`

Appendix A

GPU programs for Fourier pseudospectral simulations of the Navier-Stokes, Cubic Nonlinear Schrödinger and sine Gordon equations

This section includes the programs taken from a conference paper by Cloutier, Muite and Rigge [11]. The main purpose is to give example programs which show how to use graphics processing units (GPUs) to solve partial differential equations using Fourier methods. For further background on GPUs and programming models for GPUs see Cloutier, Muite and Rigge [11]. It should be noted that the algorithms used for the sine Gordon equation are very similar to those for the Klein Gordon equation discussed elsewhere in this tutorial. For consistency with the rest of the tutorial, only programs using CUDA Fortran and OpenACC extensions to Fortran are included although Cloutier, Muite and Rigge [11] also has CUDA C programs. GPUs enable acceleration of Fourier pseudospectral codes by factors of 10 compared to OpenMP parallelizations on a single 8 core node.

A.1 2D Navier Stokes Equations

These programs use the Crank-Nicolson method.

Listing A.1: A CUDA Fortran program to solve the 2D Navier-Stokes equations.

```




!-------------------------------------------------------------------
```

```
!
! PURPOSE
!
! This program numerically solves the 2D incompressible Navier-Stokes
! on a Square Domain [0,1]x[0,1] using pseudo-spectral methods and
! Crank-Nicolson timestepping. The numerical solution is compared to
! the exact Taylor-Green Vortex Solution.
!
! AUTHORS
!
! B. Cloutier, B.K. Muite, P. Rigge
! 4 June 2012
!
! Periodic free-slip boundary conditions and Initial conditions:
! u(x,y,0)=sin(2*pi*x)cos(2*pi*y)
! v(x,y,0)=-cos(2*pi*x)sin(2*pi*y)
! Analytical Solution (subscript denote derivatives):
! u(x,y,t)=sin(2*pi*x)cos(2*pi*y)exp(-8*pi^2*t/Re)
! v(x,y,t)=-cos(2*pi*x)sin(2*pi*y)exp(-8*pi^2*t/Re)
!   u_y(x,y,t)=-2*pi*sin(2*pi*x)sin(2*pi*y)exp(-8*pi^2*t/Re)
! v_x(x,y,t)=2*pi*sin(2*pi*x)sin(2*pi*y)exp(-8*pi^2*t/Re)
! omega=v_x-u_y
!
! .. Parameters ..
!  Nx       = number of modes in x - power of 2 for FFT
!  Ny       = number of modes in y - power of 2 for FFT
!  nplots      = number of plots produced
!  plotgap       = number of timesteps inbetween plots
!  Re          = dimensionless Renold's number
!  ReInv       = 1/Re for optimization
!  dt       = timestep size
!  dtInv       = 1/dt for optimization
!  tol         = determines when convergences is reached
!  numthreads    = number of CPUs used in calculation
! .. Scalars ..
!  i        = loop counter in x direction
!  j        = loop counter in y direction
!  n        = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  time        = times at which data is saved
!  chg         = error at each iteration
! .. Arrays (gpu) ..
!  omeg_d        = vorticity in real space
!  omeghat_d       = 2D Fourier transform of vorticity
!           at next iterate
!  omegoldhat_d    = 2D Fourier transform of vorticity at previous
!           iterate
!  nloldhat_d     = nonlinear term in Fourier space
!           at previous iterate
!  psihat_d      = 2D Fourier transform of streamfunction
!           at next iteration
!  temp1_d/temp2_d/temp3_d    = reusable complex/real space used for
!                calculations. This reduces number of
!                arrays stored.
! .. Vectors (gpu) ..
!  kx_d       = fourier frequencies in x direction
!  ky_d       = fourier frequencies in y direction
!  x_d        = x locations
!  y_d        = y locations
!  name_config    = array to store filename for data to be saved
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! This program has not been fully optimized.
!----------------------------------------------------------------------
module precision
! Precision control

integer, parameter, public :: Single = kind(0.0) ! Single precision
integer, parameter, public :: Double = kind(0.0d0) ! Double precision

integer, parameter, public :: fp_kind = Double
!integer, parameter, public :: fp_kind = Single

end module precision

module cufft

integer, public :: CUFFT_FORWARD = -1
integer, public :: CUFFT_INVERSE = 1
integer, public :: CUFFT_R2C = Z'2a' ! Real to Complex (interleaved)
integer, public :: CUFFT_C2R = Z'2c' ! Complex (interleaved) to Real
integer, public :: CUFFT_C2C = Z'29' ! Complex to Complex, interleaved
integer, public :: CUFFT_D2Z = Z'6a' ! Double to Double-Complex
integer, public :: CUFFT_Z2D = Z'6c' ! Double-Complex to Double
integer, public :: CUFFT_Z2Z = Z'69' ! Double-Complex to Double-Complex

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! cufftPlan2d(cufftHandle *plan, int nx,int ny, cufftType type)
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

interface cufftPlan2d
subroutine cufftPlan2d(plan, nx, ny, type) bind(C,name='cufftPlan2d')
use iso_c_binding
```

```
integer(c_int):: plan
integer(c_int),value:: nx, ny, type
end subroutine cufftPlan2d
end interface cufftPlan2d

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! cufftDestroy(cufftHandle plan)
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

interface cufftDestroy
subroutine cufftDestroy(plan) bind(C,name='cufftDestroy')
use iso_c_binding
integer(c_int),value:: plan
end subroutine cufftDestroy
end interface cufftDestroy

interface cufftExecD2Z
  subroutine cufftExecD2Z(plan, idata, odata) &
        & bind(C,name='cufftExecD2Z')
     use iso_c_binding
     use precision
     integer(c_int),  value  :: plan
     real(fp_kind),   device :: idata(1:nx,1:ny)
     complex(fp_kind),device :: odata(1:nx,1:ny)
  end subroutine cufftExecD2Z
end interface cufftExecD2Z

interface cufftExecZ2D
  subroutine cufftExecZ2D(plan, idata, odata) &
        & bind(C,name='cufftExecZ2D')
     use iso_c_binding
     use precision
     integer(c_int),value:: plan
     complex(fp_kind),device:: idata(1:nx,1:ny)
     real(fp_kind),device :: odata(1:nx,1:ny)
  end subroutine cufftExecZ2D
end interface cufftExecZ2D

end module cufft

PROGRAM main
use precision
use cufft
! declare variables
  IMPLICIT NONE
  INTEGER(kind=4), PARAMETER    ::  Nx=4096
INTEGER(kind=4), PARAMETER    ::  Ny=4096
INTEGER(kind=8)          ::  temp=10000000
REAL(fp_kind), PARAMETER    ::  dt=0.000125d0 !dt=0.000002d0
```

```fortran
REAL(fp_kind), PARAMETER    ::  dtInv=1.0d0/REAL(dt,kind(0d0))
REAL(fp_kind), PARAMETER  &
  ::  pi=3.14159265358979323846264338327950288419716939937510d0
REAL(fp_kind), PARAMETER    ::  Re=1.0d0
REAL(fp_kind), PARAMETER    ::  ReInv=1.0d0/REAL(Re,kind(0d0))
REAL(fp_kind), PARAMETER    ::  tol=0.1d0**10
REAL(fp_kind)          ::  scalemodes,chg
INTEGER(kind=4), PARAMETER    ::  nplots=1,plotgap=20
REAL(fp_kind),DIMENSION(:), ALLOCATABLE   ::  x,y
REAL(fp_kind),DIMENSION(:,:), ALLOCATABLE ::  omeg,omegexact
INTEGER(kind=4)                ::  i,j,n,t, AllocateStatus
INTEGER(kind=4)                ::  planz2d,pland2z, kersize
!variables used for saving data and timing
INTEGER(kind=4)                ::  start, finish, count_rate,count, iol
CHARACTER*100                ::  name_config
! Declare variables for GPU
REAL(fp_kind), DEVICE, DIMENSION(:,:), ALLOCATABLE    ::  omeg_d,nl_d,
   temp2_d,&
                               temp3_d
COMPLEX(fp_kind), DEVICE, DIMENSION(:,:), ALLOCATABLE :: omegoldhat_d,
   nloldhat_d,&
                               omeghat_d, nlhat_d, psihat_d,&
                               temp1_d
COMPLEX(fp_kind), DEVICE, DIMENSION(:), ALLOCATABLE   ::  kx_d,ky_d
REAL(kind=8),DEVICE, DIMENSION(:), ALLOCATABLE      ::  x_d,y_d

kersize=min(Nx,256)
PRINT *,'Program starting'
    PRINT *,'Grid:',Nx,'X',Ny
PRINT *,'dt:',dt
ALLOCATE(x(1:Nx),y(1:Ny),omeg(1:Nx,1:Ny),omegexact(1:Nx,1:Ny),&
    stat=AllocateStatus)
IF (AllocateStatus .ne. 0) STOP
PRINT *,'Allocated CPU arrays'
ALLOCATE(kx_d(1:Nx/2+1),ky_d(1:Ny),x_d(1:Nx),y_d(1:Ny),omeg_d(1:Nx,1:Ny)
   ,&
    omegoldhat_d(1:Nx/2+1,1:Ny),nloldhat_d(1:Nx/2+1,1:Ny),&
    omeghat_d(1:Nx/2+1,1:Ny),nl_d(1:Nx,1:Ny),&
    nlhat_d(1:Nx/2+1,1:Ny),psihat_d(1:Nx/2+1,1:Ny),temp1_d(1:Nx/2+1,1:Ny
       ),&
    temp2_d(1:Nx,1:Ny),temp3_d(1:Nx,1:Ny),stat=AllocateStatus)
IF (AllocateStatus .ne. 0) STOP
PRINT *,'Allocated GPU arrays'
CALL cufftPlan2D(pland2z,nx,ny,CUFFT_D2Z)
CALL cufftPlan2D(planz2d,nx,ny,CUFFT_Z2D)
PRINT *,'Setup FFTs'

! setup fourier frequencies
!$cuf kernel do <<< *,* >>>
DO i=1,Nx/2+1
  kx_d(i)= 2.0d0*pi*cmplx(0.0d0,1.0d0)*REAL(i-1,kind=fp_kind)
```

```
END DO
kx_d(1+Nx/2)=0.0d0
!$cuf kernel do <<< *,* >>>
DO i=1,Nx
  x_d(i)=REAL(i-1,kind(0d0))/REAL(Nx,kind=fp_kind)
END DO
!$cuf kernel do <<< *,* >>>
DO j=1,Ny/2+1
  ky_d(j)= 2.0d0*pi*cmplx(0.0d0,1.0d0)*REAL(j-1,kind=fp_kind)
END DO
ky_d(1+Ny/2)=0.0d0
  !$cuf kernel do <<< *,* >>>
DO j = 1,Ny/2 -1
  ky_d(j+1+Ny/2)=-ky_d(1-j+Ny/2)
END DO
!$cuf kernel do <<< *, * >>>
DO j=1,Ny
  y_d(j)=REAL(j-1,kind(0d0))/REAL(Ny,kind=fp_kind)
END DO
scalemodes=1.0d0/REAL(Nx*Ny,kind=fp_kind)
PRINT *,'Setup grid and fourier frequencies'

!$cuf kernel do <<<  *,*  >>>
DO j=1,Ny
  DO i=1,Nx
    omeg_d(i,j)=4.0d0*pi*sin(2.0d0*pi*x_d(i))*sin(2.0d0*pi*y_d(j))!+0.01
        d0*cos(2.0d0*pi*y_d(j))
  END DO
END DO
CALL cufftExecD2Z(pland2z,omeg_d,omeghat_d)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!get initial nonlinear term using omeghat to find psihat, u, and v!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!$cuf kernel do <<<  *,*  >>>
DO j=1,Ny
  DO i=1,Nx/2+1
    psihat_d(i,j)=-omeghat_d(i,j)/(kx_d(i)*kx_d(i)+ky_d(j)*ky_d(j)+0.10
        d0**14)
  END DO
END DO

!$cuf kernel do <<<  *,*  >>>
DO j=1,Ny
  DO i=1,Nx/2+1
    temp1_d(i,j)=psihat_d(i,j)*ky_d(j)*scalemodes
  END DO
END DO
CALL cufftExecZ2D(planz2d,temp1_d,temp3_d) !u

!$cuf kernel do <<<  *,*  >>>
```

```
DO j=1,Ny
  DO i=1,Nx/2+1
    temp1_d(i,j)=omeghat_d(i,j)*kx_d(i)
  END DO
END DO
CALL cufftExecZ2D(planz2d,temp1_d,temp2_d) !omega_x

!$cuf kernel do <<< *,* >>>
DO j=1,Ny
  DO i=1,Nx
    nl_d(i,j)=temp3_d(i,j)*temp2_d(i,j)
  END DO
END DO

!$cuf kernel do <<< *,* >>>
DO j=1,Ny
  DO i=1,Nx/2+1
    temp1_d(i,j)=-psihat_d(i,j)*kx_d(i)*scalemodes
  END DO
END DO
CALL cufftExecZ2D(planz2d,temp1_d,temp3_d) !v

!$cuf kernel do <<< *,* >>>
DO j=1,Ny
  DO i=1,Nx/2+1
    temp1_d(i,j)=omeghat_d(i,j)*ky_d(j)
  END DO
END DO
CALL cufftExecZ2D(planz2d,temp1_d,temp2_d) !omega_y

!combine to get full nonlinear term in real space
!$cuf kernel do <<< *,* >>>
DO j=1,Ny
  DO i=1,Nx
    nl_d(i,j)=(nl_d(i,j)+temp3_d(i,j)*temp2_d(i,j))*scalemodes
  END DO
END DO
CALL cufftExecD2Z(pland2z,nl_d,nlhat_d)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

temp2_d=omeg_d !omegacheck
PRINT *,'Got initial data, starting timestepping'
CALL system_clock(start,count_rate)
DO t=1,nplots
  DO n=1,plotgap
    chg=1.0d0
    nloldhat_d=nlhat_d
    omegoldhat_d=omeghat_d
    DO WHILE (chg>tol)
        !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
      DO j=1,Ny
      DO i=1,Nx/2+1
        omeghat_d(i,j)=((dtInv+0.5d0*ReInv*(kx_d(i)*kx_d(i)+ky_d(j)*
            ky_d(j)))&
             *omegoldhat_d(i,j) - 0.5d0*(nloldhat_d(i,j)+nlhat_d(i,j)))
                &
             /(dtInv-0.5d0*ReInv*(kx_d(i)*kx_d(i)+ky_d(j)*ky_d(j)))
      END DO
    END DO
      !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
    DO j=1,Ny
      DO i=1,Nx/2+1
        psihat_d(i,j)=-omeghat_d(i,j)/(kx_d(i)*kx_d(i)+ky_d(j)*ky_d(j)
            +0.10d0**14)
      END DO
    END DO
    CALL cufftExecZ2D(planz2d,omeghat_d,omeg_d)

    !check for convergence
    chg=0.0d0
      !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
    DO j=1,Ny
      DO i=1,Nx
        chg=chg+(omeg_d(i,j)-temp2_d(i,j))*(omeg_d(i,j)-temp2_d(i,j))&
        *scalemodes*scalemodes
      END DO
    END DO

    !!!!!!!!!!!!!!!!
    !nonlinear term!
    !!!!!!!!!!!!!!!!
    !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
    DO j=1,Ny
      DO i=1,Nx/2+1
        temp1_d(i,j)=psihat_d(i,j)*ky_d(j)*scalemodes
      END DO
    END DO
    CALL cufftExecZ2D(planz2d,temp1_d,temp3_d) !u

    !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
    DO j=1,Ny
      DO i=1,Nx/2+1
        temp1_d(i,j)=omeghat_d(i,j)*kx_d(i)
      END DO
    END DO
    CALL cufftExecZ2D(planz2d,temp1_d,temp2_d) !omega_x

    !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
    DO j=1,Ny
      DO i=1,Nx
        nl_d(i,j)=temp3_d(i,j)*temp2_d(i,j)
      END DO
      END DO

      !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
      DO j=1,Ny
        DO i=1,Nx/2+1
          temp1_d(i,j)=-psihat_d(i,j)*kx_d(i)*scalemodes
        END DO
      END DO
      CALL cufftExecZ2D(planz2d,temp1_d,temp3_d) !v

      !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
      DO j=1,Ny
        DO i=1,Nx/2+1
          temp1_d(i,j)=omeghat_d(i,j)*ky_d(j)
        END DO
      END DO
      CALL cufftExecZ2D(planz2d,temp1_d,temp2_d) !omega_y

      !combine to get full nonlinear term in real space
      !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
      DO j=1,Ny
        DO i=1,Nx
          nl_d(i,j)=(nl_d(i,j)+temp3_d(i,j)*temp2_d(i,j))*scalemodes
        END DO
      END DO
      CALL cufftExecD2Z(pland2z,nl_d,nlhat_d)
      !!!!!!!!!!!!!!!!

      temp2_d=omeg_d !save omegacheck
    END DO
  END DO
  !PRINT *, t*plotgap*dt
END DO
CALL system_clock(finish,count_rate)
PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),&
    'for Time stepping'

! Copy grid back to host
x=x_d
y=y_d
omeg=omeg_d

!get exact omega
DO j=1,Ny
  DO i=1,Nx
    omegexact(i,j)=4.0d0*pi*sin(2.0d0*pi*x(i))*&
      sin(2.0d0*pi*y(j))*exp(-8.0d0*ReInv*pi**2*nplots*plotgap*dt)
  END DO
END DO
!compute max error
PRINT *,'Max Error:',maxval(abs(omeg*scalemodes-omegexact))
```

```fortran

temp=temp+1
write(name_config,'(a,i0,a)') 'omega',temp,'.datbin'
INQUIRE(iolength=iol) omeg(1,1)
OPEN(unit=11,FILE=name_config,form="unformatted", access="direct",recl=
   iol)
count = 1
DO j=1,Ny
  DO i=1,Nx
    WRITE(11,rec=count) omeg(i,j)*scalemodes
    count=count+1
  END DO
END DO
CLOSE(11)

CALL cufftDestroy(planz2d)
CALL cufftDestroy(pland2z)
PRINT *,'Destroyed fft plan'

DEALLOCATE(x,y,omeg,omegexact,stat=AllocateStatus)
IF (AllocateStatus .ne. 0) STOP
PRINT *,'Deallocated CPU memory'

DEALLOCATE(kx_d,ky_d,x_d,y_d,&
    omeg_d,omegoldhat_d, nloldhat_d,omeghat_d,&
    nl_d, nlhat_d,temp1_d,temp2_d,temp3_d,&
    psihat_d,stat=AllocateStatus)
IF (AllocateStatus .ne. 0) STOP
PRINT *,'Deallocated GPU memory'
PRINT *,'Program execution complete'
END PROGRAM main
```

Listing A.2: An OpenACC Fortran program to solve the 2D Navier-Stokes equations.

```fortran

!--------------------------------------------------------------------
!
! PURPOSE
!
! This program numerically solves the 2D incompressible Navier-Stokes
! on a Square Domain [0,1]x[0,1] using pseudo-spectral methods and
! Crank-Nicolson timestepping. The numerical solution is compared to
! the exact Taylor-Green Vortex Solution.
!
! AUTHORS
!
! B. Cloutier, B.K. Muite, P. Rigge
! 4 June 2012
!
! Periodic free-slip boundary conditions and Initial conditions:
```

```
! u(x,y,0)=sin(2*pi*x)cos(2*pi*y)
! v(x,y,0)=-cos(2*pi*x)sin(2*pi*y)
! Analytical Solution (subscript denote derivatives):
! u(x,y,t)=sin(2*pi*x)cos(2*pi*y)exp(-8*pi^2*t/Re)
! v(x,y,t)=-cos(2*pi*x)sin(2*pi*y)exp(-8*pi^2*t/Re)
!   u_y(x,y,t)=-2*pi*sin(2*pi*x)sin(2*pi*y)exp(-8*pi^2*t/Re)
! v_x(x,y,t)=2*pi*sin(2*pi*x)sin(2*pi*y)exp(-8*pi^2*t/Re)
! omega=v_x-u_y
!
! .. Parameters ..
!  Nx       = number of modes in x - power of 2 for FFT
!  Ny       = number of modes in y - power of 2 for FFT
!  nplots     = number of plots produced
!  plotgap      = number of timesteps inbetween plots
!  Re         = dimensionless Renold's number
!  ReInv      = 1/Re for optimization
!  dt       = timestep size
!  dtInv      = 1/dt for optimization
!  tol        = determines when convergences is reached
!  scalemodes   = 1/(Nx*Ny), scaling after preforming FFTs
! .. Scalars ..
!  i        = loop counter in x direction
!  j        = loop counter in y direction
!  n        = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  time       = times at which data is saved
!  chg        = error at each iteration
! .. Arrays ..
!  omeg       = vorticity in real space
!  omeghat      = 2D Fourier transform of vorticity
!           at next iterate
!  omegoldhat   = 2D Fourier transform of vorticity at previous
!           iterate
!  nl       = nonlinear term
!  nlhat      = nonlinear term in Fourier space
!  nloldhat     = nonlinear term in Fourier space
!           at previous iterate
!  omegexact    = taylor-green vorticity at
!           at final step
!  psihat     = 2D Fourier transform of streamfunction
!           at next iteration
!  temp1/temp2/temp3= reusable complex/real space used for
!           calculations. This reduces number of
!           arrays stored.
! .. Vectors ..
!  kx       = fourier frequencies in x direction
!  ky       = fourier frequencies in y direction
!  x        = x locations
!  y        = y locations
!  name_config    = array to store filename for data to be saved
! REFERENCES
```

```
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External libraries required
!       Cuda FFT
!       OpenACC
!       FFTW3           -- Fastest Fourier Transform in the West
!                       (http://www.fftw.org/)
!       OpenMP

module precision
! Precision control

integer, parameter, public :: Single = kind(0.0) ! Single precision
integer, parameter, public :: Double = kind(0.0d0) ! Double precision

integer, parameter, public :: fp_kind = Double
!integer, parameter, public :: fp_kind = Single

end module precision

module cufft

integer, public :: CUFFT_FORWARD = -1
integer, public :: CUFFT_INVERSE = 1
integer, public :: CUFFT_R2C = Z'2a' ! Real to Complex (interleaved)
integer, public :: CUFFT_C2R = Z'2c' ! Complex (interleaved) to Real
integer, public :: CUFFT_C2C = Z'29' ! Complex to Complex, interleaved
integer, public :: CUFFT_D2Z = Z'6a' ! Double to Double-Complex
integer, public :: CUFFT_Z2D = Z'6c' ! Double-Complex to Double
integer, public :: CUFFT_Z2Z = Z'69' ! Double-Complex to Double-Complex

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! cufftPlan2d(cufftHandle *plan, int nx,int ny, cufftType type)
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

interface cufftPlan2d
subroutine cufftPlan2d(plan, nx, ny, type) bind(C,name='cufftPlan2d')
use iso_c_binding
integer(c_int):: plan
integer(c_int),value:: nx, ny, type
```

```
end subroutine cufftPlan2d
end interface cufftPlan2d

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! cufftDestroy(cufftHandle plan)
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

interface cufftDestroy
subroutine cufftDestroy(plan) bind(C,name='cufftDestroy')
use iso_c_binding
integer(c_int),value:: plan
end subroutine cufftDestroy
end interface cufftDestroy

interface cufftExecD2Z
  subroutine cufftExecD2Z(plan, idata, odata) &
        & bind(C,name='cufftExecD2Z')
     use iso_c_binding
     use precision
     integer(c_int),  value   :: plan
     real(fp_kind),   device  :: idata(1:nx,1:ny)
     complex(fp_kind),device  :: odata(1:nx/2+1,1:ny)
  end subroutine cufftExecD2Z
end interface cufftExecD2Z

interface cufftExecZ2D
  subroutine cufftExecZ2D(plan, idata, odata) &
        & bind(C,name='cufftExecZ2D')
     use iso_c_binding
     use precision
     integer(c_int),value:: plan
     complex(fp_kind),device:: idata(1:nx/2+1,1:ny)
     real(fp_kind),device :: odata(1:nx,1:ny)
  end subroutine cufftExecZ2D
end interface cufftExecZ2D
end module cufft


PROGRAM main
USE precision
USE cufft
USE openacc

IMPLICIT NONE
  INTEGER(kind=4), PARAMETER      ::  Nx=512
INTEGER(kind=4), PARAMETER      ::  Ny=512
REAL(kind=8), PARAMETER     ::  dt=0.000125d0
REAL(kind=8), PARAMETER     ::  dtInv=1.0d0/REAL(dt,kind(0d0))
REAL(kind=8), PARAMETER &
```

```
  ::  pi=3.14159265358979323846264338327950288419716939937510d0
REAL(kind=8), PARAMETER     ::  Re=1.0d0
REAL(kind=8), PARAMETER     ::  ReInv=1.0d0/REAL(Re,kind(0d0))
REAL(kind=8), PARAMETER     ::  tol=0.1d0**10
REAL(kind=8)          ::  scalemodes
REAL(kind=8)          ::  chg
INTEGER(kind=4), PARAMETER    ::  nplots=1, plotgap=20
COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE    ::  kx,ky
REAL(kind=8), DIMENSION(:), ALLOCATABLE     ::  x,y,time
REAL(kind=8), DIMENSION(:,:), ALLOCATABLE   :: omeg,nl, temp2, temp3,
   omegexact
COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE  :: omegoldhat, nloldhat,&
                           omeghat,nlhat, psihat,temp1
INTEGER(kind=4)              ::  i,j,n,t, allocatestatus
INTEGER(kind=4)              ::  pland2z,planz2d
INTEGER(kind=4)              ::  count, iol
CHARACTER*100               ::  name_config
INTEGER(kind=4)              ::  start, finish, count_rate
INTEGER(kind=4)              ::  ierr,vecsize,gangsize
INTEGER(kind=8)              ::  planfxy,planbxy

vecsize=32
gangsize=16
  PRINT *,'Grid:',Nx,'X',Ny
PRINT *,'dt:',dt
ALLOCATE(time(1:nplots+1),kx(1:Nx),ky(1:Ny),x(1:Nx),y(1:Ny),&
    omeg(1:Nx,1:Ny),omegoldhat(1:Nx/2+1,1:Ny),&
    nloldhat(1:Nx/2+1,1:Ny),temp3(1:Nx,1:Ny),omeghat(1:Nx/2+1,1:Ny),&
    nl(1:Nx,1:Ny),nlhat(1:Nx/2+1,1:Ny), psihat(1:Nx/2+1,1:Ny),&
    temp1(1:Nx/2+1,1:Ny),omegexact(1:Nx,1:Ny),temp2(1:Nx,1:Ny),&
    stat=AllocateStatus)
IF (AllocateStatus .ne. 0) STOP
PRINT *,'allocated space'

CALL cufftPlan2D(pland2z,nx,ny,CUFFT_D2Z)
CALL cufftPlan2D(planz2d,nx,ny,CUFFT_Z2D)

PRINT *,'Setup 2D FFTs'

! setup fourier frequencies in x-direction
!$acc data copy(kx,ky,x,y,time,temp3,omeg,nl,temp1,temp2,omegoldhat,
   nloldhat,omeghat,nlhat,psihat)
PRINT *, 'Copied arrays over to device'
!$acc kernels loop
DO i=1,Nx/2+1
  kx(i)= 2.0d0*pi*cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))
END DO
!$acc end kernels
kx(1+Nx/2)=0.0d0
!$acc kernels loop
DO i = 1,Nx/2 -1
```

```
      kx(i+1+Nx/2)=-kx(1-i+Nx/2)
    END DO
    !$acc end kernels
    !$acc kernels loop
    DO i=1,Nx
      x(i)=REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0))
    END DO
    !$acc end kernels
    ! setup fourier frequencies in y-direction
    !$acc kernels loop
    DO j=1,Ny/2+1
      ky(j)= 2.0d0*pi*cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))
    END DO
    !$acc end kernels
    ky(1+Ny/2)=0.0d0
    !$acc kernels loop
    DO j = 1,Ny/2 -1
      ky(j+1+Ny/2)=-ky(1-j+Ny/2)
    END DO
    !$acc end kernels
    !$acc kernels loop
    DO j=1,Ny
      y(j)=REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0))
    END DO
    !$acc end kernels
    scalemodes=1.0d0/REAL(Nx*Ny,kind(0d0))
    PRINT *,'Setup grid and fourier frequencies'

    !initial data
    !$acc kernels loop
    DO j=1,Ny
      DO i=1,NX
        omeg(i,j)=4.0d0*pi*sin(2.0d0*pi*x(i))*sin(2.0d0*pi*y(j))!+0.01d0*cos
            (2.0d0*pi*y(j))
      END DO
    END DO
    !$acc end kernels
    !\hat{\omega^{n,k}}
    CALL cufftExecD2Z(pland2z,omeg,omeghat)

    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !get initial nonlinear term using omeghat, psihat, u, and v!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !\hat{\psi^{n+1,k+1}}
    !$acc kernels loop gang(gangsize), vector(vecsize)
    DO j=1,Ny
      DO i=1,Nx/2+1
        psihat(i,j)=-omeghat(i,j)/(kx(i)*kx(i)+ky(j)*ky(j) + 0.1d0**14)
      END DO
    END DO
    !$acc end kernels
```

```
!\omega^{n+1,k+1}
CALL cufftExecZ2D(planz2d,omeghat,omeg)

!get \hat{\psi_y^{n+1,k+1}} used to get u, NOTE: u=\psi_y
!$acc kernels loop gang(gangsize), vector(vecsize)
DO j=1,Ny
  DO i=1,Nx/2+1
    temp1(i,j)=psihat(i,j)*ky(j)*scalemodes
  END DO
END DO
!$acc end kernels
CALL cufftExecZ2D(planz2d,temp1,temp3) !u

! \hat{\omega_x^{n,k}}
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx/2+1
    temp1(i,j)=omeghat(i,j)*kx(i)
  END DO
END DO
!$acc end kernels
! \omega_x^{n,k}
CALL cufftExecZ2D(planz2d,temp1,temp2)

! first part nonlinear term in real space
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    nl(i,j)=temp3(i,j)*temp2(i,j)
  END DO
END DO
!$acc end kernels

!get \hat{\psi_x^{n+1,k+1}} used to get v, NOTE: v=-\psi_x
!$acc kernels loop gang(gangsize), vector(vecsize)
DO j=1,Ny
  DO i=1,Nx/2+1
    temp1(i,j)=-psihat(i,j)*kx(i)*scalemodes
  END DO
END DO
!$acc end kernels
CALL cufftExecZ2D(planz2d,temp1,temp3) !v

! \hat{\omega_y^{n,k}}
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx/2+1
    temp1(i,j)=omeghat(i,j)*ky(j)
  END DO
END DO
!$acc end kernels
```

```
! \omega_y^{n,k}
CALL cufftExecZ2D(planz2d,temp1,temp2)

! get the rest of nonlinear term in real space
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    nl(i,j)=(nl(i,j)+temp3(i,j)*temp2(i,j))*scalemodes
  END DO
END DO
!$acc end kernels
! transform nonlinear term into fourier space
CALL cufftExecD2Z(pland2z,nl,nlhat)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    temp2(i,j)=omeg(i,j)
  END DO
END DO
!$acc end kernels

PRINT *,'Got initial data, starting timestepping'
time(1)=0.0d0
CALL system_clock(start,count_rate)
DO t=1,nplots
  DO n=1,plotgap
    chg=1.0d0
    ! save old values(__^{n,k} terms in equation)
    !$acc kernels loop gang(gangsize), vector(vecsize)
    DO j=1,Ny
      DO i=1,Nx/2+1
        nloldhat(i,j)=nlhat(i,j)
      END DO
    END DO
    !$acc end kernels
    !$acc kernels loop gang(gangsize), vector(vecsize)
    DO j=1,Ny
      DO i=1,Nx/2+1
        omegoldhat(i,j)=omeghat(i,j)
      END DO
    END DO
    !$acc end kernels
    DO WHILE (chg>tol)
      !Crank-Nicolson timestepping to get \hat{\omega^{n+1,k+1}}
      !$acc kernels loop gang(gangsize), vector(vecsize)
      DO j=1,Ny
        DO i=1,Nx/2+1
          omeghat(i,j)=( (dtInv+0.5d0*ReInv*(kx(i)*kx(i)+ky(j)*ky(j)))&
              *omegoldhat(i,j) - 0.5d0*(nloldhat(i,j)+nlhat(i,j)))/&
```

```
                (dtInv-0.5d0*ReInv*(kx(i)*kx(i)+ky(j)*ky(j)))
          END DO
        END DO
        !$acc end kernels
        CALL cufftExecZ2D(planz2d,omeghat,omeg)

        ! check for convergence
        chg=0.0d0
        !$acc kernels loop gang(gangsize), vector(vecsize)
        DO j=1,Ny
          DO i=1,Nx
            chg=chg+(omeg(i,j)-temp2(i,j))*(omeg(i,j)-temp2(i,j))&
            *scalemodes*scalemodes
          END DO
        END DO
        !$acc end kernels

        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        !get nonlinear term using omeghat, psihat, u, and v!
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        !\hat{\psi^{n+1,k+1}}
        !$acc kernels loop gang(gangsize), vector(vecsize)
        DO j=1,Ny
          DO i=1,Nx/2+1
            psihat(i,j)=-omeghat(i,j)/(kx(i)*kx(i)+ky(j)*ky(j) + 0.1d0
               **14)
          END DO
        END DO
        !$acc end kernels
        !\omega^{n+1,k+1}
        CALL cufftExecZ2D(planz2d,omeghat,omeg)

        !get \hat{\psi_y^{n+1,k+1}} used to get u, NOTE: u=\psi_y
        !$acc kernels loop gang(gangsize), vector(vecsize)
        DO j=1,Ny
          DO i=1,Nx/2+1
            temp1(i,j)=psihat(i,j)*ky(j)*scalemodes
          END DO
        END DO
        !$acc end kernels
        CALL cufftExecZ2D(planz2d,temp1,temp3) !u

        ! \hat{\omega_x^{n,k}}
        !$acc kernels loop
        DO j=1,Ny
          DO i=1,Nx/2+1
            temp1(i,j)=omeghat(i,j)*kx(i)
          END DO
        END DO
        !$acc end kernels
        ! \omega_x^{n,k}
```

```
      CALL cufftExecZ2D(planz2d,temp1,temp2)

      ! first part nonlinear term in real space
      !$acc kernels loop
      DO j=1,Ny
        DO i=1,Nx
          nl(i,j)=temp3(i,j)*temp2(i,j)
        END DO
      END DO
      !$acc end kernels

      !get \hat{\psi_x^{n+1,k+1}} used to get v, NOTE: v=-\psi_x
      !$acc kernels loop gang(gangsize), vector(vecsize)
      DO j=1,Ny
        DO i=1,Nx/2+1
          temp1(i,j)=-psihat(i,j)*kx(i)*scalemodes
        END DO
      END DO
      !$acc end kernels
      CALL cufftExecZ2D(planz2d,temp1,temp3)

      ! \hat{\omega_y^{n,k}}
      !$acc kernels loop
      DO j=1,Ny
        DO i=1,Nx/2+1
          temp1(i,j)=omeghat(i,j)*ky(j)
        END DO
      END DO
      !$acc end kernels
      ! \omega_y^{n,k}
      CALL cufftExecZ2D(planz2d,temp1,temp2)

      ! get the rest of nonlinear term in real space
      !$acc kernels loop
      DO j=1,Ny
        DO i=1,Nx
          nl(i,j)=(nl(i,j)+temp3(i,j)*temp2(i,j))*scalemodes
        END DO
      END DO
      !$acc end kernels
      ! transform nonlinear term into fourier space
      CALL cufftExecD2Z(pland2z,nl,nlhat)
      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

      !\omega^{n+1,k+1} is saved for next iteration
      !$acc kernels loop gang(gangsize), vector(vecsize)
      DO j=1,Ny
        DO i=1,Nx
          temp2(i,j)=omeg(i,j)
        END DO
      END DO
      !$acc end kernels
    END DO
  END DO
  time(t+1)=time(t)+dt*plotgap
  !PRINT *, time(t+1)
END DO
CALL system_clock(finish,count_rate)
PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),&
    'for Time stepping'

!get exact omega
!$acc kernels loop gang(gangsize), vector(vecsize)
DO j=1,Ny
  DO i=1,Nx
    omegexact(i,j)=4.0d0*pi*sin(2.0d0*pi*x(i))*&
      sin(2.0d0*pi*y(j))*exp(-8.0d0*ReInv*pi**2*nplots*plotgap*dt)
  END DO
END DO
!$acc end kernels
!$acc end data

!compute max error
PRINT *,'Max Error:',maxval(abs(omeg*scalemodes-omegexact))

!!!!!!!!!!!!!!!!!!!!!!!!
!copy over data to disk!
!!!!!!!!!!!!!!!!!!!!!!!!
write(name_config,'(a,i0,a)') 'omega',1,'.datbin'
INQUIRE(iolength=iol) omeg(1,1)
OPEN(unit=11,FILE=name_config,form="unformatted", access="direct",recl=
   iol)
count = 1
DO j=1,Ny
  DO i=1,Nx
    WRITE(11,rec=count) omeg(i,j)*scalemodes
    count=count+1
  END DO
END DO
CLOSE(11)

name_config = 'time.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO i=1,Nplots+1
  WRITE(11,*) time(i)
END DO
CLOSE(11)

name_config = 'xcoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
```

```
DO i=1,Nx
  WRITE(11,*) x(i)
END DO
CLOSE(11)

name_config = 'ycoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,Ny
  WRITE(11,*) y(j)
END DO
CLOSE(11)
!!!!!!!!!!!!!!!!!!!!!!!!

CALL cufftDestroy(pland2z)
CALL cufftDestroy(planz2d)

DEALLOCATE(time,temp1,temp2,temp3,kx,ky,x,y,&
    omeg,omegoldhat,omegexact, nloldhat,&
    omeghat,nl, nlhat, psihat,&
    stat=AllocateStatus)
IF (AllocateStatus .ne. 0) STOP
PRINT *,'Program execution complete'

END PROGRAM main
```

A.2 2D Cubic Nonlinear Schrödinger Equations

These programs use splitting.

Listing A.3: A CUDA Fortran program to solve the 2D Nonlinear Schrödinger equation.

```
!--------------------------------------------------------------------
!
! PURPOSE
!
! This program solves nonlinear Schrodinger equation in 2 dimensions
! i*u_t+Es*|u|^2u+u_{xx}+u_{yy}=0
! using a second order time spectral splitting scheme
!
! The boundary conditions are u(x=0,y)=u(x=2*L*\pi,y)
! and u(x,y=0)=u(x,y=2*L*\pi)
! The initial condition is u=exp(-x^2-y^2)
!
! AUTHORS
!
! B. Cloutier, B.K. Muite, P. Rigge
! 4 June 2012
!
```

```
! .. Parameters ..
!  Nx       = number of modes in x - power of 2 for FFT
!  Ny       = number of modes in y - power of 2 for FFT
!  Nt       = number of timesteps to take
!  Tmax       = maximum simulation time
!  plotgap      = number of timesteps between plots
!  pi = 3.14159265358979323846264338327950288419716939937510d0
!  L        = width of box
!  ES       = +1 for focusing and -1 for defocusing
! .. Scalars ..
!  i        = loop counter in x direction
!  j        = loop counter in y direction
!  n        = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  start      = variable to record start time of program
!  finish     = variable to record end time of program
!  count_rate   = variable for clock count rate
!  plan       = fft plan
!  dt       = timestep
!  InMass     = initial mass
!  FiMass     = final mass
!  InEner     = initial energy
!  FiEner     = final energy
! .. Arrays ..
!  u        = approximate solution
!  v        = Fourier transform of approximate solution
!  u_d        = approximate solution on device
!  v_d        = Fourier transform of approximate solution on device
!  temp1_d      = temporary array used to find mass and energy
!  temp2_d      = temporary array used to find mass and energy
! .. Vectors ..
!  kx       = fourier frequencies in x direction
!  ky       = fourier frequencies in y direction
!  x        = x locations
!  y        = y locations
!  time       = times at which save data
!  name_config    = array to store filename for data to be saved
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! This program is based on example code to demonstrate usage of Fortran
   and
! CUDA FFT routines taken from
! http://cudamusing.blogspot.com/2010/05/calling-cufft-from-cuda-fortran
   .html
!
! and
!
! http://cudamusing.blogspot.com/search?q=cublas
```

```
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
!
! External libraries required
! cufft   -- Cuda FFT library
!

!
! Define the INTERFACE to the NVIDIA CUFFT routines
!

module precision
! Precision control

integer, parameter, public :: Single = kind(0.0) ! Single precision
integer, parameter, public :: Double = kind(0.0d0) ! Double precision

integer, parameter, public :: fp_kind = Double
!integer, parameter, public :: fp_kind = Single

end module precision

module cufft

integer, public :: CUFFT_FORWARD = -1
integer, public :: CUFFT_INVERSE = 1
integer, public :: CUFFT_R2C = Z'2a' ! Real to Complex (interleaved)
integer, public :: CUFFT_C2R = Z'2c' ! Complex (interleaved) to Real
integer, public :: CUFFT_C2C = Z'29' ! Complex to Complex, interleaved
integer, public :: CUFFT_D2Z = Z'6a' ! Double to Double-Complex
integer, public :: CUFFT_Z2D = Z'6c' ! Double-Complex to Double
integer, public :: CUFFT_Z2Z = Z'69' ! Double-Complex to Double-Complex

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! cufftPlan2d(cufftHandle *plan, int nx,int ny, cufftType type)
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

interface cufftPlan2d
subroutine cufftPlan2d(plan, nx, ny, type) bind(C,name='cufftPlan2d')
use iso_c_binding
integer(c_int):: plan
integer(c_int),value:: nx, ny, type
end subroutine cufftPlan2d
end interface cufftPlan2d

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! cufftDestroy(cufftHandle plan)
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

interface cufftDestroy
subroutine cufftDestroy(plan) bind(C,name='cufftDestroy')
use iso_c_binding
integer(c_int),value:: plan
end subroutine cufftDestroy
end interface cufftDestroy

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! cufftExecZ2Z(cufftHandle plan,
! cufftDoubleComplex *idata,
! cufftDoubleComplex *odata,
! int direction;
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
interface cufftExecZ2Z
subroutine cufftExecZ2Z(plan, idata, odata, direction) &
& bind(C,name='cufftExecZ2Z')
use iso_c_binding
use precision
integer(c_int),value:: direction
integer(c_int),value:: plan
complex(fp_kind),device,dimension(1:nx,1:ny):: idata,odata
end subroutine cufftExecZ2Z
end interface cufftExecZ2Z

end module cufft

PROGRAM main
use precision
use cufft
! Declare host variables and scalars
IMPLICIT NONE
INTEGER(kind=4), PARAMETER             :: Nx=1024
INTEGER(kind=4), PARAMETER             :: Ny=1024
INTEGER(kind=4), PARAMETER             :: Nt=20
INTEGER(kind=4), PARAMETER             :: plotgap=5
REAL(fp_kind), PARAMETER    &
  ::  pi=3.14159265358979323846264338327950288419716939937510d0
REAL(fp_kind), PARAMETER             :: Lx=5.0d0
REAL(fp_kind), PARAMETER             :: Ly=5.0d0
```

```fortran
REAL(fp_kind), PARAMETER             :: Es=1.0d0
REAL(fp_kind)                     :: dt=0.10d0**5
REAL(fp_kind)                     :: scalemodes
COMPLEX(fp_kind)                    :: InMass,FiMass,InEner,FiEner
REAL(fp_kind),  DIMENSION(:), ALLOCATABLE    :: x,y
COMPLEX(fp_kind),  DIMENSION(:,:), ALLOCATABLE   :: u
REAL(fp_kind), DIMENSION(:), ALLOCATABLE     :: time
INTEGER(kind=4) :: i,j,k,n,modes,AllocateStatus,plan, kersize
INTEGER(kind=4) :: start, finish, count_rate
  CHARACTER*100 :: name_config
! Declare variables for GPU
COMPLEX(fp_kind), DEVICE, DIMENSION(:), ALLOCATABLE :: kx_d,ky_d
REAL(fp_kind), DEVICE, DIMENSION(:), ALLOCATABLE   :: x_d,y_d
COMPLEX(fp_kind), DEVICE, DIMENSION(:,:), ALLOCATABLE :: u_d,v_d,temp1_d
   ,temp2_d
REAL(fp_kind),DEVICE,DIMENSION(:), ALLOCATABLE     :: time_d

! start timing
PRINT *,'Program starting'
PRINT *,'Grid: ',Nx,'X',Ny
  ! allocate arrays on the host
ALLOCATE(x(1:Nx),y(1:Ny),u(1:Nx,1:Ny),time(1:Nt+1),&
  stat=AllocateStatus)
IF (allocatestatus .ne. 0) STOP
PRINT *,'Allocated CPU arrays'

! allocate arrays on the device
ALLOCATE(kx_d(1:Nx),ky_d(1:Nx),x_d(1:Nx),y_d(1:Nx),&
    u_d(1:Nx,1:Ny),v_d(1:Nx,1:Ny),time_d(1:Nt+1),&
    temp1_d(1:Nx,1:Ny),temp2_d(1:Nx,1:Ny),stat=AllocateStatus)
IF (allocatestatus .ne. 0) STOP
PRINT *,'Allocated GPU arrays'

kersize=min(Nx,256)
! set up ffts
CALL cufftPlan2D(plan,nx,ny,CUFFT_Z2Z)
PRINT *,'Setup FFTs'
  ! setup fourier frequencies
!$cuf kernel do <<< *, * >>>
DO i=1,1+Nx/2
  kx_d(i)= cmplx(0.0d0,1.0d0)*(i-1.0d0)/Lx
END DO
kx_d(1+Nx/2)=0.0d0
!$cuf kernel do <<< *, * >>>
DO i = 1,Nx/2 -1
  kx_d(i+1+Nx/2)=-kx_d(1-i+Nx/2)
END DO
!$cuf kernel do <<< *, * >>>
DO i=1,Nx
  x_d(i)=(-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind=fp_kind))*pi*
     Lx
```

```
END DO
!$cuf kernel do <<< *, * >>>
DO j=1,1+Ny/2
  ky_d(j)= cmplx(0.0d0,1.0d0)*(j-1.0d0)/Ly
END DO
ky_d(1+Ny/2)=0.0d0
!$cuf kernel do <<< *, * >>>
DO j = 1,Ny/2 -1
  ky_d(j+1+Ny/2)=-ky_d(1-j+Ny/2)
END DO
!$cuf kernel do <<< *, * >>>
DO j=1,Ny
  y_d(j)=(-1.0d0 + 2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind=fp_kind))*pi*
     Ly
END DO
scalemodes=1.0d0/REAL(Nx*Ny,kind=fp_kind)
PRINT *,'Setup grid and fourier frequencies'

!$cuf kernel do <<<  *,*  >>>
DO j=1,Ny
  DO i=1,Nx
    u_d(i,j)=exp(-1.0d0*(x_d(i)**2+y_d(j)**2))
  END DO
END DO
! transform initial data
CALL cufftExecZ2Z(plan,u_d,v_d,CUFFT_FORWARD)

PRINT *,'Got initial data'
! get initial mass
!$cuf kernel do <<<  *,* >>>
DO j=1,Ny
  DO i=1,Nx
    temp1_d(i,j)=abs(u_d(i,j))**2
  END DO
END DO
! Use FFT to get initial mass
CALL cufftExecZ2Z(plan,temp1_d,temp2_d,CUFFT_FORWARD)
InMass=temp2_d(1,1)
! Get initial energy
!$cuf kernel do <<<  *,* >>>
DO j=1,Ny
  DO i=1,Nx
    temp1_d(i,j)=-ES*0.25d0*abs(u_d(i,j))**4
  END DO
END DO
! Use FFT to find mean
CALL cufftExecZ2Z(plan,temp1_d,temp2_d,CUFFT_FORWARD)
InEner=temp2_d(1,1)
!$cuf kernel do <<< *,* >>>
DO j=1,Ny
  DO i=1,Nx
```

```
      temp2_d(i,j)=kx_d(i)*v_d(i,j)*scalemodes
    END DO
  END DO
  CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_INVERSE)
  !$cuf kernel do <<< *,* >>>
  DO j=1,Ny
    DO i=1,Nx
      temp2_d(i,j)=0.5d0*abs(temp1_d(i,j))**2
    END DO
  END DO
  ! Use FFT to find mean
  CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_FORWARD)
  InEner=InEner+temp1_d(1,1)
  !$cuf kernel do <<<  *,*  >>>
  DO j=1,Ny
    DO i=1,Nx
      temp2_d(i,j)=ky_d(j)*v_d(i,j)*scalemodes
    END DO
  END DO
  CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_INVERSE)
  !$cuf kernel do <<< *,* >>>
  DO j=1,Ny
    DO i=1,Nx
      temp2_d(i,j)=0.5d0*abs(temp1_d(i,j))**2
    END DO
  END DO
  ! Use FFT to find mean
  CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_FORWARD)
  InEner=InEner+temp1_d(1,1)

  ! start timing
  CALL system_clock(start,count_rate)
  ! Do first half time step
  CALL cufftExecZ2Z(plan,u_d,v_d,CUFFT_FORWARD)
  !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
  DO j=1,Ny
    DO i=1,Nx
        v_d(i,j)=exp(dt*( kx_d(i)*kx_d(i) + ky_d(j)*ky_d(j) )&
          *cmplx(0.0d0,0.50d0) )*v_d(i,j)
    END DO
  END DO

  PRINT *,'Starting timestepping'
  time(1)=0.0d0
  DO n=1,Nt
    time_d(n+1)=n*dt
    CALL cufftExecZ2Z(plan,v_d,u_d,CUFFT_INVERSE)
    !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
    DO j=1,Ny
      DO i=1,Nx
        v_d(i,j)=Es*u_d(i,j)*conjg(u_d(i,j))*scalemodes**2
      END DO
    END DO
    !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
    DO j=1,Ny
      DO i=1,Nx
        u_d(i,j)=exp(cmplx(0.0d0,-1.0d0)*dt*v_d(i,j))&
            *u_d(i,j)*scalemodes
      END DO
    END DO

    CALL cufftExecZ2Z(plan,u_d,v_d,CUFFT_FORWARD)
    !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
    DO j=1,Ny
      DO i=1,Nx
        v_d(i,j)=exp(dt*(kx_d(i)*kx_d(i) + ky_d(j)*ky_d(j))&
          *cmplx(0.0d0,1.0d0))*v_d(i,j)
      END DO
    END DO

  END DO

  ! transform back final data and do another half time step

  CALL cufftExecZ2Z(plan,v_d,u_d,CUFFT_INVERSE)
  !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
  DO j=1,Ny
    DO i=1,Nx
      v_d(i,j)=Es*u_d(i,j)*conjg(u_d(i,j))*scalemodes**2
    END DO
  END DO
  !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
  DO j=1,Ny
    DO i=1,Nx
      u_d(i,j)=exp(cmplx(0.0d0,-1.0d0)*dt*v_d(i,j))&
          *u_d(i,j)*scalemodes
    END DO
  END DO

  CALL cufftExecZ2Z(plan,u_d,v_d,CUFFT_FORWARD)
  !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
  DO j=1,Ny
    DO i=1,Nx
      v_d(i,j)=exp(dt*(kx_d(i)*kx_d(i) + ky_d(j)*ky_d(j))&
          *cmplx(0.0d0,0.50d0))*v_d(i,j)
    END DO
  END DO
  CALL cufftExecZ2Z(plan,v_d,u_d,CUFFT_INVERSE)
  ! normalize
  !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
  DO j=1,Ny
    DO i=1,Nx
      u_d(i,j)=u_d(i,j)*scalemodes
    END DO
  END DO

  CALL system_clock(finish,count_rate)
  PRINT*,'Program took ',&
    REAL(finish-start,kind(0d0))/REAL(count_rate,kind(0d0)),'s for
       execution'

  PRINT *,'Finished time stepping'

  ! calculate final mass
  !$cuf kernel do <<<  *,*  >>>
  DO j=1,Ny
    DO i=1,Nx
      temp1_d(i,j)=abs(u_d(i,j))**2
    END DO
  END DO
  ! Use FFT to get initial mass
  CALL cufftExecZ2Z(plan,temp1_d,temp2_d,CUFFT_FORWARD)
  FiMass=temp2_d(1,1)

  PRINT*,'Initial mass',InMass
  PRINT*,'Final mass',FiMass
  PRINT*,'Final Mass/Initial Mass', &
     ABS(REAL(FiMass,kind=fp_kind)/REAL(InMass,kind=fp_kind))


  ! Get final energy
  !$cuf kernel do <<<  *,*  >>>
  DO j=1,Ny
    DO i=1,Nx
      temp1_d(i,j)=-ES*0.25d0*abs(u_d(i,j))**4
    END DO
  END DO
  ! Use FFT to find mean
  CALL cufftExecZ2Z(plan,temp1_d,temp2_d,CUFFT_FORWARD)
  FiEner=temp2_d(1,1)
  !$cuf kernel do <<<  *,*  >>>
  DO j=1,Ny
    DO i=1,Nx
      temp2_d(i,j)=kx_d(i)*v_d(i,j)*scalemodes
    END DO
  END DO
  CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_INVERSE)
  !$cuf kernel do <<<  *,*  >>>
  DO j=1,Ny
    DO i=1,Nx
      temp2_d(i,j)=0.5d0*abs(temp1_d(i,j))**2
    END DO
  END DO
```

```fortran
! Use FFT to find mean
CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_FORWARD)
FiEner=FiEner+temp1_d(1,1)
!$cuf kernel do <<< *,* >>>
DO j=1,Ny
  DO i=1,Nx
    temp2_d(i,j)=ky_d(j)*v_d(i,j)*scalemodes
  END DO
END DO
CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_INVERSE)
!$cuf kernel do <<<  *,*  >>>
DO j=1,Ny
  DO i=1,Nx
    temp2_d(i,j)=0.5d0*abs(temp1_d(i,j))**2
  END DO
END DO
! Use FFT to find mean
CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_FORWARD)
FiEner=FiEner+temp1_d(1,1)

PRINT*,'Initial energy',InEner
PRINT*,'Final energy',FiEner
PRINT*,'Final Energy/Initial Energy', &
   ABS(REAL(FiEner,kind=fp_kind)/REAL(InEner,kind=fp_kind))

! Copy results back to host
u=u_d
time=time_d
x=x_d
y=y_d

name_config = 'ufinal.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,Ny
  DO i=1,Nx
    WRITE(11,*) abs(u(i,j))**2
  END DO
END DO
CLOSE(11)

name_config = 'tdata.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,1+Nt/plotgap
  WRITE(11,*) time(j)
END DO
CLOSE(11)

name_config = 'xcoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
```

```
REWIND(11)
DO i=1,Nx
  WRITE(11,*) x(i)
END DO
CLOSE(11)

name_config = 'ycoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,Ny
  WRITE(11,*) y(j)
END DO
CLOSE(11)
PRINT *,'Saved data'

! Destroy the plan
CALL cufftDestroy(plan)

DEALLOCATE(kx_d,ky_d,x_d,y_d,&
    u_d,v_d,time_d,&
    temp1_d,temp2_d,&
    stat=AllocateStatus)
IF (allocatestatus .ne. 0) STOP
DEALLOCATE(x,y,u,time,&
  stat=AllocateStatus)
IF (allocatestatus .ne. 0) STOP
PRINT *,'deallocated memory'
PRINT *,'Program execution complete'
END PROGRAM main
```

Listing A.4: An OpenACC Fortran program to solve the 2D Nonlinear Schrödinger equation.

```
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This program solves nonlinear Schrodinger equation in 2 dimensions
! i*u_t+Es*|u|^2u+u_{xx}+u_{yy}=0
! using a second order time spectral splitting scheme
!
! The boundary conditions are u(x=0,y)=u(2*Lx*\pi,y),
! u(x,y=0)=u(x,y=2*Ly*\pi)
! The initial condition is u=exp(-x^2-y^2)
!
! AUTHORS
!
! B. Cloutier, B.K. Muite, P. Rigge
! 4 June 2012
!
```

```
! .. Parameters ..
!  Nx        = number of modes in x - power of 2 for FFT
!  Ny        = number of modes in y - power of 2 for FFT
!  Nt        = number of timesteps to take
!  Tmax        = maximum simulation time
!  plotgap       = number of timesteps between plots
!  FFTW_IN_PLACE  = value for FFTW input
!  FFTW_MEASURE   = value for FFTW input
!  FFTW_EXHAUSTIVE  = value for FFTW input
!  FFTW_PATIENT   = value for FFTW input
!  FFTW_ESTIMATE  = value for FFTW input
!  FFTW_FORWARD     = value for FFTW input
!  FFTW_BACKWARD  = value for FFTW input
!  pi = 3.14159265358979323846264338327950288419716939937510d0
!  Lx        = width of box in x direction
!  Ly        = width of box in y direction
!  ES        = +1 for focusing and -1 for defocusing
! .. Scalars ..
!  i         = loop counter in x direction
!  j         = loop counter in y direction
!  n         = loop counter for timesteps direction
!  allocatestatus = error indicator during allocation
!  numthreads   = number of openmp threads
!  ierr        = error return code
!  start       = variable to record start time of program
!  finish      = variable to record end time of program
!  count_rate   = variable for clock count rate
!  planfxy      = Forward 2d fft plan
!  planbxy      = Backward 2d fft plan
!  dt        = timestep
!  InMass      = initial mass
!  FiMass      = final mass
!  InEner      = initial energy
!  FiEner      = final energy
! .. Arrays ..
!  u         = approximate solution
!  v         = Fourier transform of approximate solution
!  temp1       = temporary field
!  temp2       = temporary field
! .. Vectors ..
!  kx        = fourier frequencies in x direction
!  ky        = fourier frequencies in y direction
!  x         = x locations
!  y         = y locations
!  time        = times at which save data
!  name_config    = array to store filename for data to be saved
!
! REFERENCES
!
! This program is based on example code to demonstrate usage of Fortran
   and
```

```
! CUDA FFT routines taken from
! http://cudamusing.blogspot.com/2010/05/CALLing-cufft-from-cuda-fortran
  .html
!
! and
!
! http://cudamusing.blogspot.com/search?q=cublas
!
! ACKNOWLEDGEMENTS
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!----------------------------------------------------------------------
! External routines required
! precision
! cufft
!
! External libraries required
! CuFFT  -- Cuda FFT Library
! OpenACC

!
! Define the INTERFACE to the NVIDIA CUFFT routines
!

module precision
! Precision control

integer, parameter, public :: Single = kind(0.0) ! Single precision
integer, parameter, public :: Double = kind(0.0d0) ! Double precision

integer, parameter, public :: fp_kind = Double
!integer, parameter, public :: fp_kind = Single

end module precision

module cufft

integer, public :: CUFFT_FORWARD = -1
integer, public :: CUFFT_INVERSE = 1
integer, public :: CUFFT_R2C = Z'2a' ! Real to Complex (interleaved)
integer, public :: CUFFT_C2R = Z'2c' ! Complex (interleaved) to Real
integer, public :: CUFFT_C2C = Z'29' ! Complex to Complex, interleaved
integer, public :: CUFFT_D2Z = Z'6a' ! Double to Double-Complex
integer, public :: CUFFT_Z2D = Z'6c' ! Double-Complex to Double
integer, public :: CUFFT_Z2Z = Z'69' ! Double-Complex to Double-Complex
```

```fortran

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! cufftPlan2d(cufftHandle *plan, int nx,int ny, cufftType type)
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

interface cufftPlan2d
subroutine cufftPlan2d(plan, nx, ny, type) bind(C,name='cufftPlan2d')
use iso_c_binding
integer(c_int):: plan
integer(c_int),value:: nx, ny, type
end subroutine cufftPlan2d
end interface cufftPlan2d

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! cufftDestroy(cufftHandle plan)
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

interface cufftDestroy
subroutine cufftDestroy(plan) bind(C,name='cufftDestroy')
use iso_c_binding
integer(c_int),value:: plan
end subroutine cufftDestroy
end interface cufftDestroy

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! cufftExecZ2Z(cufftHandle plan,
! cufftDoubleComplex *idata,
! cufftDoubleComplex *odata,
! int direction;
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
interface cufftExecZ2Z
subroutine cufftExecZ2Z(plan, idata, odata, direction) &
& bind(C,name='cufftExecZ2Z')
use iso_c_binding
use precision
integer(c_int),value:: direction
integer(c_int),value:: plan
complex(fp_kind),device,dimension(1:nx,1:ny):: idata,odata
end subroutine cufftExecZ2Z
end interface cufftExecZ2Z
end module cufft

PROGRAM main
USE precision
USE cufft
USE openacc

! Declare variables
IMPLICIT NONE
INTEGER(kind=4), PARAMETER  ::  Nx=128
INTEGER(kind=4), PARAMETER  ::  Ny=128
INTEGER(kind=4), PARAMETER  ::  Nt=20
INTEGER(kind=4), PARAMETER  ::  plotgap=20
REAL(fp_kind), PARAMETER    ::  &
pi=3.14159265358979323846264338327950288419716939937510d0
REAL(fp_kind), PARAMETER    ::  Lx=5.0d0
REAL(fp_kind), PARAMETER    ::  Ly=5.0d0
REAL(fp_kind), PARAMETER    ::  Es=1.0d0
REAL(fp_kind)          ::  dt=0.10d0**5
REAL(fp_kind)          :: scalemodes
COMPLEX(fp_kind)         :: InMass,FiMass,InEner,FiEner
COMPLEX(fp_kind), DIMENSION(:), ALLOCATABLE   ::  kx
COMPLEX(fp_kind), DIMENSION(:), ALLOCATABLE   ::  ky
REAL(fp_kind),     DIMENSION(:), ALLOCATABLE  ::  x
REAL(fp_kind),     DIMENSION(:), ALLOCATABLE  ::  y
COMPLEX(fp_kind), DIMENSION(:,:), ALLOCATABLE ::  u,v,temp1,temp2
REAL(fp_kind),   DIMENSION(:), ALLOCATABLE    ::  time
INTEGER(kind=4)        ::  i,j,k,n,allocatestatus,ierr, vecsize,gangsize
REAL(fp_kind)        ::  start_time,stop_time
INTEGER(kind=4)        ::  plan
  CHARACTER*100        ::  name_config

vecsize=32
gangsize=16
PRINT *,'Program starting'
PRINT *,'Grid: ',Nx,'X',Ny

ALLOCATE(kx(1:Nx),ky(1:Nx),x(1:Nx),y(1:Nx),u(1:Nx,1:Ny),&
    v(1:Nx,1:Ny),temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny),&
    time(1:1+Nt/plotgap),stat=allocatestatus)
IF (allocatestatus .ne. 0) stop
PRINT *,'allocated memory'

!$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
   time)

! set up ffts
CALL cufftPlan2D(plan,nx,ny,CUFFT_Z2Z)
PRINT *,'Setup FFTs'

! setup fourier frequencies
!$acc kernels loop
DO i=1,1+Nx/2
  kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
END DO
!$acc end kernels
```

```
kx(1+Nx/2)=0.0d0
!$acc kernels loop
DO i = 1,Nx/2 -1
  kx(i+1+Nx/2)=-kx(1-i+Nx/2)
END DO
!$acc end kernels
!$acc kernels loop
  DO i=1,Nx
  x(i)=(-1.0d0+2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)) )*pi*Lx
END DO
!$acc end kernels
!$acc kernels loop
DO j=1,1+Ny/2
  ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
END DO
!$acc end kernels
ky(1+Ny/2)=0.0d0
!$acc kernels loop
DO j = 1,Ny/2 -1
  ky(j+1+Ny/2)=-ky(1-j+Ny/2)
END DO
!$acc end kernels
!$acc kernels loop
  DO j=1,Ny
  y(j)=(-1.0d0+2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)) )*pi*Ly
END DO
!$acc end kernels
scalemodes=1.0d0/REAL(Nx*Ny,kind(0d0))
PRINT *,'Setup grid and fourier frequencies'
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    u(i,j)=exp(-1.0d0*(x(i)**2 +y(j)**2))
  END DO
END DO
!$acc end kernels
! transform initial data
CALL cufftExecZ2Z(plan,u,v,CUFFT_FORWARD)

PRINT *,'Got initial data'
! get initial mass
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    temp1(i,j)=abs(u(i,j))**2
  END DO
END DO
!$acc end kernels
! Use FFT to get initial mass
CALL cufftExecZ2Z(plan,temp1,temp2,CUFFT_FORWARD)
!$acc end data
```

```
InMass=temp2(1,1)
! Get initial energy
!$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
   time)
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    temp1(i,j)=-ES*0.25d0*abs(u(i,j))**4
  END DO
END DO
!$acc end kernels
! Use FFT to find mean
CALL cufftExecZ2Z(plan,temp1,temp2,CUFFT_FORWARD)
!$acc end data
InEner=temp2(1,1)
!$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
   time)
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    temp2(i,j)=kx(i)*v(i,j)*scalemodes
  END DO
END DO
!$acc end kernels
CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_INVERSE)
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    temp2(i,j)=0.5d0*abs(temp1(i,j))**2
  END DO
END DO
!$acc end kernels
! Use FFT to find mean
CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_FORWARD)
!$acc end data
InEner=InEner+temp1(1,1)
!$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
   time)
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    temp2(i,j)=ky(j)*v(i,j)*scalemodes
  END DO
END DO
!$acc end kernels
CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_INVERSE)
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    temp2(i,j)=0.5d0*abs(temp1(i,j))**2
  END DO
END DO
!$acc end kernels
! Use FFT to find mean
CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_FORWARD)
!$acc end data
InEner=InEner+temp1(1,1)
!$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
   time)
CALL cpu_time(start_time)


! transform initial data and do first half time step
!$acc kernels loop gang(gangsize), vector(vecsize)
DO j=1,Ny
  DO i=1,Nx
    v(i,j)=exp(0.5d0*dt*(kx(i)*kx(i) + ky(j)*ky(j))&
        *cmplx(0.0d0,1.0d0))*v(i,j)
  END DO
END DO
!$acc end kernels
PRINT *,'Got initial data, starting timestepping'
time(1)=0.0d0
DO n=1,Nt
  CALL cufftExecZ2Z(plan,v,u,CUFFT_INVERSE)
    !$acc kernels loop gang(gangsize), vector(vecsize)
  DO j=1,Ny
    DO i=1,Nx
      v(i,j)=Es*u(i,j)*conjg(u(i,j))*scalemodes**2
    END DO
  END DO
    !$acc end kernels
    !$acc kernels loop gang(gangsize), vector(vecsize)
  DO j=1,Ny
    DO i=1,Nx
      u(i,j)=exp(cmplx(0.0d0,-1.0d0)*dt*v(i,j))&
          *u(i,j)*scalemodes
    END DO
  END DO
    !$acc end kernels
  CALL cufftExecZ2Z(plan,u,v,CUFFT_FORWARD)
    !$acc kernels loop gang(gangsize), vector(vecsize)
  DO j=1,Ny
    DO i=1,Nx
      v(i,j)=exp(dt*(kx(i)*kx(i) + ky(j)*ky(j))&
          *cmplx(0.0d0,1.0d0))*v(i,j)
    END DO
  END DO
    !$acc end kernels
  IF (mod(n,plotgap)==0) then
    time(1+n/plotgap)=n*dt
    PRINT *,'time',n*dt
  END IF
END DO
! transform back final data and do another half time step
CALL cufftExecZ2Z(plan,v,u,CUFFT_INVERSE)
!$acc kernels loop gang(gangsize), vector(vecsize)
DO j=1,Ny
  DO i=1,Nx
    v(i,j)=Es*u(i,j)*conjg(u(i,j))*scalemodes**2
  END DO
END DO
!$acc end kernels
!$acc kernels loop gang(gangsize), vector(vecsize)
DO j=1,Ny
  DO i=1,Nx
    u(i,j)=exp(cmplx(0,-1)*dt*v(i,j))*u(i,j)*scalemodes
  END DO
END DO
!$acc end kernels
CALL cufftExecZ2Z(plan,u,v,CUFFT_FORWARD)
!$acc kernels loop gang(gangsize), vector(vecsize)
DO j=1,Ny
  DO i=1,Nx
    v(i,j)=exp(0.5d0*dt*(kx(i)*kx(i) + ky(j)*ky(j))&
        *cmplx(0.0d0,1.0d0))*v(i,j)
  END DO
END DO
!$acc end kernels
CALL cufftExecZ2Z(plan,v,u,CUFFT_INVERSE)
!$acc kernels loop gang(gangsize), vector(vecsize)
DO j=1,Ny
  DO i=1,Nx
    u(i,j)=u(i,j)*scalemodes
  END DO
END DO
!$acc end kernels
PRINT *,'Finished time stepping'
CALL cpu_time(stop_time)
!$acc end data
PRINT*,'Program took ',stop_time-start_time,&
  'for Time stepping'
!$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
    time)

! calculate final mass
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    temp1(i,j)=abs(u(i,j))**2
  END DO
END DO
!$acc end kernels
```

```fortran
! Use FFT to get initial mass
CALL cufftExecZ2Z(plan,temp1,temp2,CUFFT_FORWARD)
!$acc end data
FiMass=temp2(1,1)


! Get final energy
!$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
    time)
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    temp1(i,j)=-ES*0.25d0*abs(u(i,j))**4
  END DO
END DO
!$acc end kernels
! Use FFT to find mean
CALL cufftExecZ2Z(plan,temp1,temp2,CUFFT_FORWARD)
!$acc end data
FiEner=temp2(1,1)
!$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
    time)
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    temp2(i,j)=kx(i)*v(i,j)*scalemodes
  END DO
END DO
!$acc end kernels
CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_INVERSE)
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    temp2(i,j)=0.5d0*abs(temp1(i,j))**2
  END DO
END DO
!$acc end kernels
! Use FFT to find mean
CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_FORWARD)
!$acc end data
FiEner=FiEner+temp1(1,1)
!$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
    time)
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    temp2(i,j)=ky(j)*v(i,j)*scalemodes
  END DO
END DO
!$acc end kernels
CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_INVERSE)
```

```
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    temp2(i,j)=0.5d0*abs(temp1(i,j))**2
  END DO
END DO
!$acc end kernels
! Use FFT to find mean
CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_FORWARD)
!$acc end data
FiEner=FiEner+temp1(1,1)

PRINT *,'Results copied back to host'
PRINT*,'Initial mass',InMass
PRINT*,'Final mass',FiMass
PRINT*,'Final Mass/Initial Mass', &
   ABS(REAL(FiMass,kind(0d0))/REAL(InMass,kind(0d0)))
PRINT*,'Initial energy',InEner
PRINT*,'Final energy',FiEner
PRINT*,'Final Energy/Initial Energy', &
   ABS(REAL(FiEner,kind(0d0))/REAL(InEner,kind(0d0)))

name_config = 'ufinal.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,Ny
  DO i=1,Nx
    WRITE(11,*) abs(u(i,j))**2
  END DO
END DO
CLOSE(11)

name_config = 'tdata.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO j=1,1+Nt/plotgap
  WRITE(11,*) time(j)
END DO
CLOSE(11)

name_config = 'xcoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
DO i=1,Nx
  WRITE(11,*) x(i)
END DO
CLOSE(11)

name_config = 'ycoord.dat'
OPEN(unit=11,FILE=name_config,status="UNKNOWN")
REWIND(11)
```

```fortran
  DO j=1,Ny
    WRITE(11,*) y(j)
  END DO
  CLOSE(11)
  PRINT *,'Saved data'

  CALL cufftDestroy(plan)

  DEALLOCATE(u,v,temp1,temp2,time,kx,ky,x,y,stat=allocatestatus)
  IF (allocatestatus .ne. 0) STOP
  PRINT *,'Deallocated memory'

    PRINT *,'Program execution complete'
  END PROGRAM main
```

A.3 2D sine-Gordon Equations

These programs use a semi-explicit method that is similar to that used for the Klein-Gordon equation. Only the main program is included here, and the auxiliary subroutines can be downloaded from Cloutier, Muite and Rigge [11]

Listing A.5: A CUDA Fortran program to solve the 2D sine-Gordon equation.

```fortran
!--------------------------------------------------------------------
!
!
! PURPOSE
!
! This program solves nonlinear sine-Gordon equation in 2 dimensions
! u_{tt}-u_{xx}-u_{yy}=-sin(u)
! using a second order implicit-explicit time stepping scheme.
!
! The boundary conditions are u(x=0,y)=u(2*Lx*\pi,y),
!       u(x,y=0)=u(x,y=2*Ly*\pi)
! The initial condition is set in initialdata.f90
!
! AUTHORS
!
! B. Cloutier, B.K. Muite, P. Rigge
! 4 June 2012
!
! .. Parameters ..
!  Nx                            = number of modes in x - power of 2 for FFT
!  Ny                            = number of modes in y - power of 2 for FFT
!  Nt                            = number of timesteps to take
!  plotgap                       = number of timesteps between plots
!   FFTW_IN_PLACE                 = value for FFTW input
!   FFTW_MEASURE                  = value for FFTW input
!   FFTW_EXHAUSTIVE               = value for FFTW input
!   FFTW_PATIENT                  = value for FFTW input
!   FFTW_ESTIMATE                 = value for FFTW input
!   FFTW_FORWARD                  = value for FFTW input
!   FFTW_BACKWARD                 = value for FFTW input
!   pi                            = 3.1415926535...
!   Lx                            = width of box in x direction
!   Ly                            = width of box in y direction
! .. Scalars ..
!   i                             = loop counter in x direction
!   j                             = loop counter in y direction
!   n                             = loop counter for timesteps direction
!   allocatestatus                = error indicator during allocation
!   start                         = variable to record start time of program
!   finish                        = variable to record end time of program
!   count_rate                    = variable for clock count rate
!   planfxy                       = Forward 2d fft plan  (FFTW)
!   planbxy                       = Backward 2d fft plan (FFTW)
!   planf                         = Forward 2d fft plan  (CUFFT)
!   planb                         = Backward 2d fft plan (CUFFT)
!   dt                            = timestep
!   ierr                          = error code
!   plotnum                       = number of plot
! .. Arrays ..
!   u                             = approximate solution
!   uold                          = approximate solution
!   u_d                           = approximate solution (on GPU)
!   v_d                           = Fourier transform of approximate
    solution (on GPU)
!   uold_d                        = approximate solution (on GPU)
!   vold_d                        = Fourier transform of approximate
    solution (on GPU)
!   nonlinhat_d                   = Fourier transform of nonlinear term, sin
    (u) (on GPU)
!   temp1                         = extra space for energy computation
!   temp2                         = extra space for energy computation
!   savearray                     = temp array to save out to disk
! .. Vectors ..
!   kx                            = Fourier frequencies in x direction
!   ky                            = Fourier frequencies in y direction
!   kx_d                          = Fourier frequencies in x direction (on
    GPU)
!   ky_d                          = Fourier frequencies in y direction (on
    GPU)
!   x                             = x locations
!   y                             = y locations
!   time                          = times at which save data
!   en                            = total energy
!   enstr                         = strain energy
!   enpot                        = potential energy
!   enkin                        = kinetic energy
!   name_config              = array to store filename for data to be saved
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! This program is based on example code to demonstrate usage of Fortran
   and
! CUDA FFT routines taken from
! http://cudamusing.blogspot.com/2010/05/CALLing-cufft-from-cuda-fortran.
   html
!
! and
!
! http://cudamusing.blogspot.com/search?q=cublas
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
!       getgrid.f90     -- Get initial grid of points
!       initialdata.f90 -- Get initial data
!       enercalc.f90    -- Subroutine to calculate the energy
!       savedata.f90    -- Save initial data
! External libraries required
!       Cuda FFT        -- http://developer.nvidia.com/cufft
!       FFTW3           -- Fastest Fourier Transform in the West
!                       (http://www.fftw.org/)

module precision
  ! Precision control
  integer, parameter, public :: Single = kind(0.0) ! Single precision
  integer, parameter, public :: Double = kind(0.0d0) ! Double precision
  !
  integer, parameter, public :: fp_kind = Double
  !integer, parameter, public :: fp_kind = Single
end module precision

module cufft
  integer, public :: CUFFT_FORWARD = -1
  integer, public :: CUFFT_INVERSE = 1
  integer, public :: CUFFT_R2C = Z'2a' ! Real to Complex (interleaved)
  integer, public :: CUFFT_C2R = Z'2c' ! Complex (interleaved) to Real
  integer, public :: CUFFT_C2C = Z'29' ! Complex to Complex, interleaved
  integer, public :: CUFFT_D2Z = Z'6a' ! Double to Double-Complex
  integer, public :: CUFFT_Z2D = Z'6c' ! Double-Complex to Double
  integer, public :: CUFFT_Z2Z = Z'69' ! Double-Complex to Double-Complex
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !
  ! cufftPlan2d(cufftHandle *plan, int nx,int ny, cufftType type,int batch
     )
  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  interface cufftPlan2d
     subroutine cufftPlan2d(plan, nx, ny, type) bind(C,name='cufftPlan2d')
       use iso_c_binding
       integer(c_int):: plan
       integer(c_int),value:: nx, ny, type
     end subroutine cufftPlan2d
  end interface cufftPlan2d
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !
  ! cufftDestroy(cufftHandle plan)
  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  interface cufftDestroy
     subroutine cufftDestroy(plan) bind(C,name='cufftDestroy')
       use iso_c_binding
       integer(c_int),value:: plan
     end subroutine cufftDestroy
  end interface cufftDestroy
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !
  ! cufftExecD2Z(cufftHandle plan,
  ! cufftDoubleReal    *idata,
  ! cufftDoubleComplex *odata)
  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  interface cufftExecD2Z
     subroutine cufftExecD2Z(plan, idata, odata) &
          & bind(C,name='cufftExecD2Z')
       use iso_c_binding
       use precision
       integer(c_int),  value  :: plan
       real(fp_kind),   device :: idata(1:nx,1:ny)
       complex(fp_kind),device :: odata(1:nx,1:ny)
     end subroutine cufftExecD2Z
  end interface cufftExecD2Z
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !
  ! cufftExecD2Z(cufftHandle plan,
  ! cufftDoubleComplex *idata,
  ! cufftDoubleReal    *odata)
  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
  interface cufftExecZ2D
     subroutine cufftExecZ2D(plan, idata, odata) &
          & bind(C,name='cufftExecZ2D')
       use iso_c_binding
       use precision
       integer(c_int),value   :: plan
       complex(fp_kind),device:: idata(1:nx,1:ny)
       real(fp_kind),device   :: odata(1:nx,1:ny)
     end subroutine cufftExecZ2D
  end interface cufftExecZ2D
end module cufft

PROGRAM sg2d
  USE precision
  USE cudafor
  USE cufft
  ! Declare variables
  IMPLICIT NONE
  INTEGER(kind=4), PARAMETER                              :: Nx=1024
  INTEGER(kind=4), PARAMETER                              :: Ny=Nx
  INTEGER(kind=4), PARAMETER                              :: Nt=500
  INTEGER(kind=4), PARAMETER                              :: plotgap=Nt+1
  REAL(kind=8), PARAMETER                                 :: &
       pi=3.14159265358979323846264338327950288419716939937510d0
  REAL(kind=8), PARAMETER                                 :: Lx=5.0d0
  REAL(kind=8), PARAMETER                                 :: Ly=5.0d0
  REAL(kind=8)                                            :: dt=0.001d0
  COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE              :: kx,ky
  REAL(kind=8),         DIMENSION(:), ALLOCATABLE         :: x,y
  REAL   (kind=8), DIMENSION(:,:), ALLOCATABLE            :: u,uold
  COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE            :: temp1,temp2
  REAL(kind=8), DIMENSION(:,:), ALLOCATABLE               :: savearray
  REAL(kind=8), DIMENSION(:), ALLOCATABLE                 :: time,enkin,enstr
     ,enpot,en
  REAL(kind=8)                                            :: scalemodes
  INTEGER(kind=4)                                         :: ierr,i,j,n,
     allocatestatus
  INTEGER(kind=4)                                         :: start, finish,
     count_rate, plotnum
  INTEGER(kind=4), PARAMETER                              :: FFTW_IN_PLACE =
     8, FFTW_MEASURE = 0, &
       FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32, FFTW_ESTIMATE = 64
  INTEGER(kind=4),PARAMETER                               :: FFTW_FORWARD =
     -1, FFTW_BACKWARD=1
  INTEGER(kind=8)                                         :: planfxy,planbxy
  CHARACTER*100                                           :: name_config
  INTEGER(kind=4)                                         :: planf,planb,
     kersize
  ! GPU variables
  COMPLEX(fp_kind),DEVICE,DIMENSION(:), ALLOCATABLE       :: kx_d,ky_d
  COMPLEX(fp_kind),DEVICE,DIMENSION(:,:), ALLOCATABLE     :: vold_d,v_d,
     nonlinhat_d
  REAL   (fp_kind),DEVICE,DIMENSION(:,:), ALLOCATABLE  :: uold_d,u_d
  ! print run information
  PRINT *,"Nx=", Nx
  PRINT *,"Ny=", Ny
  PRINT *,"Nt=", Nt
  PRINT *,"Lx=", Lx
  PRINT *,"Ly=", Ly
  PRINT *,"dt=", dt
  kersize=min(Nx,256)
  ALLOCATE(kx(1:Nx),ky(1:Ny),kx_d(1:Nx),ky_d(1:Ny),x(1:Nx),y(1:Ny),&
       u(1:Nx,1:Ny),uold(1:Nx,1:Ny),u_d(1:Nx,1:Ny),uold_d(1:Nx,1:Ny),&
       v_d(1:Nx/2+1,1:Ny),vold_d(1:Nx/2+1,1:Ny),&
       savearray(1:Nx,1:Ny),time(1:1+Nt/plotgap),enkin(1:1+Nt/plotgap+1),&
       enstr(1:1+Nt/plotgap+1),enpot(1:1+Nt/plotgap+1),en(1:1+Nt/plotgap)
          ,&
       nonlinhat_d(1:Nx/2+1,1:Ny),&
       temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny),&
       stat=allocatestatus)
  IF (allocatestatus .ne. 0) stop
  PRINT *,'allocated arrays'
  scalemodes=1.0d0/REAL(Nx*Ny,kind(0d0))
  ! set up cuda ffts
  call cufftPlan2D(planf,nx,ny,CUFFT_D2Z)
  call cufftPlan2D(planb,nx,ny,CUFFT_Z2D)
  ! set up fftw ffts
  CALL dfftw_plan_dft_2d_(planfxy,Nx,Ny,u,temp2,FFTW_FORWARD,FFTW_ESTIMATE
     )
  CALL dfftw_plan_dft_2d_(planbxy,Nx,Ny,temp2,u,FFTW_BACKWARD,
     FFTW_ESTIMATE)
  PRINT *,'Setup FFTs'
  ! setup grid, wave numbers
  CALL getgrid(Nx,Ny,Lx,Ly,pi,name_config,x,y,kx,ky)
  kx_d=kx
  ky_d=ky
  PRINT *,'Got grid and fourier frequencies'

  CALL initialdata(Nx,Ny,x,y,u,uold)
  u_d=u
  uold_d=uold
  plotnum=1
  name_config = 'data/u'
  savearray=REAL(u)
  ! CALL savedata(Nx,Ny,plotnum,name_config,savearray) ! disabled for
     benchmarking
  PRINT *,'data saved'

  CALL enercalc(Nx,Ny,planfxy,planbxy,dt,enkin(plotnum),enstr(plotnum),&
       enpot(plotnum),en(plotnum),kx,ky,temp1,temp2,u,uold)
  call cufftExecD2Z(planf,u_d,v_d)
  call cufftExecD2Z(planf,uold_d,vold_d)
```

```
PRINT *,'Got initial data, starting timestepping'
time(plotnum)=0.0d0
CALL system_clock(start,count_rate)
DO n=1,Nt
   !$cuf kernel do(2) <<< (1,*), (kersize,1) >>>
   DO j=1,Ny
      DO i=1,Nx
         uold_d(i,j)=u_d(i,j)
      END DO
   END DO
   !$cuf kernel do(2) <<< (1,*), (kersize,1) >>>
   DO j=1,Ny
      DO i=1,Nx
         u_d(i,j)=sin(u_d(i,j))
      END DO
   END DO
   call cufftExecD2Z(planf,u_d,nonlinhat_d)
   !$cuf kernel do(2) <<< (1,*), (kersize,1) >>>
   DO j=1,Ny
      DO i=1,Nx/2+1
         nonlinhat_d(i,j)=scalemodes*( 0.25*(kx_d(i)*kx_d(i) + ky_d(j)*
            ky_d(j))&
              *(2.0d0*v_d(i,j)+vold_d(i,j))&
              +(2.0d0*v_d(i,j)-vold_d(i,j))/(dt*dt)&
              -nonlinhat_d(i,j) )&
              /(1/(dt*dt)-0.25*(kx_d(i)*kx_d(i) + ky_d(j)*ky_d(j)))
      END DO
   END DO
   !$cuf kernel do(2) <<< (1,*), (kersize,1) >>>
   DO j=1,Ny
      DO i=1,Nx/2+1
         vold_d(i,j)=v_d(i,j)
      END DO
   END DO
   !$cuf kernel do(2) <<< (1,*), (kersize,1) >>>
   DO j=1,Ny
      DO i=1,Nx/2+1
         v_d(i,j)=nonlinhat_d(i,j)/scalemodes
      END DO
   END DO
   call cufftExecZ2D(planb,nonlinhat_d,u_d)
   IF (mod(n,plotgap)==0) then
      plotnum=plotnum+1
      time(plotnum)=n*dt
      PRINT *,'time',n*dt
      u=u_d
      uold=uold_d
      ! savearray=REAL(u,kind(0d0)) ! disabled for benchmarking
      ! CALL savedata(Nx,Ny,plotnum,name_config,savearray)
      CALL enercalc(Nx,Ny,planfxy,planbxy,dt,enkin(plotnum),enstr(
         plotnum),&
```

```fortran
                enpot(plotnum),en(plotnum),kx,ky,temp1,temp2,u,uold)
        END IF
    END DO
    CALL system_clock(finish,count_rate)
    PRINT *,'Finished time stepping'
    u=u_d
    uold=uold_d
    ! compute energy at the end
    CALL enercalc(Nx,Ny,planfxy,planbxy,dt,enkin(plotnum+1),enstr(plotnum+1)
        ,&
        enpot(plotnum+1),en(plotnum+1),kx,ky,temp1,temp2,u,uold)

    PRINT*,'Program took ',&
        REAL(finish-start,kind(0d0))/REAL(count_rate,kind(0d0)),&
        'for Time stepping'
    CALL saveresults(Nt,plotgap,time(1:1+n/plotgap),en(1:1+n/plotgap+1),&
        enstr(1:1+n/plotgap+1),enkin(1:1+n/plotgap+1),enpot(1:1+n/plotgap
            +1))

    ! Save times at which output was made in text format
    PRINT *,'Saved data'

    call cufftDestroy(planf)
    call cufftDestroy(planb)
    PRINT *,'Destroy CUFFT Plans'
    call dfftw_destroy_plan_(planbxy)
    call dfftw_destroy_plan_(planfxy)
    PRINT *,'Destroy FFTW Plans'
    DEALLOCATE(kx,ky,x,y,u,uold,time,enkin,enstr,enpot,en,savearray,temp1,
        temp2,&
        stat=allocatestatus)
    IF (allocatestatus .ne. 0) STOP
    PRINT *,'Deallocated host arrays'
    DEALLOCATE(uold_d,vold_d,u_d,v_d,nonlinhat_d,&
        kx_d,ky_d,&
        stat=allocatestatus)
    IF (allocatestatus .ne. 0) STOP
    PRINT *,'Deallocated gpu arrays'
    PRINT *,'Program execution complete'
END PROGRAM sg2d
```

Listing A.6: An OpenACC Fortran program to solve the 2D sine-Gordon equation.

```fortran
!-------------------------------------------------------------------
!
!
! PURPOSE
!
! This program solves nonlinear sine-Gordon equation in 2 dimensions
! u_{tt}-u_{xx}-u_{yy}=-sin(u)
! using a second order implicit-explicit time stepping scheme.
!
! The boundary conditions are u(x=0,y)=u(2*Lx*\pi,y),
!       u(x,y=0)=u(x,y=2*Ly*\pi)
! The initial condition is set in initialdata.f90
!
! AUTHORS
!
! B. Cloutier, B.K. Muite, P. Rigge
! 4 June 2012
!
! .. Parameters ..
!  Nx                            = number of modes in x - power of 2 for
   FFT
!  Ny                            = number of modes in y - power of 2 for
   FFT
!  Nt                            = number of timesteps to take
!  plotgap                       = number of timesteps between plots
!  FFTW_IN_PLACE                 = value for FFTW input
!  FFTW_MEASURE                  = value for FFTW input
!  FFTW_EXHAUSTIVE               = value for FFTW input
!  FFTW_PATIENT                  = value for FFTW input
!  FFTW_ESTIMATE                 = value for FFTW input
!  FFTW_FORWARD                  = value for FFTW input
!  FFTW_BACKWARD                 = value for FFTW input
!  pi                            = 3.1415926535...
!  Lx                            = width of box in x direction
!  Ly                            = width of box in y direction
! .. Scalars ..
!  i                             = loop counter in x direction
!  j                             = loop counter in y direction
!  n                             = loop counter for timesteps direction
!  allocatestatus                = error indicator during allocation
!  start                         = variable to record start time of program
!  finish                        = variable to record end time of program
!  count_rate                    = variable for clock count rate
!  planfxy                       = Forward 2d fft plan  (FFTW)
!  planbxy                       = Backward 2d fft plan (FFTW)
!  planf                         = Forward 2d fft plan  (CUFFT)
!  planb                         = Backward 2d fft plan (CUFFT)
!  dt                            = timestep
!  ierr                          = error code
!  plotnum                       = number of plot
! .. Arrays ..
!  u                             = approximate solution
!  uold                          = approximate solution
!  v                             = Fourier transform of approximate
   solution
!  vold                          = Fourier transform of approximate
   solution
```

```
!  nonlinhat                      = Fourier transform of nonlinear term, sin
   (u)
!  temp1                          = extra space for energy computation
!  temp2                          = extra space for energy computation
!  savearray                      = temp array to save out to disk
! .. Vectors ..
!  kx                             = fourier frequencies in x direction
!  ky                             = fourier frequencies in y direction
!  x                              = x locations
!  y                              = y locations
!  time                           = times at which save data
!  en                             = total energy
!  enstr                          = strain energy
!  enpot                          = potential energy
!  enkin                          = kinetic energy
!  name_config                    = array to store filename for data to be
   saved
!
! REFERENCES
!
! ACKNOWLEDGEMENTS
!
! This program is based on example code to demonstrate usage of Fortran
   and
! CUDA FFT routines taken from
! http://cudamusing.blogspot.com/2010/05/CALLing-cufft-from-cuda-fortran.
   html
!
! and
!
! http://cudamusing.blogspot.com/search?q=cublas
!
! ACCURACY
!
! ERROR INDICATORS AND WARNINGS
!
! FURTHER COMMENTS
! Check that the initial iterate is consistent with the
! boundary conditions for the domain specified
!--------------------------------------------------------------------
! External routines required
!       getgrid.f90     -- Get initial grid of points
!       initialdata.f90 -- Get initial data
!       enercalc.f90    -- Subroutine to calculate the energy
!       savedata.f90    -- Save initial data
! External libraries required
!       Cuda FFT
!       OpenACC
!       FFTW3           -- Fastest Fourier Transform in the West
!                          (http://www.fftw.org/)
!       OpenMP
```

```
module precision
  ! Precision control
  integer, parameter, public :: Single = kind(0.0) ! Single precision
  integer, parameter, public :: Double = kind(0.0d0) ! Double precision
  !
  integer, parameter, public :: fp_kind = Double
  !integer, parameter, public :: fp_kind = Single
end module precision

module cufft
  integer, public :: CUFFT_FORWARD = -1
  integer, public :: CUFFT_INVERSE = 1
  integer, public :: CUFFT_R2C = Z'2a' ! Real to Complex (interleaved)
  integer, public :: CUFFT_C2R = Z'2c' ! Complex (interleaved) to Real
  integer, public :: CUFFT_C2C = Z'29' ! Complex to Complex, interleaved
  integer, public :: CUFFT_D2Z = Z'6a' ! Double to Double-Complex
  integer, public :: CUFFT_Z2D = Z'6c' ! Double-Complex to Double
  integer, public :: CUFFT_Z2Z = Z'69' ! Double-Complex to Double-Complex
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !
  ! cufftPlan2d(cufftHandle *plan, int nx,int ny, cufftType type,int batch
     )
  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  interface cufftPlan2d
     subroutine cufftPlan2d(plan, nx, ny, type) bind(C,name='cufftPlan2d')
       use iso_c_binding
       integer(c_int):: plan
       integer(c_int),value:: nx, ny, type
     end subroutine cufftPlan2d
  end interface cufftPlan2d
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !
  ! cufftDestroy(cufftHandle plan)
  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  interface cufftDestroy
     subroutine cufftDestroy(plan) bind(C,name='cufftDestroy')
       use iso_c_binding
       integer(c_int),value:: plan
     end subroutine cufftDestroy
  end interface cufftDestroy
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !
  ! cufftExecD2Z(cufftHandle plan,
  ! cufftDoubleReal    *idata,
  ! cufftDoubleComplex *odata)
  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  interface cufftExecD2Z
     subroutine cufftExecD2Z(plan, idata, odata) &
```

```
          & bind(C,name='cufftExecD2Z')
       use iso_c_binding
       use precision
       integer(c_int),  value   :: plan
       real(fp_kind),   device  :: idata(1:nx,1:ny)
       complex(fp_kind),device  :: odata(1:nx,1:ny)
     end subroutine cufftExecD2Z
  end interface cufftExecD2Z
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !
  ! cufftExecD2Z(cufftHandle plan,
  ! cufftDoubleComplex *idata,
  ! cufftDoubleReal    *odata)
  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  interface cufftExecZ2D
     subroutine cufftExecZ2D(plan, idata, odata) &
          & bind(C,name='cufftExecZ2D')
       use iso_c_binding
       use precision
       integer(c_int),value   :: plan
       complex(fp_kind),device:: idata(1:nx,1:ny)
       real(fp_kind),device   :: odata(1:nx,1:ny)
     end subroutine cufftExecZ2D
  end interface cufftExecZ2D
end module cufft

PROGRAM sg2d
  USE precision
  USE cufft
  USE openacc
  ! Declare variables
  IMPLICIT NONE
  INTEGER(kind=4), PARAMETER                           :: Nx=1024
  INTEGER(kind=4), PARAMETER                           :: Ny=Nx
  INTEGER(kind=4), PARAMETER                           :: Nt=500
  INTEGER(kind=4), PARAMETER                           :: plotgap=Nt+1
  REAL(kind=8), PARAMETER                              :: &
       pi=3.14159265358979323846264338327950288419716939937510d0
  REAL(kind=8), PARAMETER                              :: Lx=5.0d0
  REAL(kind=8), PARAMETER                              :: Ly=5.0d0
  REAL(kind=8)                                         :: dt=0.001d0
  COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE           :: kx,ky
  REAL(kind=8),          DIMENSION(:), ALLOCATABLE     :: x,y
  REAL   (kind=8), DIMENSION(:,:), ALLOCATABLE         :: u,uold
  COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE         :: temp1,temp2,v,
      vold,nonlinhat
  REAL(kind=8), DIMENSION(:,:), ALLOCATABLE            :: savearray
  REAL(kind=8), DIMENSION(:), ALLOCATABLE              :: time,enkin,enstr
      ,enpot,en
```

```
INTEGER(kind=4)                                              :: ierr,i,j,n,
   allocatestatus
INTEGER(kind=4)                                              :: start, finish,
   count_rate, plotnum
INTEGER(kind=4), PARAMETER                                   :: FFTW_IN_PLACE =
   8, FFTW_MEASURE = 0, &
     FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32, FFTW_ESTIMATE = 64
INTEGER(kind=4),PARAMETER                                    :: FFTW_FORWARD =
   -1, FFTW_BACKWARD=1
INTEGER(kind=8)                                              :: planfxy,planbxy
CHARACTER*100                                                :: name_config
INTEGER(kind=4)                                              :: planf,planb
! print run information
PRINT *,"Nx=", Nx
PRINT *,"Ny=", Ny
PRINT *,"Nt=", Nt
PRINT *,"Lx=", Lx
PRINT *,"Ly=", Ly
PRINT *,"dt=", dt
ALLOCATE(kx(1:Nx),ky(1:Ny),x(1:Nx),y(1:Ny),u(1:Nx,1:Ny),uold(1:Nx,1:Ny)
    ,&
     v(1:Nx/2+1,1:Ny),vold(1:Nx/2+1,1:Ny),nonlinhat(1:Nx/2+1,1:Ny),&
     savearray(1:Nx,1:Ny),time(1:1+Nt/plotgap),enkin(1:1+Nt/plotgap+1),&
     enstr(1:1+Nt/plotgap+1),enpot(1:1+Nt/plotgap+1),en(1:1+Nt/plotgap)
         ,&
     temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny),&
     stat=allocatestatus)
IF (allocatestatus .ne. 0) stop
PRINT *,'allocated arrays'
! set up cuda ffts
call cufftPlan2D(planf,nx,ny,CUFFT_D2Z)
call cufftPlan2D(planb,nx,ny,CUFFT_Z2D)
! set up fftw ffts
CALL dfftw_plan_dft_2d_(planfxy,Nx,Ny,u,temp2,FFTW_FORWARD,FFTW_ESTIMATE
   )
CALL dfftw_plan_dft_2d_(planbxy,Nx,Ny,temp2,u,FFTW_BACKWARD,
   FFTW_ESTIMATE)
PRINT *,'Setup FFTs'
! setup grid, wave numbers
!$acc data copy(x, y, kx, ky, vold, v, nonlinhat, uold, u)
!$acc kernels loop
DO i=1,1+Nx/2
   kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
END DO
!$acc end kernels
kx(1+Nx/2)=0.0d0
!$acc kernels loop
DO i = 1,Nx/2 -1
   kx(i+1+Nx/2)=-kx(1-i+Nx/2)
END DO
!$acc end kernels
```

```
!$acc kernels loop
DO i=1,Nx
   x(i)=(-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)))*pi*Lx
END DO
!$acc end kernels
!$acc kernels loop
DO j=1,1+Ny/2
   ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
END DO
!$acc end kernels
ky(1+Ny/2)=0.0d0
!$acc kernels loop
DO j = 1,Ny/2 -1
   ky(j+1+Ny/2)=-ky(1-j+Ny/2)
END DO
!$acc end kernels
!$acc kernels loop
DO j=1,Ny
   y(j)=(-1.0d0 + 2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)))*pi*Ly
END DO
!$acc end kernels
PRINT *,'Got grid and fourier frequencies'
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
      u(i,j)=0.5d0*exp(-1.0d0*(x(i)**2 +y(j)**2))
  END DO
END DO
!$acc end kernels
!$acc kernels loop
DO j=1,Ny
  DO i=1,Nx
    uold(i,j)=0.5d0*exp(-1.0d0*(x(i)**2 +y(j)**2))
  END DO
END DO
!$acc end kernels
savearray=REAL(u)
plotnum=1
name_config = 'data/u'
! CALL savedata(Nx,Ny,plotnum,name_config,savearray) ! disabled for
   benchmarking
PRINT *,'data saved'
!$acc end data
CALL enercalc(Nx,Ny,planfxy,planbxy,dt,enkin(plotnum),enstr(plotnum),&
     enpot(plotnum),en(plotnum),kx(1:Nx),ky(1:Ny),temp1,temp2,&
     u(1:Nx,1:Ny),uold(1:Nx,1:Ny))
!$acc data copy(x, y, kx, ky, vold, v, nonlinhat, uold, u)
call cufftExecD2Z(planf,u,v)
call cufftExecD2Z(planf,uold,vold)
PRINT *,'Got initial data, starting timestepping'
time(plotnum)=0.0d0
```

```fortran
CALL system_clock(start,count_rate)
DO n=1,Nt
   !$acc kernels loop
   DO j=1,Ny
      DO i=1,Nx
         uold(i,j)=u(i,j)
         u(i,j)=sin(u(i,j))
      END DO
   END DO
   !$acc end kernels
   call cufftExecD2Z(planf,u,nonlinhat)
   !$acc kernels loop
   DO j=1,Ny
      DO i=1,Nx/2+1
         nonlinhat(i,j)=( 0.25*(kx(i)*kx(i) + ky(j)*ky(j))&
               *(2.0d0*v(i,j)+vold(i,j))+(2.0d0*v(i,j)-vold(i,j))/(dt*dt)
                   &
               -nonlinhat(i,j) )/(1/(dt*dt)-0.25*(kx(i)*kx(i) + ky(j)*ky(
                   j)))
         vold(i,j)=v(i,j)
         v(i,j)=nonlinhat(i,j)
         ! prescale nonlinhat
         nonlinhat(i,j)=nonlinhat(i,j)/REAL(Nx*Ny,kind(0d0))
      END DO
   END DO
   !$acc end kernels
   call cufftExecZ2D(planb,nonlinhat,u)
END DO
CALL system_clock(finish,count_rate)
!$acc end data
PRINT *,'Finished time stepping'
! compute energy at the end
! savearray=REAL(u(1:Nx,1:Ny),kind(0d0)) ! disabled for benchmarking
! CALL savedata(Nx,Ny,plotnum+1,name_config,savearray)
CALL enercalc(Nx,Ny,planfxy,planbxy,dt,enkin(plotnum+1),enstr(plotnum+1)
    ,&
      enpot(plotnum+1),en(plotnum+1),kx,ky,temp1,temp2,u(1:Nx,1:Ny),uold
          (1:Nx,1:Ny))
PRINT*,'Program took ',&
      REAL(finish-start,kind(0d0))/REAL(count_rate,kind(0d0)),&
      'for Time stepping'
CALL saveresults(Nt,plotgap,time(1:1+n/plotgap),en(1:1+n/plotgap+1),&
      enstr(1:1+n/plotgap+1),enkin(1:1+n/plotgap+1),enpot(1:1+n/plotgap
          +1))

! Save times at which output was made in text format
PRINT *,'Saved data'

call cufftDestroy(planf)
call cufftDestroy(planb)
PRINT *,'Destroy CUFFT Plans'
```

```
  call dfftw_destroy_plan_(planbxy)
  call dfftw_destroy_plan_(planfxy)
  PRINT *,'Destroy FFTW Plans'
  DEALLOCATE(kx,ky,x,y,u,uold,time,enkin,enstr,enpot,en,savearray,temp1,
     temp2,&
       stat=allocatestatus)
  IF (allocatestatus .ne. 0) STOP
  PRINT *,'Deallocated host arrays'
  PRINT *,'Program execution complete'
END PROGRAM sg2d
```

Appendix B

Python Programs

Since Matlab requires a licence, we have also included Python versions of some of the Matlab programs. These programs have been tested in Python 2.7 (which can be obtained from `http://python.org/`), they also require Matlplotlib (version 1.10, which can be obtained from `http://matplotlib.sourceforge.net/index.html`), Mayavi (`http://github.enthought.com/mayavi/mayavi/index.html`) and numpy (`http://numpy.scipy.org/`). These programs have been tested primarily with the Enthought Python distribution.

Listing B.1: A Python program to demonstrate instability of different time-stepping methods. Compare this to the Matlab implementation in listing 5.1.

```python
#!/usr/bin/env python
"""
A program to demonstrate instability of timestepping methods#
when the timestep is inappropriately choosen.################
"""

from math import exp
import matplotlib.pyplot as plt
import numpy

#Differential equation: y'(t)=-l*y(t) y(t_0)=y_0
#Initial Condition, y(t_0)=1 where t_0=0

# Definition of the Grid
h = 0.1          # Time step size
t0 = 0           # Initial value
tmax = 4           # Value to be computed y(tmax)
Npoints = int((tmax-t0)/h)  # Number of points in the Grid

t = [t0]

# Initial data
l = 0.1
y0 = 1      # Initial condition y(t0)=y0
```

```
y_be = [y0]   # Variables holding the value given by the Backward Euler
    Iteration
y_fe = [y0]   # Variables holding the value given by the Forward Euler
    Iteration
y_imr = [y0]    # Variables holding the value given by the Midpoint Rule
    Iteration

for i in xrange(Npoints):
    y_fe.append(y_be[-1]*(1-l*h))
    y_be.append(y_fe[-1]/(1+l*h))
    y_imr.append(y_imr[-1]*(2-l*h)/(2+l*h))
    t.append(t[-1]+h)


print
print "Exact Value:          y(%d)=%f" % (tmax, exp(-4))
print "Backward Euler Value: y(%d)=%f" % (tmax, y_be[-1])
print "Forward Euler Value:  y(%d)=%f" % (tmax, y_fe[-1])
print "Midpoint Rule Value:  y(%d)=%f" % (tmax, y_imr[-1])

# Exact Solution
tt=numpy.arange(0,tmax,0.001)
exact = numpy.exp(-l*tt)

# Plot
plt.figure()
plt.plot(tt,exact,'r-',t,y_fe,'b:',t,y_be,'g--',t,y_imr,'k-.');
plt.xlabel('time')
plt.ylabel('y')
plt.legend(('Exact','Forward Euler','Backward Euler',
    'Implicit Midpoint Rule'))
plt.show()
```

Listing B.2: A Python program to solve the heat equation using forward Euler time-stepping. Compare this to the Matlab implementation in listing 8.1.

```
#!/usr/bin/env python
"""
Solving Heat Equation using pseudo-spectral and Forward Euler
u_t= \alpha*u_xx
BC= u(0)=0, u(2*pi)=0
IC=sin(x)
"""

import math
import numpy
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator
```

```

# Grid
N = 64                     # Number of steps
h = 2*math.pi/N                 # step size
x = h*numpy.arange(0,N)    # discretize x-direction

alpha = 0.5                # Thermal Diffusivity constant
t = 0
dt = .001

# Initial conditions
v = numpy.sin(x)
I = complex(0,1)
k = numpy.array([I*y for y in range(0,N/2) + [0] + range(-N/2+1,0)])
k2=k**2;

# Setting up Plot
tmax = 5; tplot = .1;
plotgap = int(round(tplot/dt))
nplots  = int(round(tmax/tplot))

data = numpy.zeros((nplots+1,N))
data[0,:] = v
tdata = [t]

for i in xrange(nplots):
    v_hat = numpy.fft.fft(v)

    for n in xrange(plotgap):
        v_hat = v_hat+dt*alpha*k2*v_hat # FE timestepping

    v = numpy.real(numpy.fft.ifft(v_hat))   # back to real space
    data[i+1,:] = v

    # real time vector
    t = t+plotgap*dt
    tdata.append(t)

# Plot using mesh
xx,tt = (numpy.mat(A) for A in (numpy.meshgrid(x,tdata)))

fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(xx, tt, data,rstride=1, cstride=1, cmap=cm.jet,
        linewidth=0, antialiased=False)
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.xlabel('x')
plt.ylabel('t')
plt.show()
```

Listing B.3: A Python program to solve the heat equation using backward Euler time-stepping. Compare this to the Matlab implementation in listing 8.2.

```python
#!/usr/bin/env python
"""
Solving Heat Equation using pseudospectral methods with Backwards Euler:
u_t= \alpha*u_xx
BC = u(0)=0 and u(2*pi)=0 (Periodic)
IC=sin(x)
"""

import math
import numpy
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator

# Grid
N = 64; h = 2*math.pi/N; x = [h*i for i in xrange(1,N+1)]

# Initial conditions
v = [math.sin(y) for y in x]
alpha = 0.5
t = 0
dt = .001 #Timestep size

# (ik)^2 Vector
I = complex(0,1)
k = numpy.array([I*n for n in range(0,N/2) + [0] + range(-N/2+1,0)])
k2=k**2;

# Setting up Plot
tmax = 5.0; tplot = 0.1
plotgap= int(round(tplot/dt))
nplots = int(round(tmax/tplot))
data = numpy.zeros((nplots+1,N))
data[0,:] = v
tdata = [t]

for i in xrange(nplots):
    v_hat = numpy.fft.fft(v)  # convert to fourier space
    for n in xrange(plotgap):
        v_hat = v_hat / (1-dt*alpha*k2) # backward Euler timestepping

    v = numpy.fft.ifft(v_hat)   # convert back to real space
    data[i+1,:] = numpy.real(v)   # records data

    t = t+plotgap*dt    # records real time
    tdata.append(t)

# Plot using mesh
```

```
xx,tt = (numpy.mat(A) for A in (numpy.meshgrid(x,tdata)))

fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(xx, tt, data,rstride=1, cstride=1, cmap=cm.jet,
        linewidth=0, antialiased=False)
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.xlabel('x')
plt.ylabel('t')
plt.show()
```

Listing B.4: A Python program to solve the 2D Allen Cahn equation using implicit explicit time-stepping. Compare this to the Matlab implementation in listing 8.4.

```
#!/usr/bin/env python
"""
Solving 2D Allen-Cahn Eq using pseudo-spectral with Implicit/Explicit
u_t= epsilon(u_{xx}+u_{yy}) + u - u^3
where u-u^3 is treated explicitly and u_{xx} and u_{yy} is treated
    implicitly
BC = Periodic
IC=v=sin(2*pi*x)+0.5*cos(4*pi*y)
"""

import math
import numpy
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator
import time

plt.ion()

# Setup the grid
N = 64; h = 1.0/N;
x = [h*i for i in xrange(1,N+1)]
y = [h*i for i in xrange(1,N+1)]
dt = 0.05
xx,yy = (numpy.mat(A) for A in (numpy.meshgrid(x,y)))

# Initial Conditions
u = numpy.array(numpy.sin(2*math.pi*xx) + 0.5*numpy.cos(4*math.pi*yy),
    dtype=float)

epsilon = 0.01

# (ik) and (ik)^2 vectors in x and y direction
I = complex(0,1)
k_x = numpy.array([I*n for n in range(0,N/2) + [0] + range(-N/2+1,0)])
```

```
k_y = k_x

kxx = numpy.zeros((N,N), dtype=complex)
kyy = numpy.zeros((N,N), dtype=complex)
for i in xrange(N):
    for j in xrange(N):
        kxx[i,j] = k_x[i]**2
        kyy[i,j] = k_y[j]**2

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(xx, yy, u,rstride=1, cstride=1, cmap=cm.jet,
        linewidth=0, antialiased=False)
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.xlabel('x')
plt.ylabel('y')
plt.show()

v_hat = numpy.zeros((N,N), dtype=complex)
v_hat = numpy.fft.fft2(u)

for n in xrange(100):
    # calculate nonlinear term in real space
    v_nl = numpy.array(u**3, dtype=complex)
  # FFT for nonlinear and linear terms
    v_nl = numpy.fft.fft2(v_nl)
    v_hat = (v_hat*(1+1/dt) - v_nl)
    v_hat=v_hat/(1/dt - (kxx+kyy)*epsilon)  # Implicit/Explicit
       timestepping
    u = numpy.real(numpy.fft.ifft2(v_hat))
    # Remove old plot before drawing
    ax.collections.remove(surf)
    surf = ax.plot_surface(xx, yy, u,rstride=1, cstride=1, cmap=cm.jet,
         linewidth=0, antialiased=False)
    plt.draw()
plt.show()
```

Listing B.5: A Python program to demonstrate fixed-point iteration. Compare this to the Matlab implementation in listing 9.1.

```
#!/usr/bin/env python
"""
A program to solve y'=y^2 using the backward Euler
method and fixed point iteration
This is not optimized and is very simple
"""

import time
import matplotlib.pyplot as plt

```

```
N = 1000     # Number of timesteps
tmax = 0.99   # Maximum time
y0 = 1
t0 = 0       # Initial value
tol = pow(0.1,10) # Tolerance for fixed point iterations
h = tmax/N    # Time step

y = [y0]     # Variables holding the values of iterations
t = [t0]     # Times of discrete solutions



T0 = time.clock()
for i in xrange(N):
  yold = y[i]
  ynew = y[i]
  err = 1
  while err > tol:
    ynew = h*pow(yold,2)+y[i]
    err = abs(ynew-yold)
    yold = ynew
  y.append(ynew)
  t.append(t[i]+h)

T = time.clock() - T0
yexact = [1.0/(1.0-x) for x in t]

print
print "Exact value:                 y(%d)=%f" % (tmax, 1/(1-tmax))
print "Value given by aproximation: y(%d)=%f" % (tmax, y[-1])
maxerr=(max([abs(y[i] - yexact[i]) for i in xrange(len(y))]))
print "Maximum error:               %f" % maxerr
print "Elapsed time is %f" % (T)

plt.figure()
plt.plot(t,y,'r+',t,yexact,'b-.')
plt.xlabel('Time')
plt.ylabel('Solution')
plt.legend(('Backward Euler', 'Exact solution'))
plt.title('Numerical solution of dy/dt=y^2')
plt.show()
```

Listing B.6: A Python program to demonstrate Newton iteration. Compare this to the Matlab implementation in listing 9.2.

```
#!/usr/bin/env python
"""
A program to solve y'=y^2 using the backward Euler
method and Newton iteration
This is not optimized and is very simple
```

```
"""

import time
import matplotlib.pyplot as plt

N = 1000               # Number of timesteps
tmax = 0.99            # Maximum value
t0 = 0                 # Initial t value
y0 = 1                 # Initial value y(t0) = y0
tol = 0.1**10          # Tolerance for fixed point iterations
h = (tmax - t0)/N      # Time step

y = [y0]               # List for discrete solutions
t = [t0]               # List with grid of discrete values of t

T0 = time.clock()          #Start timing

for i in xrange(N):
    yold = y[i]
    ynew = y[i]
    err =1
    while err > tol:
        ynew = yold-(yold-y[i]-h*(yold**2))/(1-2*h*yold)
        err = abs(ynew-yold)
        yold = ynew
    y.append(ynew)
    t.append(t[-1]+h)

T = time.clock() - T0    # Stop timing


print "Exact value y(%f) = %f " % (t[N], 1/(1-t[N]))
print "Value given by approx y_n(%f) = %f" % (t[N], y[N])
print "The error = y-y_n = %f " % (abs(1/(1-t[N]) - y[N]))
print "Time taken = %f " % (T)

yexact = [1.0/(1.0-x) for x in t]

plt.figure();
plt.plot(t,y,'r+',t,yexact,'b-.');
plt.xlabel('Time')
plt.ylabel('Solution')
plt.legend(('Backward Euler', 'Exact Solution'))
plt.title('Numerical solution of dy/dt=y^2')
plt.show()
```

Listing B.7: A Python program which uses Strang splitting to solve an ODE. Compare this to the Matlab implementation in listing ??.

```
"""
```

```
A program to solve u_t'=u(u-1) using a Strang
splitting method
"""

import time
import numpy
import matplotlib.pyplot as plt

Nt = 1000   # Number of timesteps
tmax = 1.0    # Maximum time
dt=tmax/Nt      # increment between times
u0 = 0.8        # Initial value
t0 = 0      # Starting time
u = [u0]    # Variables holding the values of iterations
t = [t0]    # Times of discrete solutions

T0 = time.clock()
for i in xrange(Nt):
  c = -1.0/u[i]
  utemp=-1.0/(c+0.5*dt)
  utemp2=utemp*numpy.exp(-dt)
  c = -1.0/utemp2
  unew=-1.0/(c+0.5*dt)
  u.append(unew)
  t.append(t[i]+dt)

T = time.clock() - T0
uexact = [4.0/(4.0+numpy.exp(tt)) for tt in t]

print
print "Elapsed time is %f" % (T)

plt.figure()
plt.plot(t,u,'r+',t,uexact,'b-.')
plt.xlabel('Time')
plt.ylabel('Solution')
plt.legend(('Numerical Solution', 'Exact solution'))
plt.title('Numerical solution of du/dt=u(u-1)')
plt.show()
```

Listing B.8: A Python program which uses Strang splitting to solve the one-dimensional nonlinear Schrödinger equation. Compare this to the Matlab implementation in listing 12.2.

```
"""
A program to solve the 1D Nonlinear Schrodinger equation using a
second order splitting method. The numerical solution is compared
to an exact soliton solution. The exact equation solved is
iu_t+u_{xx}+2|u|^2u=0

More information on visualization can be found on the Mayavi
```

```
website, in particular:
http://github.enthought.com/mayavi/mayavi/mlab.html
which was last checked on 6 April 2012

"""

import math
import numpy
import matplotlib.pyplot as plt
import time

plt.ion()

# Grid
Lx=16.0       # Period 2*pi*Lx
Nx=8192       # Number of harmonics
Nt=1000       # Number of time slices
tmax=1.0    # Maximum time
dt=tmax/Nt  # time step
plotgap=10  # time steps between plots
Es= -1.0     # focusing (+1) or defocusing (-1) parameter
numplots=Nt/plotgap  # number of plots to make

x = [i*2.0*math.pi*(Lx/Nx) for i in xrange(-Nx/2,1+Nx/2)]
k_x = (1.0/Lx)*numpy.array([complex(0,1)*n for n in range(0,Nx/2) \
+ [0] + range(-Nx/2+1,0)])

k2xm=numpy.zeros((Nx), dtype=float)
xx=numpy.zeros((Nx), dtype=float)

for i in xrange(Nx):
   k2xm[i] = numpy.real(k_x[i]**2)
   xx[i]=x[i]


# allocate arrays
usquared=numpy.zeros((Nx), dtype=float)
pot=numpy.zeros((Nx), dtype=float)
u=numpy.zeros((Nx), dtype=complex)
uexact=numpy.zeros((Nx), dtype=complex)
una=numpy.zeros((Nx), dtype=complex)
unb=numpy.zeros((Nx), dtype=complex)
v=numpy.zeros((Nx), dtype=complex)
vna=numpy.zeros((Nx), dtype=complex)
vnb=numpy.zeros((Nx), dtype=complex)
mass=numpy.zeros((Nx), dtype=complex)
test=numpy.zeros((numplots-1),dtype=float)
tdata=numpy.zeros((numplots-1), dtype=float)

t=0.0
u=4.0*numpy.exp(complex(0,1.0)*t)*\
   (numpy.cosh(3.0*xx)+3.0*numpy.exp(8.0*complex(0,1.0)*t)*numpy.cosh(xx))
      \
   /(numpy.cosh(4*xx)+4.0*numpy.cosh(2.0*xx)+3.0*numpy.cos(8.0*t))
uexact=u
v=numpy.fft.fftn(u)
usquared=abs(u)**2
fig =plt.figure()
ax = fig.add_subplot(311)
ax.plot(xx,numpy.real(u),'b-')
plt.xlabel('x')
plt.ylabel('real u')
ax = fig.add_subplot(312)
ax.plot(xx,numpy.imag(u),'b-')
plt.xlabel('x')
plt.ylabel('imaginary u')
ax = fig.add_subplot(313)
ax.plot(xx,abs(u-uexact),'b-')
plt.xlabel('x')
plt.ylabel('error')
plt.show()


# initial mass
usquared=abs(u)**2
mass=numpy.fft.fftn(usquared)
ma=numpy.real(mass[0])
ma0=ma
tdata[0]=t
plotnum=0
#solve pde and plot results
for nt in xrange(numplots-1):
    for n in xrange(plotgap):
        vna=v*numpy.exp(complex(0,0.5)*dt*k2xm)
        una=numpy.fft.ifftn(vna)
        usquared=2.0*abs(una)**2
        pot=Es*usquared
        unb=una*numpy.exp(complex(0,-1)*dt*pot)
        vnb=numpy.fft.fftn(unb)
        v=vnb*numpy.exp(complex(0,0.5)*dt*k2xm)
        u=numpy.fft.ifftn(v)
        t+=dt
    plotnum+=1
    usquared=abs(u)**2
    uexact = 4.0*numpy.exp(complex(0,1.0)*t)*\
      (numpy.cosh(3.0*xx)+3.0*numpy.exp(8.0*complex(0,1.0)*t)*numpy.cosh(
         xx))\
      /(numpy.cosh(4*xx)+4.0*numpy.cosh(2.0*xx)+3.0*numpy.cos(8.0*t))
    ax = fig.add_subplot(311)
    plt.cla()
    ax.plot(xx,numpy.real(u),'b-')
    plt.title(t)
    plt.xlabel('x')
    plt.ylabel('real u')
    ax = fig.add_subplot(312)
    plt.cla()
    ax.plot(xx,numpy.imag(u),'b-')
    plt.xlabel('x')
    plt.ylabel('imaginary u')
    ax = fig.add_subplot(313)
    plt.cla()
    ax.plot(xx,abs(u-uexact),'b-')
    plt.xlabel('x')
    plt.ylabel('error')
    plt.draw()
    mass=numpy.fft.fftn(usquared)
    ma=numpy.real(mass[0])
    test[plotnum-1]=numpy.log(abs(1-ma/ma0))
    print(test[plotnum-1])
    tdata[plotnum-1]=t

plt.ioff()
plt.show()
```

Listing B.9: A Python program which uses Strang splitting to solve the two-dimensional nonlinear Schrödinger equation. Compare this to the Matlab implementation in listing 12.3.

```
"""
A program to solve the 2D Nonlinear Schrodinger equation using a
second order splitting method

More information on visualization can be found on the Mayavi
website, in particular:
http://github.enthought.com/mayavi/mayavi/mlab.html
which was last checked on 6 April 2012

"""

import math
import numpy
from mayavi import mlab
import matplotlib.pyplot as plt
import time

# Grid
Lx=4.0      # Period 2*pi*Lx
Ly=4.0      # Period 2*pi*Ly
Nx=64       # Number of harmonics
Ny=64       # Number of harmonics
Nt=100      # Number of time slices
tmax=1.0    # Maximum time
dt=tmax/Nt  # time step
```

```
plotgap=10  # time steps between plots
Es= 1.0     # focusing (+1) or defocusing (-1) parameter
numplots=Nt/plotgap  # number of plots to make

x = [i*2.0*math.pi*(Lx/Nx) for i in xrange(-Nx/2,1+Nx/2)]
y = [i*2.0*math.pi*(Ly/Ny) for i in xrange(-Ny/2,1+Ny/2)]
k_x = (1.0/Lx)*numpy.array([complex(0,1)*n for n in range(0,Nx/2) \
+ [0] + range(-Nx/2+1,0)])
k_y = (1.0/Ly)*numpy.array([complex(0,1)*n for n in range(0,Ny/2) \
+ [0] + range(-Ny/2+1,0)])

k2xm=numpy.zeros((Nx,Ny), dtype=float)
k2ym=numpy.zeros((Nx,Ny), dtype=float)
xx=numpy.zeros((Nx,Ny), dtype=float)
yy=numpy.zeros((Nx,Ny), dtype=float)


for i in xrange(Nx):
    for j in xrange(Ny):
            k2xm[i,j] = numpy.real(k_x[i]**2)
            k2ym[i,j] = numpy.real(k_y[j]**2)
            xx[i,j]=x[i]
            yy[i,j]=y[j]


# allocate arrays
usquared=numpy.zeros((Nx,Ny), dtype=float)
pot=numpy.zeros((Nx,Ny), dtype=float)
u=numpy.zeros((Nx,Ny), dtype=complex)
una=numpy.zeros((Nx,Ny), dtype=complex)
unb=numpy.zeros((Nx,Ny), dtype=complex)
v=numpy.zeros((Nx,Ny), dtype=complex)
vna=numpy.zeros((Nx,Ny), dtype=complex)
vnb=numpy.zeros((Nx,Ny), dtype=complex)
mass=numpy.zeros((Nx,Ny), dtype=complex)
test=numpy.zeros((numplots-1),dtype=float)
tdata=numpy.zeros((numplots-1), dtype=float)

u=numpy.exp(-(xx**2 + yy**2 ))
v=numpy.fft.fftn(u)
usquared=abs(u)**2
src = mlab.surf(xx,yy,usquared,colormap='YlGnBu',warp_scale='auto')
mlab.scalarbar()
mlab.xlabel('x',object=src)
mlab.ylabel('y',object=src)
mlab.zlabel('abs(u)^2',object=src)

# initial mass
usquared=abs(u)**2
mass=numpy.fft.fftn(usquared)
ma=numpy.real(mass[0,0])
```

```
print(ma)
ma0=ma
t=0.0
tdata[0]=t
plotnum=0
#solve pde and plot results
for nt in xrange(numplots-1):
    for n in xrange(plotgap):
        vna=v*numpy.exp(complex(0,0.5)*dt*(k2xm+k2ym))
        una=numpy.fft.ifftn(vna)
        usquared=abs(una)**2
        pot=Es*usquared
        unb=una*numpy.exp(complex(0,-1)*dt*pot)
        vnb=numpy.fft.fftn(unb)
        v=vnb*numpy.exp(complex(0,0.5)*dt*(k2xm+k2ym) )
        u=numpy.fft.ifftn(v)
        t+=dt
    plotnum+=1
    usquared=abs(u)**2
    src.mlab_source.scalars = usquared
    mass=numpy.fft.fftn(usquared)
    ma=numpy.real(mass[0,0])
    test[plotnum-1]=numpy.log(abs(1-ma/ma0))
    print(test[plotnum-1])
    tdata[plotnum-1]=t

plt.figure()
plt.plot(tdata,test,'r-')
plt.title('Time Dependence of Change in Mass')
plt.show()
```

Listing B.10: A Python program which uses Strang splitting to solve the three-dimensional nonlinear Schrödinger equation. Compare this to the Matlab implementation in listing 12.4.

```
"""
A program to solve the 3D Nonlinear Schrodinger equation using a
second order splitting method

More information on visualization can be found on the Mayavi
website, in particular:
http://github.enthought.com/mayavi/mayavi/mlab.html
which was last checked on 6 April 2012

"""

import math
import numpy
from mayavi import mlab
import matplotlib.pyplot as plt
import time
```

```

# Grid
Lx=4.0      # Period 2*pi*Lx
Ly=4.0      # Period 2*pi*Ly
Lz=4.0      # Period 2*pi*Lz
Nx=64       # Number of harmonics
Ny=64       # Number of harmonics
Nz=64       # Number of harmonics
Nt=100      # Number of time slices
tmax=1.0    # Maximum time
dt=tmax/Nt  # time step
plotgap=10  # time steps between plots
Es= 1.0     # focusing (+1) or defocusing (-1) parameter
numplots=Nt/plotgap  # number of plots to make

x = [i*2.0*math.pi*(Lx/Nx) for i in xrange(-Nx/2,1+Nx/2)]
y = [i*2.0*math.pi*(Ly/Ny) for i in xrange(-Ny/2,1+Ny/2)]
z = [i*2.0*math.pi*(Lz/Nz) for i in xrange(-Nz/2,1+Nz/2)]
k_x = (1.0/Lx)*numpy.array([complex(0,1)*n for n in range(0,Nx/2) \
+ [0] + range(-Nx/2+1,0)])
k_y = (1.0/Ly)*numpy.array([complex(0,1)*n for n in range(0,Ny/2) \
+ [0] + range(-Ny/2+1,0)])
k_z = (1.0/Lz)*numpy.array([complex(0,1)*n for n in range(0,Nz/2) \
+ [0] + range(-Nz/2+1,0)])

k2xm=numpy.zeros((Nx,Ny,Nz), dtype=float)
k2ym=numpy.zeros((Nx,Ny,Nz), dtype=float)
k2zm=numpy.zeros((Nx,Ny,Nz), dtype=float)
xx=numpy.zeros((Nx,Ny,Nz), dtype=float)
yy=numpy.zeros((Nx,Ny,Nz), dtype=float)
zz=numpy.zeros((Nx,Ny,Nz), dtype=float)


for i in xrange(Nx):
    for j in xrange(Ny):
        for k in xrange(Nz):
            k2xm[i,j,k] = numpy.real(k_x[i]**2)
            k2ym[i,j,k] = numpy.real(k_y[j]**2)
            k2zm[i,j,k] = numpy.real(k_z[k]**2)
            xx[i,j,k]=x[i]
            yy[i,j,k]=y[j]
            zz[i,j,k]=z[k]


# allocate arrays
usquared=numpy.zeros((Nx,Ny,Nz), dtype=float)
pot=numpy.zeros((Nx,Ny,Nz), dtype=float)
u=numpy.zeros((Nx,Ny,Nz), dtype=complex)
una=numpy.zeros((Nx,Ny,Nz), dtype=complex)
unb=numpy.zeros((Nx,Ny,Nz), dtype=complex)
v=numpy.zeros((Nx,Ny,Nz), dtype=complex)
```

```
vna=numpy.zeros((Nx,Ny,Nz), dtype=complex)
vnb=numpy.zeros((Nx,Ny,Nz), dtype=complex)
mass=numpy.zeros((Nx,Ny,Nz), dtype=complex)
test=numpy.zeros((numplots-1),dtype=float)
tdata=numpy.zeros((numplots-1), dtype=float)

u=numpy.exp(-(xx**2 + yy**2 + zz**2))
v=numpy.fft.fftn(u)
usquared=abs(u)**2
src = mlab.pipeline.scalar_field(xx,yy,zz,usquared,colormap='YlGnBu')
mlab.pipeline.iso_surface(src, contours=[usquared.min()+0.1*usquared.ptp()
    , ],
   colormap='YlGnBu',opacity=0.85)
mlab.pipeline.iso_surface(src, contours=[usquared.max()-0.1*usquared.ptp()
    , ],
   colormap='YlGnBu',opacity=1.0)
mlab.pipeline.image_plane_widget(src,plane_orientation='z_axes',
                            slice_index=Nz/2,colormap='YlGnBu',
                            opacity=0.01)
mlab.pipeline.image_plane_widget(src,plane_orientation='y_axes',
                            slice_index=Ny/2,colormap='YlGnBu',
                            opacity=0.01)
mlab.pipeline.image_plane_widget(src,plane_orientation='x_axes',
                            slice_index=Nx/2,colormap='YlGnBu',
                            opacity=0.01)
mlab.scalarbar()
mlab.xlabel('x',object=src)
mlab.ylabel('y',object=src)
mlab.zlabel('z',object=src)

# initial mass
usquared=abs(u)**2
mass=numpy.fft.fftn(usquared)
ma=numpy.real(mass[0,0,0])
print(ma)
ma0=ma
t=0.0
tdata[0]=t
plotnum=0
#solve pde and plot results
for nt in xrange(numplots-1):
    for n in xrange(plotgap):
        vna=v*numpy.exp(complex(0,0.5)*dt*(k2xm+k2ym+k2zm))
        una=numpy.fft.ifftn(vna)
        usquared=abs(una)**2
        pot=Es*usquared
        unb=una*numpy.exp(complex(0,-1)*dt*pot)
        vnb=numpy.fft.fftn(unb)
        v=vnb*numpy.exp(complex(0,0.5)*dt*(k2xm+k2ym+k2zm) )
        u=numpy.fft.ifftn(v)
        t+=dt
    plotnum+=1
    usquared=abs(u)**2
    src.mlab_source.scalars = usquared
    mass=numpy.fft.fftn(usquared)
    ma=numpy.real(mass[0,0,0])
    test[plotnum-1]=numpy.log(abs(1-ma/ma0))
    print(test[plotnum-1])
    tdata[plotnum-1]=t

plt.figure()
plt.plot(tdata,test,'r-')
plt.title('Time Dependence of Change in Mass')
plt.show()
```

Listing B.11: A Python program which finds a numerical solution to the 2D Navier-Stokes equation. Compare this to the Matlab implementation in listing 13.1.

```
#!/usr/bin/env python
"""
Numerical solution of the 2D incompressible Navier-Stokes on a
Square Domain [0,1]x[0,1] unumpy.sing a Fourier pseudo-spectral method
and Crank-Nicolson timestepmath.ping. The numerical solution is compared
    to
the exact Taylor-Green Vortex solution of the Navier-Stokes equations

Periodic free-slip boundary conditions and Initial conditions:
    u(x,y,0)=sin(2*pi*x)cos(2*pi*y)
    v(x,y,0)=-cos(2*pi*x)sin(2*pi*y)
Analytical Solution:
    u(x,y,t)=sin(2*pi*x)cos(2*pi*y)exp(-8*pi^2*nu*t)
    v(x,y,t)=-cos(2*pi*x)sin(2*pi*y)exp(-8*pi^2*nu*t)
"""

import math
import numpy
import matplotlib.pyplot as plt
from mayavi import mlab
import time

# Grid
N=64; h=1.0/N
x = [h*i for i in xrange(1,N+1)]
y = [h*i for i in xrange(1,N+1)]
numpy.savetxt('x.txt',x)

xx=numpy.zeros((N,N), dtype=float)
yy=numpy.zeros((N,N), dtype=float)

for i in xrange(N):
    for j in xrange(N):
        xx[i,j] = x[i]
        yy[i,j] = y[j]


dt=0.0025; t=0.0; tmax=0.10
#nplots=int(tmax/dt)
Rey=1

u=numpy.zeros((N,N), dtype=float)
v=numpy.zeros((N,N), dtype=float)
u_y=numpy.zeros((N,N), dtype=float)
v_x=numpy.zeros((N,N), dtype=float)
omega=numpy.zeros((N,N), dtype=float)
# Initial conditions
for i in range(len(x)):
    for j in range(len(y)):
        u[i][j]=numpy.sin(2*math.pi*x[i])*numpy.cos(2*math.pi*y[j])
        v[i][j]=-numpy.cos(2*math.pi*x[i])*numpy.sin(2*math.pi*y[j])
        u_y[i][j]=-2*math.pi*numpy.sin(2*math.pi*x[i])*numpy.sin(2*math.pi
           *y[j])
        v_x[i][j]=2*math.pi*numpy.sin(2*math.pi*x[i])*numpy.sin(2*math.pi*
           y[j])
        omega[i][j]=v_x[i][j]-u_y[i][j]

src = mlab.imshow(xx,yy,omega,colormap='jet')
mlab.scalarbar(object=src)
mlab.xlabel('x',object=src)
mlab.ylabel('y',object=src)


# Wavenumber
k_x = 2*math.pi*numpy.array([complex(0,1)*n for n in range(0,N/2) \
+ [0] + range(-N/2+1,0)])
k_y=k_x

kx=numpy.zeros((N,N), dtype=complex)
ky=numpy.zeros((N,N), dtype=complex)
kxx=numpy.zeros((N,N), dtype=complex)
kyy=numpy.zeros((N,N), dtype=complex)

for i in xrange(N):
    for j in xrange(N):
        kx[i,j] = k_x[i]
        ky[i,j] = k_y[j]
        kxx[i,j] = k_x[i]**2
        kyy[i,j] = k_y[j]**2

tol=10**(-10)
psihat=numpy.zeros((N,N), dtype=complex)
omegahat=numpy.zeros((N,N), dtype=complex)
omegahatold=numpy.zeros((N,N), dtype=complex)
```

```
nlhat=numpy.zeros((N,N), dtype=complex)
nlhatold=numpy.zeros((N,N), dtype=complex)
dpsix=numpy.zeros((N,N), dtype=float)
dpsiy=numpy.zeros((N,N), dtype=float)
omegacheck=numpy.zeros((N,N), dtype=float)
omegaold=numpy.zeros((N,N), dtype=float)
temp=numpy.zeros((N,N), dtype=float)
omegahat=numpy.fft.fft2(omega)
nlhat=numpy.fft.fft2(u*numpy.fft.ifft2(omegahat*kx)+\
v*numpy.fft.ifft2(omegahat*ky))
while (t<=tmax):
    chg=1.0

    # Save old values
    uold=u
    vold=v
    omegaold=omega
    omegacheck=omega
    omegahatold = omegahat
    nlhatold=nlhat

    while(chg>tol):
        # nolinear {n+1,k}
        nlhat=numpy.fft.fft2(u*numpy.fft.ifft2(omegahat*kx)+\
        v*numpy.fft.ifft2(omegahat*ky))

        # Crank-Nicolson timestepmath.ping
        omegahat=((1/dt + 0.5*(1/Rey)*(kxx+kyy))*omegahatold \
        -0.5*(nlhatold+nlhat)) \
        /(1/dt -0.5*(1/Rey)*(kxx+kyy))

        psihat=-omegahat/(kxx+kyy)
        psihat[0][0]=0
        psihat[N/2][N/2]=0
        psihat[N/2][0]=0
        psihat[0][N/2]=0

        dpsix = numpy.real(numpy.fft.ifft2(psihat*kx))
        dpsiy = numpy.real(numpy.fft.ifft2(psihat*ky))
        u=dpsiy
        v=-1.0*dpsix

        omega=numpy.real(numpy.fft.ifft2(omegahat))
        temp=abs(omega-omegacheck)
        chg=numpy.max(temp)
        print(chg)
        omegacheck=omega
    t+=dt
    src.mlab_source.scalars = omega

omegaexact=numpy.zeros((N,N), dtype=float)
```

```
for i in range(len(x)):
    for j in range(len(y)):
        uexact_y=-2*math.pi*numpy.sin(2*math.pi*x[i])*numpy.sin(2*math.pi*
            x[j])\
        *numpy.exp(-8*(math.pi**2)*t/Rey)
        vexact_x=2*math.pi*numpy.sin(2*math.pi*x[i])*numpy.sin(2*math.pi*y
            [j])\
        *numpy.exp(-8*(math.pi**2)*t/Rey)
        omegaexact[i][j]=vexact_x-uexact_y
numpy.savetxt('Error.txt',abs(omegaexact-omega))
```

Listing B.12: A Python program to solve the one-dimensional Klein Gordon equation (14.1) using the time discretization in eq. (14.3). Compare this to the Matlab implementation in listing 14.1.

```
"""
A program to solve the 1D Klein Gordon equation using a
second order semi-explicit method. The numerical solution is
compared to an exact solution

More information on visualization can be found on the Mayavi
website, in particular:
http://github.enthought.com/mayavi/mayavi/mlab.html
which was last checked on 6 April 2012

"""

import math
import numpy
import matplotlib.pyplot as plt
import time

plt.ion()

# Grid
Lx=64.0     # Period 2*pi*Lx
Nx=4096     # Number of harmonics
Nt=500        # Number of time slices
tmax=5.0      # Maximum time
c=0.5    # Wave speed
dt=tmax/Nt    # time step
plotgap=10    # time steps between plots
Es= 1.0       # focusing (+1) or defocusing (-1) parameter
numplots=Nt/plotgap   # number of plots to make

x = [i*2.0*math.pi*(Lx/Nx) for i in xrange(-Nx/2,1+Nx/2)]
k_x = (1.0/Lx)*numpy.array([complex(0,1)*n for n in range(0,Nx/2) \
+ [0] + range(-Nx/2+1,0)])

kxm=numpy.zeros((Nx), dtype=complex)
```

```
xx=numpy.zeros((Nx), dtype=float)

for i in xrange(Nx):
        kxm[i] = k_x[i]
        xx[i] = x[i]


# allocate arrays
unew=numpy.zeros((Nx), dtype=float)
u=numpy.zeros((Nx), dtype=float)
uexact=numpy.zeros((Nx), dtype=float)
uold=numpy.zeros((Nx), dtype=float)
vnew=numpy.zeros((Nx), dtype=complex)
v=numpy.zeros((Nx), dtype=complex)
vold=numpy.zeros((Nx), dtype=complex)
ux=numpy.zeros((Nx), dtype=float)
vx=numpy.zeros((Nx), dtype=complex)
Kineticenergy=numpy.zeros((Nx), dtype=complex)
Potentialenergy=numpy.zeros((Nx), dtype=complex)
Strainenergy=numpy.zeros((Nx), dtype=complex)
EnKin=numpy.zeros((numplots), dtype=float)
EnPot=numpy.zeros((numplots), dtype=float)
EnStr=numpy.zeros((numplots), dtype=float)
En=numpy.zeros((numplots), dtype=float)
Enchange=numpy.zeros((numplots-1),dtype=float)
tdata=numpy.zeros((numplots), dtype=float)
nonlin=numpy.zeros((Nx), dtype=float)
nonlinhat=numpy.zeros((Nx), dtype=complex)

t=0.0
u=numpy.sqrt(2)/(numpy.cosh((xx-c*t)/numpy.sqrt(1.0-c**2)))
uexact=numpy.sqrt(2)/(numpy.cosh((xx-c*t)/numpy.sqrt(1.0-c**2)))
uold=numpy.sqrt(2)/(numpy.cosh((xx+c*dt)/numpy.sqrt(1.0-c**2)))
v=numpy.fft.fftn(u)
vold=numpy.fft.fftn(uold)
fig=plt.figure()
ax=fig.add_subplot(211)
ax.plot(xx,u,'b-')
plt.xlabel('x')
plt.ylabel('u')
ax=fig.add_subplot(212)
ax.plot(xx,abs(u-uexact),'b-')
plt.xlabel('x')
plt.ylabel('error')
plt.show()
# initial energy
vx=0.5*kxm*(v+vold)
ux=numpy.real(numpy.fft.ifftn(vx))
Kineticenergy=0.5*((u-uold)/dt)**2
Strainenergy=0.5*(ux)**2
Potentialenergy=0.5*(0.5*(u+uold))**2 - Es*0.25*(0.5*(u+uold))**4
```

```
Kineticenergy=numpy.fft.fftn(Kineticenergy)
Strainenergy=numpy.fft.fftn(Strainenergy)
Potentialenergy=numpy.fft.fftn(Potentialenergy)
EnKin[0]=numpy.real(Kineticenergy[0])
EnPot[0]=numpy.real(Potentialenergy[0])
EnStr[0]=numpy.real(Strainenergy[0])
En[0]=EnStr[0]+EnPot[0]+EnKin[0]
En0=En[0]
tdata[0]=t
plotnum=0
#solve pde and plot results
for nt in xrange(numplots-1):
    for n in xrange(plotgap):
        nonlin=u**3
        nonlinhat=numpy.fft.fftn(nonlin)
        vnew=( (0.25*(kxm**2  - 1)*(2*v+vold)
          +(2*v-vold)/(dt*dt) +Es*nonlinhat)/
          (1/(dt*dt) - (kxm**2  -1)*0.25 ) )
        unew=numpy.real(numpy.fft.ifftn(vnew))
        t+=dt
        # update old terms
        vold=v
        v=vnew
        uold=u
        u=unew
    plotnum+=1
    uexact=numpy.sqrt(2)/(numpy.cosh((xx-c*t)/numpy.sqrt(1.0-c**2)))
    ax = fig.add_subplot(211)
    plt.cla()
    ax.plot(xx,u,'b-')
    plt.title(t)
    plt.xlabel('x')
    plt.ylabel('u')
    ax = fig.add_subplot(212)
    plt.cla()
    ax.plot(xx,abs(u-uexact),'b-')
    plt.xlabel('x')
    plt.ylabel('error')
    plt.draw()
    vx=0.5*kxm*(v+vold)
    ux=numpy.real(numpy.fft.ifftn(vx))
    Kineticenergy=0.5*((u-uold)/dt)**2
    Strainenergy=0.5*(ux)**2
    Potentialenergy=0.5*(0.5*(u+uold))**2 - Es*0.25*(0.5*(u+uold))**4
    Kineticenergy=numpy.fft.fftn(Kineticenergy)
    Strainenergy=numpy.fft.fftn(Strainenergy)
    Potentialenergy=numpy.fft.fftn(Potentialenergy)
    EnKin[plotnum]=numpy.real(Kineticenergy[0])
    EnPot[plotnum]=numpy.real(Potentialenergy[0])
    EnStr[plotnum]=numpy.real(Strainenergy[0])
    En[plotnum]=EnStr[plotnum]+EnPot[plotnum]+EnKin[plotnum]
    Enchange[plotnum-1]=numpy.log(abs(1-En[plotnum]/En0))
    tdata[plotnum]=t

plt.ioff()

plt.figure()
plt.plot(tdata,En,'r+',tdata,EnKin,'b:',tdata,EnPot,'g-.',tdata,EnStr,'y--')
plt.xlabel('Time')
plt.ylabel('Energy')
plt.legend(('Total', 'Kinetic','Potential','Strain'))
plt.title('Time Dependence of Energy Components')
plt.show()

plt.figure()
plt.plot(Enchange,'r-')
plt.title('Time Dependence of Change in Total Energy')
plt.show()
```

Listing B.13: A Python program to solve the one-dimensional Klein Gordon equation (14.1) using the time discretization in eq. (14.4). Compare this to the Matlab implementation in listing 14.2.

```
"""
A program to solve the 1D Klein Gordon equation using a
second order semi-explicit method. The numerical solution is
compared to an exact solution

More information on visualization can be found on the Mayavi
website, in particular:
http://github.enthought.com/mayavi/mayavi/mlab.html
which was last checked on 6 April 2012

"""

import math
import numpy
import matplotlib.pyplot as plt
import time

plt.ion()

# Grid
Lx=64.0     # Period 2*pi*Lx
Nx=4096     # Number of harmonics
Nt=500        # Number of time slices
tmax=5.0      # Maximum time
c=0.5    # Wave speed
dt=tmax/Nt    # time step
plotgap=10    # time steps between plots
```

```
Es= 1.0      # focusing (+1) or defocusing (-1) parameter
numplots=Nt/plotgap  # number of plots to make
tol=0.1**12  # tolerance for fixed point iterations

x = [i*2.0*math.pi*(Lx/Nx) for i in xrange(-Nx/2,1+Nx/2)]
k_x = (1.0/Lx)*numpy.array([complex(0,1)*n for n in range(0,Nx/2) \
+ [0] + range(-Nx/2+1,0)])

kxm=numpy.zeros((Nx), dtype=complex)
xx=numpy.zeros((Nx), dtype=float)

for i in xrange(Nx):
        kxm[i] = k_x[i]
        xx[i] = x[i]

# allocate arrays
unew=numpy.zeros((Nx), dtype=float)
u=numpy.zeros((Nx), dtype=float)
utemp=numpy.zeros((Nx), dtype=float)
uexact=numpy.zeros((Nx), dtype=float)
uold=numpy.zeros((Nx), dtype=float)
vnew=numpy.zeros((Nx), dtype=complex)
v=numpy.zeros((Nx), dtype=complex)
vold=numpy.zeros((Nx), dtype=complex)
ux=numpy.zeros((Nx), dtype=float)
vx=numpy.zeros((Nx), dtype=complex)
Kineticenergy=numpy.zeros((Nx), dtype=complex)
Potentialenergy=numpy.zeros((Nx), dtype=complex)
Strainenergy=numpy.zeros((Nx), dtype=complex)
EnKin=numpy.zeros((numplots), dtype=float)
EnPot=numpy.zeros((numplots), dtype=float)
EnStr=numpy.zeros((numplots), dtype=float)
En=numpy.zeros((numplots), dtype=float)
Enchange=numpy.zeros((numplots-1),dtype=float)
tdata=numpy.zeros((numplots), dtype=float)
nonlin=numpy.zeros((Nx), dtype=float)
nonlinhat=numpy.zeros((Nx), dtype=complex)

t=0.0
u=numpy.sqrt(2)/(numpy.cosh((xx-c*t)/numpy.sqrt(1.0-c**2)))
uexact=numpy.sqrt(2)/(numpy.cosh((xx-c*t)/numpy.sqrt(1.0-c**2)))
uold=numpy.sqrt(2)/(numpy.cosh((xx+c*dt)/numpy.sqrt(1.0-c**2)))
v=numpy.fft.fftn(u)
vold=numpy.fft.fftn(uold)
fig=plt.figure()
ax=fig.add_subplot(211)
ax.plot(xx,u,'b-')
plt.xlabel('x')
plt.ylabel('u')
ax=fig.add_subplot(212)
ax.plot(xx,abs(u-uexact),'b-')
```

```python
plt.xlabel('x')
plt.ylabel('error')
plt.show()
# initial energy
vx=0.5*kxm*(v+vold)
ux=numpy.real(numpy.fft.ifftn(vx))
Kineticenergy=0.5*((u-uold)/dt)**2
Strainenergy=0.5*(ux)**2
Potentialenergy=0.5*(0.5*(u+uold))**2 - Es*0.25*(0.5*(u+uold))**4
Kineticenergy=numpy.fft.fftn(Kineticenergy)
Strainenergy=numpy.fft.fftn(Strainenergy)
Potentialenergy=numpy.fft.fftn(Potentialenergy)
EnKin[0]=numpy.real(Kineticenergy[0])
EnPot[0]=numpy.real(Potentialenergy[0])
EnStr[0]=numpy.real(Strainenergy[0])
En[0]=EnStr[0]+EnPot[0]+EnKin[0]
En0=En[0]
tdata[0]=t
plotnum=0
#solve pde and plot results
for nt in xrange(numplots-1):
    for n in xrange(plotgap):
        nonlin=(u**2+uold**2)*(u+uold)/4.0
        nonlinhat=numpy.fft.fftn(nonlin)
        chg=1
        unew=u
        while (chg>tol):
            utemp=unew
            vnew=( (0.25*(kxm**2  - 1)*(2*v+vold)\
                 +(2*v-vold)/(dt*dt) +Es*nonlinhat)\
                 /(1/(dt*dt) - (kxm**2  -1)*0.25 ) )
            unew=numpy.real(numpy.fft.ifftn(vnew))
            nonlin=(unew**2+uold**2)*(unew+uold)/4.0
            nonlinhat=numpy.fft.fftn(nonlin)
            chg=numpy.max(abs(unew-utemp))
        t+=dt
        # update old terms
        vold=v
        v=vnew
        uold=u
        u=unew
    plotnum+=1
    uexact=numpy.sqrt(2)/(numpy.cosh((xx-c*t)/numpy.sqrt(1.0-c**2)))
    ax = fig.add_subplot(211)
    plt.cla()
    ax.plot(xx,u,'b-')
    plt.title(t)
    plt.xlabel('x')
    plt.ylabel('u')
    ax = fig.add_subplot(212)
    plt.cla()
    ax.plot(xx,abs(u-uexact),'b-')
    plt.xlabel('x')
    plt.ylabel('error')
    plt.draw()
    vx=0.5*kxm*(v+vold)
    ux=numpy.real(numpy.fft.ifftn(vx))
    Kineticenergy=0.5*((u-uold)/dt)**2
    Strainenergy=0.5*(ux)**2
    Potentialenergy=0.5*(0.5*(u+uold))**2 - Es*0.25*(0.5*(u+uold))**4
    Kineticenergy=numpy.fft.fftn(Kineticenergy)
    Strainenergy=numpy.fft.fftn(Strainenergy)
    Potentialenergy=numpy.fft.fftn(Potentialenergy)
    EnKin[plotnum]=numpy.real(Kineticenergy[0])
    EnPot[plotnum]=numpy.real(Potentialenergy[0])
    EnStr[plotnum]=numpy.real(Strainenergy[0])
    En[plotnum]=EnStr[plotnum]+EnPot[plotnum]+EnKin[plotnum]
    Enchange[plotnum-1]=numpy.log(abs(1-En[plotnum]/En0))
    tdata[plotnum]=t

plt.ioff()

plt.figure()
plt.plot(tdata,En,'r+',tdata,EnKin,'b:',tdata,EnPot,'g-.',tdata,EnStr,'y--
    ')
plt.xlabel('Time')
plt.ylabel('Energy')
plt.legend(('Total', 'Kinetic','Potential','Strain'))
plt.title('Time Dependence of Energy Components')
plt.show()

plt.figure()
plt.plot(Enchange,'r-')
plt.title('Time Dependence of Change in Total Energy')
plt.show()
```

Listing B.14: A Python program to solve the two-dimensional Klein Gordon equation (14.1) using the time discretization in eq. (14.4). Compare this to the Matlab implementation in listing 14.3.

```
#!/usr/bin/env python
"""
A program to solve the 2D Klein Gordon equation using a
second order semi-explicit method

More information on visualization can be found on the Mayavi
website, in particular:
http://github.enthought.com/mayavi/mayavi/mlab.html
which was last checked on 6 April 2012

"""
```

```

import math
import numpy
from mayavi import mlab
import matplotlib.pyplot as plt
import time


# Grid
Lx=3.0       # Period 2*pi*Lx
Ly=3.0       # Period 2*pi*Ly
Nx=512       # Number of harmonics
Ny=512       # Number of harmonics
Nt=200       # Number of time slices
tmax=5.0    # Maximum time
dt=tmax/Nt   # time step
plotgap=10   # time steps between plots
Es= 1.0     # focusing (+1) or defocusing (-1) parameter
numplots=Nt/plotgap  # number of plots to make

x = [i*2.0*math.pi*(Lx/Nx) for i in xrange(-Nx/2,1+Nx/2)]
y = [i*2.0*math.pi*(Ly/Ny) for i in xrange(-Ny/2,1+Ny/2)]
k_x = (1.0/Lx)*numpy.array([complex(0,1)*n for n in range(0,Nx/2) \
+ [0] + range(-Nx/2+1,0)])
k_y = (1.0/Ly)*numpy.array([complex(0,1)*n for n in range(0,Ny/2) \
+ [0] + range(-Ny/2+1,0)])

kxm=numpy.zeros((Nx,Ny), dtype=complex)
kym=numpy.zeros((Nx,Ny), dtype=complex)
xx=numpy.zeros((Nx,Ny), dtype=float)
yy=numpy.zeros((Nx,Ny), dtype=float)


for i in xrange(Nx):
    for j in xrange(Ny):
        kxm[i,j] = k_x[i]
        kym[i,j] = k_y[j]
        xx[i,j] = x[i]
        yy[i,j] = y[j]


# allocate arrays
unew=numpy.zeros((Nx,Ny), dtype=float)
u=numpy.zeros((Nx,Ny), dtype=float)
uold=numpy.zeros((Nx,Ny), dtype=float)
vnew=numpy.zeros((Nx,Ny), dtype=complex)
v=numpy.zeros((Nx,Ny), dtype=complex)
vold=numpy.zeros((Nx,Ny), dtype=complex)
ux=numpy.zeros((Nx,Ny), dtype=float)
uy=numpy.zeros((Nx,Ny), dtype=float)
vx=numpy.zeros((Nx,Ny), dtype=complex)
```

```
vy=numpy.zeros((Nx,Ny), dtype=complex)
Kineticenergy=numpy.zeros((Nx,Ny), dtype=complex)
Potentialenergy=numpy.zeros((Nx,Ny), dtype=complex)
Strainenergy=numpy.zeros((Nx,Ny), dtype=complex)
EnKin=numpy.zeros((numplots), dtype=float)
EnPot=numpy.zeros((numplots), dtype=float)
EnStr=numpy.zeros((numplots), dtype=float)
En=numpy.zeros((numplots), dtype=float)
Enchange=numpy.zeros((numplots-1),dtype=float)
tdata=numpy.zeros((numplots), dtype=float)
nonlin=numpy.zeros((Nx,Ny), dtype=float)
nonlinhat=numpy.zeros((Nx,Ny), dtype=complex)

u=0.1*numpy.exp(-(xx**2 + yy**2))*numpy.sin(10*xx+12*yy)
uold=u
v=numpy.fft.fft2(u)
vold=numpy.fft.fft2(uold)
src = mlab.surf(xx,yy,u,colormap='YlGnBu',warp_scale='auto')
mlab.scalarbar(object=src)
mlab.xlabel('x',object=src)
mlab.ylabel('y',object=src)
mlab.zlabel('u',object=src)
# initial energy
vx=0.5*kxm*(v+vold)
vy=0.5*kym*(v+vold)
ux=numpy.fft.ifft2(vx)
uy=numpy.fft.ifft2(vy)
Kineticenergy=0.5*((u-uold)/dt)**2
Strainenergy=0.5*(ux)**2 + 0.5*(uy)**2
Potentialenergy=0.5*(0.5*(u+uold))**2 - Es*0.25*(0.5*(u+uold))**4
Kineticenergy=numpy.fft.fft2(Kineticenergy)
Strainenergy=numpy.fft.fft2(Strainenergy)
Potentialenergy=numpy.fft.fft2(Potentialenergy)
EnKin[0]=numpy.real(Kineticenergy[0,0])
EnPot[0]=numpy.real(Potentialenergy[0,0])
EnStr[0]=numpy.real(Strainenergy[0,0])
En[0]=EnStr[0]+EnPot[0]+EnKin[0]
En0=En[0]
t=0.0
tdata[0]=t
plotnum=0
#solve pde and plot results
for nt in xrange(numplots-1):
    for n in xrange(plotgap):
        nonlin=u**3
        nonlinhat=numpy.fft.fft2(nonlin)
        vnew=( (0.25*(kxm**2 + kym**2 - 1)*(2*v+vold)
          +(2*v-vold)/(dt*dt) +Es*nonlinhat)/
          (1/(dt*dt) - (kxm**2 + kym**2 -1)*0.25 ) )
        unew=numpy.real(numpy.fft.ifft2(vnew))
        t+=dt
        # update old terms
        vold=v
        v=vnew
        uold=u
        u=unew
    plotnum+=1
    src.mlab_source.scalars = unew
    vx=0.5*kxm*(v+vold)
    vy=0.5*kym*(v+vold)
    ux=numpy.fft.ifft2(vx)
    uy=numpy.fft.ifft2(vy)
    Kineticenergy=0.5*((u-uold)/dt)**2
    Strainenergy=0.5*(ux)**2 + 0.5*(uy)**2
    Potentialenergy=0.5*(0.5*(u+uold))**2 - Es*0.25*(0.5*(u+uold))**4
    Kineticenergy=numpy.fft.fft2(Kineticenergy)
    Strainenergy=numpy.fft.fft2(Strainenergy)
    Potentialenergy=numpy.fft.fft2(Potentialenergy)
    EnKin[plotnum]=numpy.real(Kineticenergy[0,0])
    EnPot[plotnum]=numpy.real(Potentialenergy[0,0])
    EnStr[plotnum]=numpy.real(Strainenergy[0,0])
    En[plotnum]=EnStr[plotnum]+EnPot[plotnum]+EnKin[plotnum]
    Enchange[plotnum-1]=numpy.log(abs(1-En[plotnum]/En0))
    tdata[plotnum]=t


plt.figure()
plt.plot(tdata,En,'r+',tdata,EnKin,'b:',tdata,EnPot,'g-.',tdata,EnStr,'y--
    ')
plt.xlabel('Time')
plt.ylabel('Energy')
plt.legend(('Total', 'Kinetic','Potential','Strain'))
plt.title('Time Dependence of Energy Components')
plt.show()

plt.figure()
plt.plot(Enchange,'r-')
plt.title('Time Dependence of Change in Total Energy')
plt.show()
```

Listing B.15: A Python program to solve the two-dimensional Klein Gordon equation (14.1) using the time discretization in eq. (14.4). Compare this to the Matlab implementation in listing 14.3.

```
#!/usr/bin/env python
"""
A program to solve the 2D Klein Gordon equation using a
second order semi-explicit method

More information on visualization can be found on the Mayavi
website, in particular:
```

```
http://github.enthought.com/mayavi/mayavi/mlab.html
which was last checked on 6 April 2012

"""

import math
import numpy
from mayavi import mlab
import matplotlib.pyplot as plt
import time


# Grid
Lx=3.0       # Period 2*pi*Lx
Ly=3.0       # Period 2*pi*Ly
Nx=512       # Number of harmonics
Ny=512       # Number of harmonics
Nt=200       # Number of time slices
tmax=5.0    # Maximum time
dt=tmax/Nt  # time step
plotgap=10  # time steps between plots
Es= 1.0     # focusing (+1) or defocusing (-1) parameter
numplots=Nt/plotgap  # number of plots to make

x = [i*2.0*math.pi*(Lx/Nx) for i in xrange(-Nx/2,1+Nx/2)]
y = [i*2.0*math.pi*(Ly/Ny) for i in xrange(-Ny/2,1+Ny/2)]
k_x = (1.0/Lx)*numpy.array([complex(0,1)*n for n in range(0,Nx/2) \
+ [0] + range(-Nx/2+1,0)])
k_y = (1.0/Ly)*numpy.array([complex(0,1)*n for n in range(0,Ny/2) \
+ [0] + range(-Ny/2+1,0)])

kxm=numpy.zeros((Nx,Ny), dtype=complex)
kym=numpy.zeros((Nx,Ny), dtype=complex)
xx=numpy.zeros((Nx,Ny), dtype=float)
yy=numpy.zeros((Nx,Ny), dtype=float)


for i in xrange(Nx):
    for j in xrange(Ny):
        kxm[i,j] = k_x[i]
        kym[i,j] = k_y[j]
        xx[i,j] = x[i]
        yy[i,j] = y[j]


# allocate arrays
unew=numpy.zeros((Nx,Ny), dtype=float)
u=numpy.zeros((Nx,Ny), dtype=float)
uold=numpy.zeros((Nx,Ny), dtype=float)
vnew=numpy.zeros((Nx,Ny), dtype=complex)
v=numpy.zeros((Nx,Ny), dtype=complex)
```

```
vold=numpy.zeros((Nx,Ny), dtype=complex)
ux=numpy.zeros((Nx,Ny), dtype=float)
uy=numpy.zeros((Nx,Ny), dtype=float)
vx=numpy.zeros((Nx,Ny), dtype=complex)
vy=numpy.zeros((Nx,Ny), dtype=complex)
Kineticenergy=numpy.zeros((Nx,Ny), dtype=complex)
Potentialenergy=numpy.zeros((Nx,Ny), dtype=complex)
Strainenergy=numpy.zeros((Nx,Ny), dtype=complex)
EnKin=numpy.zeros((numplots), dtype=float)
EnPot=numpy.zeros((numplots), dtype=float)
EnStr=numpy.zeros((numplots), dtype=float)
En=numpy.zeros((numplots), dtype=float)
Enchange=numpy.zeros((numplots-1),dtype=float)
tdata=numpy.zeros((numplots), dtype=float)
nonlin=numpy.zeros((Nx,Ny), dtype=float)
nonlinhat=numpy.zeros((Nx,Ny), dtype=complex)

u=0.1*numpy.exp(-(xx**2 + yy**2))*numpy.sin(10*xx+12*yy)
uold=u
v=numpy.fft.fft2(u)
vold=numpy.fft.fft2(uold)
src = mlab.surf(xx,yy,u,colormap='YlGnBu',warp_scale='auto')
mlab.scalarbar(object=src)
mlab.xlabel('x',object=src)
mlab.ylabel('y',object=src)
mlab.zlabel('u',object=src)
# initial energy
vx=0.5*kxm*(v+vold)
vy=0.5*kym*(v+vold)
ux=numpy.fft.ifft2(vx)
uy=numpy.fft.ifft2(vy)
Kineticenergy=0.5*((u-uold)/dt)**2
Strainenergy=0.5*(ux)**2 + 0.5*(uy)**2
Potentialenergy=0.5*(0.5*(u+uold))**2 - Es*0.25*(0.5*(u+uold))**4
Kineticenergy=numpy.fft.fft2(Kineticenergy)
Strainenergy=numpy.fft.fft2(Strainenergy)
Potentialenergy=numpy.fft.fft2(Potentialenergy)
EnKin[0]=numpy.real(Kineticenergy[0,0])
EnPot[0]=numpy.real(Potentialenergy[0,0])
EnStr[0]=numpy.real(Strainenergy[0,0])
En[0]=EnStr[0]+EnPot[0]+EnKin[0]
En0=En[0]
t=0.0
tdata[0]=t
plotnum=0
#solve pde and plot results
for nt in xrange(numplots-1):
    for n in xrange(plotgap):
        nonlin=u**3
        nonlinhat=numpy.fft.fft2(nonlin)
        vnew=( (0.25*(kxm**2 + kym**2 - 1)*(2*v+vold)
          +(2*v-vold)/(dt*dt) +Es*nonlinhat)/
          (1/(dt*dt) - (kxm**2 + kym**2 -1)*0.25 ) )
        unew=numpy.real(numpy.fft.ifft2(vnew))
        t+=dt
        # update old terms
        vold=v
        v=vnew
        uold=u
        u=unew
    plotnum+=1
    src.mlab_source.scalars = unew
    vx=0.5*kxm*(v+vold)
    vy=0.5*kym*(v+vold)
    ux=numpy.fft.ifft2(vx)
    uy=numpy.fft.ifft2(vy)
    Kineticenergy=0.5*((u-uold)/dt)**2
    Strainenergy=0.5*(ux)**2 + 0.5*(uy)**2
    Potentialenergy=0.5*(0.5*(u+uold))**2 - Es*0.25*(0.5*(u+uold))**4
    Kineticenergy=numpy.fft.fft2(Kineticenergy)
    Strainenergy=numpy.fft.fft2(Strainenergy)
    Potentialenergy=numpy.fft.fft2(Potentialenergy)
    EnKin[plotnum]=numpy.real(Kineticenergy[0,0])
    EnPot[plotnum]=numpy.real(Potentialenergy[0,0])
    EnStr[plotnum]=numpy.real(Strainenergy[0,0])
    En[plotnum]=EnStr[plotnum]+EnPot[plotnum]+EnKin[plotnum]
    Enchange[plotnum-1]=numpy.log(abs(1-En[plotnum]/En0))
    tdata[plotnum]=t


plt.figure()
plt.plot(tdata,En,'r+',tdata,EnKin,'b:',tdata,EnPot,'g-.',tdata,EnStr,'y--
    ')
plt.xlabel('Time')
plt.ylabel('Energy')
plt.legend(('Total', 'Kinetic','Potential','Strain'))
plt.title('Time Dependence of Energy Components')
plt.show()

plt.figure()
plt.plot(Enchange,'r-')
plt.title('Time Dependence of Change in Total Energy')
plt.show()
```

Listing B.16: A Python program to solve the two-dimensional Klein Gordon equation (14.1) using the time discretization in eq. (14.4). Compare this to the Matlab implementation in listing 14.3.

```
#!/usr/bin/env python
"""
A program to solve the 2D Klein Gordon equation using a
```

```
second order semi-explicit method

More information on visualization can be found on the Mayavi
website, in particular:
http://github.enthought.com/mayavi/mayavi/mlab.html
which was last checked on 6 April 2012

"""

import math
import numpy
from mayavi import mlab
import matplotlib.pyplot as plt
import time


# Grid
Lx=3.0      # Period 2*pi*Lx
Ly=3.0      # Period 2*pi*Ly
Nx=512      # Number of harmonics
Ny=512      # Number of harmonics
Nt=200      # Number of time slices
tmax=5.0   # Maximum time
dt=tmax/Nt  # time step
plotgap=10  # time steps between plots
Es= 1.0    # focusing (+1) or defocusing (-1) parameter
numplots=Nt/plotgap  # number of plots to make

x = [i*2.0*math.pi*(Lx/Nx) for i in xrange(-Nx/2,1+Nx/2)]
y = [i*2.0*math.pi*(Ly/Ny) for i in xrange(-Ny/2,1+Ny/2)]
k_x = (1.0/Lx)*numpy.array([complex(0,1)*n for n in range(0,Nx/2) \
+ [0] + range(-Nx/2+1,0)])
k_y = (1.0/Ly)*numpy.array([complex(0,1)*n for n in range(0,Ny/2) \
+ [0] + range(-Ny/2+1,0)])

kxm=numpy.zeros((Nx,Ny), dtype=complex)
kym=numpy.zeros((Nx,Ny), dtype=complex)
xx=numpy.zeros((Nx,Ny), dtype=float)
yy=numpy.zeros((Nx,Ny), dtype=float)


for i in xrange(Nx):
    for j in xrange(Ny):
        kxm[i,j] = k_x[i]
        kym[i,j] = k_y[j]
        xx[i,j] = x[i]
        yy[i,j] = y[j]


# allocate arrays
unew=numpy.zeros((Nx,Ny), dtype=float)
```

```
u=numpy.zeros((Nx,Ny), dtype=float)
uold=numpy.zeros((Nx,Ny), dtype=float)
vnew=numpy.zeros((Nx,Ny), dtype=complex)
v=numpy.zeros((Nx,Ny), dtype=complex)
vold=numpy.zeros((Nx,Ny), dtype=complex)
ux=numpy.zeros((Nx,Ny), dtype=float)
uy=numpy.zeros((Nx,Ny), dtype=float)
vx=numpy.zeros((Nx,Ny), dtype=complex)
vy=numpy.zeros((Nx,Ny), dtype=complex)
Kineticenergy=numpy.zeros((Nx,Ny), dtype=complex)
Potentialenergy=numpy.zeros((Nx,Ny), dtype=complex)
Strainenergy=numpy.zeros((Nx,Ny), dtype=complex)
EnKin=numpy.zeros((numplots), dtype=float)
EnPot=numpy.zeros((numplots), dtype=float)
EnStr=numpy.zeros((numplots), dtype=float)
En=numpy.zeros((numplots), dtype=float)
Enchange=numpy.zeros((numplots-1),dtype=float)
tdata=numpy.zeros((numplots), dtype=float)
nonlin=numpy.zeros((Nx,Ny), dtype=float)
nonlinhat=numpy.zeros((Nx,Ny), dtype=complex)

u=0.1*numpy.exp(-(xx**2 + yy**2))*numpy.sin(10*xx+12*yy)
uold=u
v=numpy.fft.fft2(u)
vold=numpy.fft.fft2(uold)
src = mlab.surf(xx,yy,u,colormap='YlGnBu',warp_scale='auto')
mlab.scalarbar(object=src)
mlab.xlabel('x',object=src)
mlab.ylabel('y',object=src)
mlab.zlabel('u',object=src)
# initial energy
vx=0.5*kxm*(v+vold)
vy=0.5*kym*(v+vold)
ux=numpy.fft.ifft2(vx)
uy=numpy.fft.ifft2(vy)
Kineticenergy=0.5*((u-uold)/dt)**2
Strainenergy=0.5*(ux)**2 + 0.5*(uy)**2
Potentialenergy=0.5*(0.5*(u+uold))**2 - Es*0.25*(0.5*(u+uold))**4
Kineticenergy=numpy.fft.fft2(Kineticenergy)
Strainenergy=numpy.fft.fft2(Strainenergy)
Potentialenergy=numpy.fft.fft2(Potentialenergy)
EnKin[0]=numpy.real(Kineticenergy[0,0])
EnPot[0]=numpy.real(Potentialenergy[0,0])
EnStr[0]=numpy.real(Strainenergy[0,0])
En[0]=EnStr[0]+EnPot[0]+EnKin[0]
En0=En[0]
t=0.0
tdata[0]=t
plotnum=0
#solve pde and plot results
for nt in xrange(numplots-1):
    for n in xrange(plotgap):
        nonlin=u**3
        nonlinhat=numpy.fft.fft2(nonlin)
        vnew=( (0.25*(kxm**2 + kym**2 - 1)*(2*v+vold)
          +(2*v-vold)/(dt*dt) +Es*nonlinhat)/
          (1/(dt*dt) - (kxm**2 + kym**2 -1)*0.25 ) )
        unew=numpy.real(numpy.fft.ifft2(vnew))
        t+=dt
        # update old terms
        vold=v
        v=vnew
        uold=u
        u=unew
    plotnum+=1
    src.mlab_source.scalars = unew
    vx=0.5*kxm*(v+vold)
    vy=0.5*kym*(v+vold)
    ux=numpy.fft.ifft2(vx)
    uy=numpy.fft.ifft2(vy)
    Kineticenergy=0.5*((u-uold)/dt)**2
    Strainenergy=0.5*(ux)**2 + 0.5*(uy)**2
    Potentialenergy=0.5*(0.5*(u+uold))**2 - Es*0.25*(0.5*(u+uold))**4
    Kineticenergy=numpy.fft.fft2(Kineticenergy)
    Strainenergy=numpy.fft.fft2(Strainenergy)
    Potentialenergy=numpy.fft.fft2(Potentialenergy)
    EnKin[plotnum]=numpy.real(Kineticenergy[0,0])
    EnPot[plotnum]=numpy.real(Potentialenergy[0,0])
    EnStr[plotnum]=numpy.real(Strainenergy[0,0])
    En[plotnum]=EnStr[plotnum]+EnPot[plotnum]+EnKin[plotnum]
    Enchange[plotnum-1]=numpy.log(abs(1-En[plotnum]/En0))
    tdata[plotnum]=t


plt.figure()
plt.plot(tdata,En,'r+',tdata,EnKin,'b:',tdata,EnPot,'g-.',tdata,EnStr,'y--
    ')
plt.xlabel('Time')
plt.ylabel('Energy')
plt.legend(('Total', 'Kinetic','Potential','Strain'))
plt.title('Time Dependence of Energy Components')
plt.show()

plt.figure()
plt.plot(Enchange,'r-')
plt.title('Time Dependence of Change in Total Energy')
plt.show()
```

Listing B.17: A Python program to solve the three-dimensional Klein Gordon equation (14.1) using the time discretization in eq. (14.3). Compare this to the Matlab implementation in

listing 14.4.

```
#!/usr/bin/env python
"""
A program to solve the 3D Klein Gordon equation using a
second order semi-explicit method

More information on visualization can be found on the Mayavi
website, in particular:
http://github.enthought.com/mayavi/mayavi/mlab.html
which was last checked on 6 April 2012

"""

import math
import numpy
from mayavi import mlab
import matplotlib.pyplot as plt
import time


# Grid
Lx=2.0      # Period 2*pi*Lx
Ly=2.0      # Period 2*pi*Ly
Lz=2.0      # Period 2*pi*Lz
Nx=64       # Number of harmonics
Ny=64       # Number of harmonics
Nz=64       # Number of harmonics
Nt=2000       # Number of time slices
tmax=10.0   # Maximum time
dt=tmax/Nt  # time step
plotgap=10  # time steps between plots
Es= -1.0    # focusing (+1) or defocusing (-1) parameter
numplots=Nt/plotgap  # number of plots to make

x = [i*2.0*math.pi*(Lx/Nx) for i in xrange(-Nx/2,1+Nx/2)]
y = [i*2.0*math.pi*(Ly/Ny) for i in xrange(-Ny/2,1+Ny/2)]
z = [i*2.0*math.pi*(Lz/Nz) for i in xrange(-Nz/2,1+Nz/2)]
k_x = (1.0/Lx)*numpy.array([complex(0,1)*n for n in range(0,Nx/2) \
+ [0] + range(-Nx/2+1,0)])
k_y = (1.0/Ly)*numpy.array([complex(0,1)*n for n in range(0,Ny/2) \
+ [0] + range(-Ny/2+1,0)])
k_z = (1.0/Lz)*numpy.array([complex(0,1)*n for n in range(0,Nz/2) \
+ [0] + range(-Nz/2+1,0)])

kxm=numpy.zeros((Nx,Ny,Nz), dtype=complex)
kym=numpy.zeros((Nx,Ny,Nz), dtype=complex)
kzm=numpy.zeros((Nx,Ny,Nz), dtype=complex)
xx=numpy.zeros((Nx,Ny,Nz), dtype=float)
yy=numpy.zeros((Nx,Ny,Nz), dtype=float)
zz=numpy.zeros((Nx,Ny,Nz), dtype=float)

```

```

for i in xrange(Nx):
    for j in xrange(Ny):
        for k in xrange(Nz):
            kxm[i,j,k] = k_x[i]
            kym[i,j,k] = k_y[j]
            kzm[i,j,k] = k_z[k]
            xx[i,j,k]=x[i]
            yy[i,j,k]=y[j]
            zz[i,j,k]=z[k]


# allocate arrays
unew=numpy.zeros((Nx,Ny,Nz), dtype=float)
u=numpy.zeros((Nx,Ny,Nz), dtype=float)
uold=numpy.zeros((Nx,Ny,Nz), dtype=float)
vnew=numpy.zeros((Nx,Ny,Nz), dtype=complex)
v=numpy.zeros((Nx,Ny,Nz), dtype=complex)
vold=numpy.zeros((Nx,Ny,Nz), dtype=complex)
ux=numpy.zeros((Nx,Ny,Nz), dtype=float)
uy=numpy.zeros((Nx,Ny,Nz), dtype=float)
uz=numpy.zeros((Nx,Ny,Nz), dtype=float)
vx=numpy.zeros((Nx,Ny,Nz), dtype=complex)
vy=numpy.zeros((Nx,Ny,Nz), dtype=complex)
vz=numpy.zeros((Nx,Ny,Nz), dtype=complex)
Kineticenergy=numpy.zeros((Nx,Ny,Nz), dtype=complex)
Potentialenergy=numpy.zeros((Nx,Ny,Nz), dtype=complex)
Strainenergy=numpy.zeros((Nx,Ny,Nz), dtype=complex)
EnKin=numpy.zeros((numplots), dtype=float)
EnPot=numpy.zeros((numplots), dtype=float)
EnStr=numpy.zeros((numplots), dtype=float)
En=numpy.zeros((numplots), dtype=float)
Enchange=numpy.zeros((numplots-1),dtype=float)
tdata=numpy.zeros((numplots), dtype=float)
nonlin=numpy.zeros((Nx,Ny,Nz), dtype=float)
nonlinhat=numpy.zeros((Nx,Ny,Nz), dtype=complex)

u=0.1*numpy.exp(-(xx**2 + yy**2 + zz**2))
uold=u
v=numpy.fft.fftn(u)
vold=numpy.fft.fftn(uold)
#src=mlab.contour3d(xx,yy,zz,u,colormap='jet',opacity=0.1,contours=4)
src = mlab.pipeline.scalar_field(xx,yy,zz,u,colormap='YlGnBu')
mlab.pipeline.iso_surface(src, contours=[u.min()+0.1*u.ptp(), ],
   colormap='YlGnBu',opacity=0.85)
mlab.pipeline.iso_surface(src, contours=[u.max()-0.1*u.ptp(), ],
   colormap='YlGnBu',opacity=1.0)
mlab.pipeline.image_plane_widget(src,plane_orientation='z_axes',
                            slice_index=Nz/2,colormap='YlGnBu',
                            opacity=0.01)
mlab.pipeline.image_plane_widget(src,plane_orientation='y_axes',
                            slice_index=Ny/2,colormap='YlGnBu',
                            opacity=0.01)
mlab.pipeline.image_plane_widget(src,plane_orientation='x_axes',
                            slice_index=Nx/2,colormap='YlGnBu',
                            opacity=0.01)
mlab.scalarbar()
mlab.xlabel('x',object=src)
mlab.ylabel('y',object=src)
mlab.zlabel('z',object=src)

# initial energy
vx=0.5*kxm*(v+vold)
vy=0.5*kym*(v+vold)
vz=0.5*kzm*(v+vold)
ux=numpy.fft.ifftn(vx)
uy=numpy.fft.ifftn(vy)
uz=numpy.fft.ifftn(vz)
Kineticenergy=0.5*((u-uold)/dt)**2
Strainenergy=0.5*(ux)**2 + 0.5*(uy)**2 + 0.5*(uz)**2
Potentialenergy=0.5*(0.5*(u+uold))**2 - Es*0.25*(0.5*(u+uold))**4
Kineticenergy=numpy.fft.fftn(Kineticenergy)
Strainenergy=numpy.fft.fftn(Strainenergy)
Potentialenergy=numpy.fft.fftn(Potentialenergy)
EnKin[0]=numpy.real(Kineticenergy[0,0,0])
EnPot[0]=numpy.real(Potentialenergy[0,0,0])
EnStr[0]=numpy.real(Strainenergy[0,0,0])
En[0]=EnStr[0]+EnPot[0]+EnKin[0]
En0=En[0]
t=0.0
tdata[1]=t
plotnum=0
#solve pde and plot results
for nt in xrange(numplots-1):
    for n in xrange(plotgap):
        nonlin=u**3
        nonlinhat=numpy.fft.fftn(nonlin)
        vnew=( (0.25*(kxm**2 + kym**2 + kzm**2 - 1)*(2*v+vold)
          +(2*v-vold)/(dt*dt) +Es*nonlinhat)/
          (1/(dt*dt) - (kxm**2 + kym**2 + kzm**2 -1)*0.25 ) )
        unew=numpy.real(numpy.fft.ifftn(vnew))
        t+=dt
        # update old terms
        vold=v
        v=vnew
        uold=u
        u=unew
    plotnum+=1
    src.mlab_source.scalars = unew
    vx=0.5*kxm*(v+vold)
    vy=0.5*kym*(v+vold)
    vz=0.5*kzm*(v+vold)
    ux=numpy.fft.ifftn(vx)
    uy=numpy.fft.ifftn(vy)
    uz=numpy.fft.ifftn(vz)
    Kineticenergy=0.5*((u-uold)/dt)**2
    Strainenergy=0.5*(ux)**2 + 0.5*(uy)**2 + 0.5*(uz)**2
    Potentialenergy=0.5*(0.5*(u+uold))**2 - Es*0.25*(0.5*(u+uold))**4
    Kineticenergy=numpy.fft.fftn(Kineticenergy)
    Strainenergy=numpy.fft.fftn(Strainenergy)
    Potentialenergy=numpy.fft.fftn(Potentialenergy)
    EnKin[plotnum]=numpy.real(Kineticenergy[0,0,0])
    EnPot[plotnum]=numpy.real(Potentialenergy[0,0,0])
    EnStr[plotnum]=numpy.real(Strainenergy[0,0,0])
    En[plotnum]=EnStr[plotnum]+EnPot[plotnum]+EnKin[plotnum]
    Enchange[plotnum-1]=numpy.log(abs(1-En[plotnum]/En0))
    tdata[plotnum]=t


plt.figure()
plt.plot(tdata,En,'r+',tdata,EnKin,'b:',tdata,EnPot,'g-.',tdata,EnStr,'y--
    ')
plt.xlabel('Time')
plt.ylabel('Energy')
plt.legend(('Total', 'Kinetic','Potential','Strain'))
plt.title('Time Dependence of Energy Components')
plt.show()

plt.figure()
plt.plot(Enchange,'r-')
plt.title('Time Dependence of Change in Total Energy')
plt.show()
```