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 3, 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, Divakar Viswanath and Jared Whitehead 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:

- 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.

- The Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

- 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 any 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

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

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.

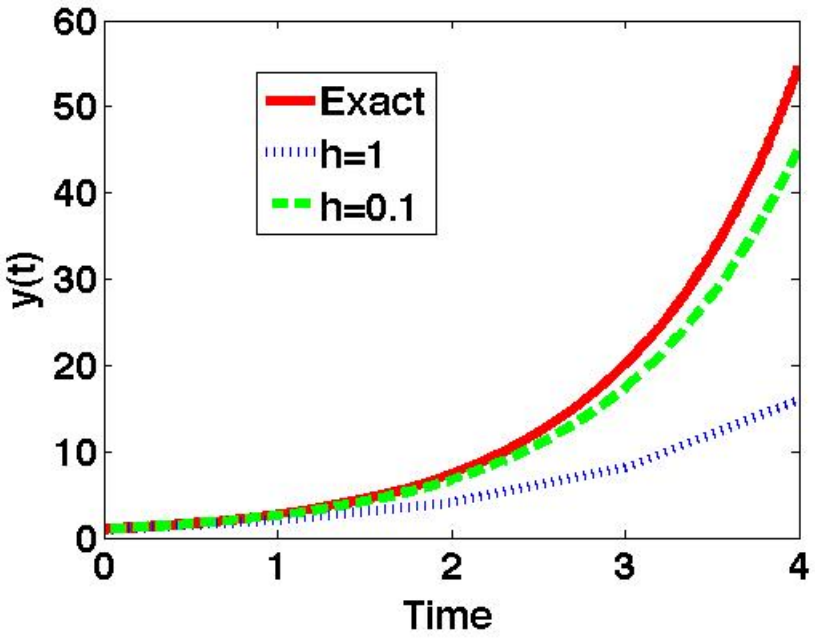

Figure 5.1: A numerical solution to the ODE in eq. (5.1) with $f(t, y) = y$ demonstrating the accuracy of the Forward Euler method for different choices of timestep.

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

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

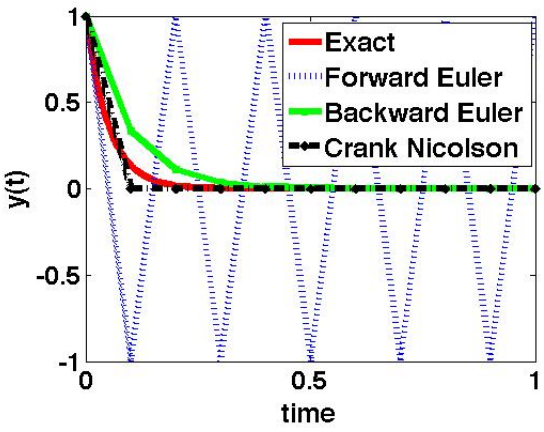

Figure 5.2: A numerical solution to the ODE in eq. (5.10) with $\lambda = 20$ and with a timestep of $h = 0.1$ demonstrating the instability of the Forward Euler method and the stability of the Backward Euler and Crank Nicolson methods.

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.

```
% 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 $0 \leq k < n$, 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 $k \neq l$, 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

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

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

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

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`.

```
3
4  N = 512;                                % Number of grid points.
5  h = 2*pi/N;                             % Size of each grid.
6  x = h*(1:N);                            % Variable x as an array.
7  t = .05*pi;                             % Time to plot solution at
8  dt = .001;                              % Appropriate time step.
9  u0 = zeros(1,N);                        % Array to hold initial data
10 u0(N/2+1:N)= ones(1,N/2);               % Defining the initial data
11 k=(1i*[0:N/2-1 0 -N/2+1:-1]);           % Fourier wavenumbers
12 k3=k.^3;
13 u=ifft(exp(k3*t).*fft(u0));             % Calculate the solution
14 plot(x,u,'r-');                         % Plot the solution
15 xlabel x; ylabel u;                     % Label the axes of the graphs
16 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
```

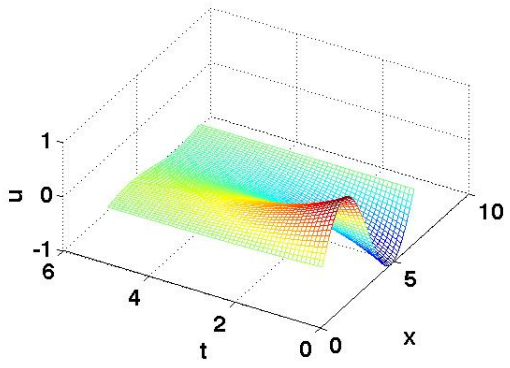

Figure 8.1: A numerical solution to the heat equation, eq. (8.1) computed using the backward Euler method.

```
32      end
33      v = real(ifft(v_hat)); %Back to real space
34      data(i+1,:) = v;
35      t=t+plotgap*dt;
36      tdata = [tdata; t]; %Time vector
37  end
38
39  %Plot using mesh
40  mesh(x,tdata,data), grid on,
41  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,

$$\frac{\partial \hat{u}_k}{\partial t} = \alpha(ik)^2 \hat{u}_k \tag{8.6}$$

$$\frac{\hat{u}_k^{n+1} - \hat{u}_k^n}{h} = \alpha(ik)^2 \hat{u}_k^{n+1} \tag{8.7}$$

$$\hat{u}_k^{n+1}(1 - \alpha h(ik)^2) = \hat{u}_k^n \tag{8.8}$$

$$\hat{u}_k^{n+1} = \frac{\hat{u}_k^n}{(1 - \alpha h(ik)^2)}. \tag{8.9}$$

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 $\text{FFT}(u^3) \neq \text{FFT}(u)^3$, 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.

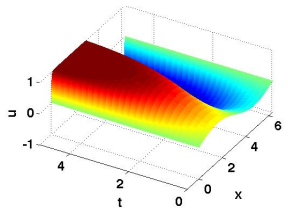

Figure 8.2: A numerical solution to the 1D Allen-Cahn equation, eq. (8.10), with $\epsilon = 0.001$ and $u(x, t = 0) = 0.25 \sin(x)$ computed using an implicit explicit method.

```
26
27 for i = 1:nplots
28     for n = 1:plotgap
29         v_hat = fft(v); %converts to Fourier space
30         vv = v.^3;      %computes nonlinear term in real space
31         vv = fft(vv);   %converts nonlinear term to Fourier space
32         v_hat = (v_hat*(1/dt+1) - vv)./(1/dt-k2*epsilon); %Implicit/
               Explicit
33         v = ifft(v_hat); %Solution back to real space
34     end
35     data(i+1,:) = real(v); %Records data each "plotgap"
36     t=t+plotgap*dt; %Real time
37     tdata = [tdata; t];
38 end
39
40 mesh(x,tdata,data), grid on, axis([-1 2*pi 0 tmax -1 1]),
41 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

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

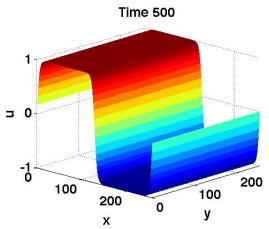

Figure 8.3: A numerical solution to the 2D Allen-Cahn equation, eq. (8.16) at time $t = 500$ with $\epsilon = 0.1$ and $u(x, y, t = 0) = \sin(2\pi x) + 0.001\cos(16\pi x)$ computed using an implicit explicit method.

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.

```
1 %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. □

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

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

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

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

   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
```

```
21  !  FFTW_PATIENT   = value for FFTW input
22  !  FFTW_ESTIMATE  = value for FFTW input
23  !  FFTW_FORWARD     = value for FFTW input
24  !  FFTW_BACKWARD  = value for FFTW input
25  !  pi = 3.14159265358979323846264338327950288419716939937510d0
26  !  L         = width of box
27  !  alpha       = heat conductivity
28  ! .. Scalars ..
29  !  i         = loop counter in x direction
30  !  n         = loop counter for timesteps direction
31  !  allocatestatus = error indicator during allocation
32  !  start       = variable to record start time of program
33  !  finish      = variable to record end time of program
34  !  count_rate    = variable for clock count rate
35  !  planfx      = Forward 1d fft plan in x
36  !  planbx      = Backward 1d fft plan in x
37  !  dt        = timestep
38  ! .. Arrays ..
39  !  u         = approximate REAL solution
40  !  v         = Fourier transform of approximate solution
41  !  vna         = temporary field
42  ! .. Vectors ..
43  !  kx        = fourier frequencies in x direction
44  !  x         = x locations
45  !  time        = times at which save data
46  !  name_config     = array to store filename for data to be saved
47  !
48  ! REFERENCES
49  !
50  ! ACKNOWLEDGEMENTS
51  !
52  ! ACCURACY
53  !
54  ! ERROR INDICATORS AND WARNINGS
55  !
56  ! FURTHER COMMENTS
57  ! Check that the initial iterate is consistent with the
58  ! boundary conditions for the domain specified
59  !
      ------------------------------------------------------------------

60  ! External routines required
61  !
62  ! External libraries required
63  ! FFTW3  -- Fast Fourier Transform in the West Library
64  !     (http://www.fftw.org/)
65
66  PROGRAM main
67
68  ! Declare variables
69  IMPLICIT NONE
```

```
70  INTEGER(kind=4),  PARAMETER    ::  Nx=64
71  INTEGER(kind=4),  PARAMETER ::  Nt=20
72  REAL(kind=8), PARAMETER &
73    ::  pi=3.14159265358979323846264338327950288419716939937510d0
74  REAL(kind=8), PARAMETER ::  L=5.0d0
75  REAL(kind=8), PARAMETER ::  alpha=0.50d0
76  REAL(kind=8)  ::  dt=0.2d0/REAL(Nt,kind(0d0))
77  COMPLEX(KIND=8), DIMENSION(:),ALLOCATABLE ::  kx
78  REAL(kind=8), DIMENSION(:),ALLOCATABLE  ::  x
79  COMPLEX(KIND=8), DIMENSION(:,:),ALLOCATABLE ::  u,v
80  REAL(kind=8), DIMENSION(:),ALLOCATABLE  ::  time
81  COMPLEX(KIND=8), DIMENSION(:),ALLOCATABLE ::  vna
82  INTEGER(kind=4) ::  i,j,k,n
83  INTEGER(kind=4) :: start, finish, count_rate, AllocateStatus
84  INTEGER(kind=4), PARAMETER  :: FFTW_IN_PLACE = 8, FFTW_MEASURE = 0,
        &
85    FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32, FFTW_ESTIMATE = 64
86  INTEGER(kind=4), PARAMETER :: FFTW_FORWARD = -1, FFTW_BACKWARD=1
87  COMPLEX(KIND=8), DIMENSION(:),ALLOCATABLE :: fftfx,fftbx
88  INTEGER(kind=8) :: planfx,planbx
89  CHARACTER*100 :: name_config
90
91  CALL system_clock(start,count_rate)
92  ALLOCATE(kx(1:Nx),x(1:Nx),u(1:Nx,1:1+Nt),v(1:Nx,1:1+Nt),&
93        time(1:1+Nt),vna(1:Nx),fftfx(1:Nx),fftbx(1:Nx),&
94        stat=AllocateStatus)
95    IF (AllocateStatus .ne. 0) STOP
96
97    ! set up ffts
98  CALL dfftw_plan_dft_1d(planfx,Nx,fftfx(1:Nx),fftbx(1:Nx),&
99      FFTW_FORWARD,FFTW_ESTIMATE)
100 CALL dfftw_plan_dft_1d(planbx,Nx,fftbx(1:Nx),fftfx(1:Nx),&
101     FFTW_BACKWARD,FFTW_ESTIMATE)
102
103 PRINT *,'Setup FFTs'
104
105 ! setup fourier frequencies
106 DO i=1,1+Nx/2
107   kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/L
108 END DO
109 kx(1+Nx/2)=0.00d0
110 DO i = 1,Nx/2 -1
111   kx(i+1+Nx/2)=-kx(1-i+Nx/2)
112 END DO
113 DO i=1,Nx
114   x(i)=(-1.00d0 + 2.00d0*REAL(i-1,kind(0d0))/REAL(Nx,KIND(0d0)))*pi
        *L
115 END DO
116
117 PRINT *,'Setup grid and fourier frequencies and splitting
      coefficients'
```

```
118
119   u(1:Nx,1)=sin(x(1:Nx))
120     ! transform initial data
121   CALL dfftw_execute_dft_(planfx,u(1:Nx,1),v(1:Nx,1))
122   PRINT *,'Got initial data, starting timestepping'
123   time(1)=0.0d0
124
125   vna(1:Nx)=v(1:Nx,1)
126   PRINT *,'Starting timestepping'
127   DO n=1,Nt
128     DO i=1,Nx
129       vna(i)=vna(i)/(1-dt*kx(i)*kx(i))
130     END DO
131     PRINT *,'storing plot data ',n
132     time(n+1)=time(n)+dt
133     v(1:Nx,n+1)=vna(1:Nx)
134     CALL dfftw_execute_dft_(planbx,v(1:Nx,n+1),u(1:Nx,n+1))
135     u(1:Nx,n+1)=u(1:Nx,n+1)/REAL(Nx,KIND(0d0))  ! normalize
136   END DO
137   PRINT *,'Finished time stepping'
138   CALL system_clock(finish,count_rate)
139   PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),'for
        execution'
140
141   ! Write data out to disk
142
143   name_config = 'u.dat'
144   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
145   REWIND(11)
146   DO j=1,1+Nt
147     DO i=1,Nx
148       WRITE(11,*) REAL(u(i,j))
149     END DO
150   END DO
151   CLOSE(11)
152
153   name_config = 'tdata.dat'
154   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
155   REWIND(11)
156   DO j=1,1+Nt
157     WRITE(11,*) time(j)
158   END DO
159   CLOSE(11)
160
161   name_config = 'xcoord.dat'
162   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
163   REWIND(11)
164   DO i=1,Nx
165     WRITE(11,*) x(i)
166   END DO
167   CLOSE(11)
```

```
168
169   PRINT *,'Saved data'
170   DEALLOCATE(kx,x,u,v,&
171         time,vna,fftfx,fftbx,&
172         stat=AllocateStatus)
173     IF (AllocateStatus .ne. 0) STOP
174
175   CALL dfftw_destroy_plan(planbx)
176   CALL dfftw_destroy_plan(planfx)
177   CALL dfftw_cleanup()
178   PRINT *,'Program execution complete'
179   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.

```
1  #!/bin/bash
2  #PBS -N heatequation
3  #PBS -l nodes=1,walltime=00:10:00
4  #PBS -l qos=math471f11_flux
5  #PBS -A math471f11_flux
6  #PBS -q flux
7  #PBS -M your_uniqname@umich.edu
8  #PBS -m abe
9  #PBS -V
10 # Create a local directory to run and copy your files to local.
11 # Let PBS handle your output
12 mkdir /tmp/${PBS_JOBID}
13 cp ${HOME}/ParallelMethods/Heat/heatequation /tmp/${PBS_JOBID}/
      heatequation
14 cd /tmp/${PBS_JOBID}
15 ./heatequation
16 #Clean up your files
17 cd
18 cd ParallelMethods/Heat
19 # Retrieve your output
20 cp /tmp/${PBS_JOBID}/u.dat ${HOME}/ParallelMethods/Heat/u.dat
21 cp /tmp/${PBS_JOBID}/xcoord.dat ${HOME}/ParallelMethods/Heat/xcoord.
      dat
22 cp /tmp/${PBS_JOBID}/tdata.dat ${HOME}/ParallelMethods/Heat/tdata.dat
23
24 /bin/rm -rf /tmp/${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.

```
% 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.

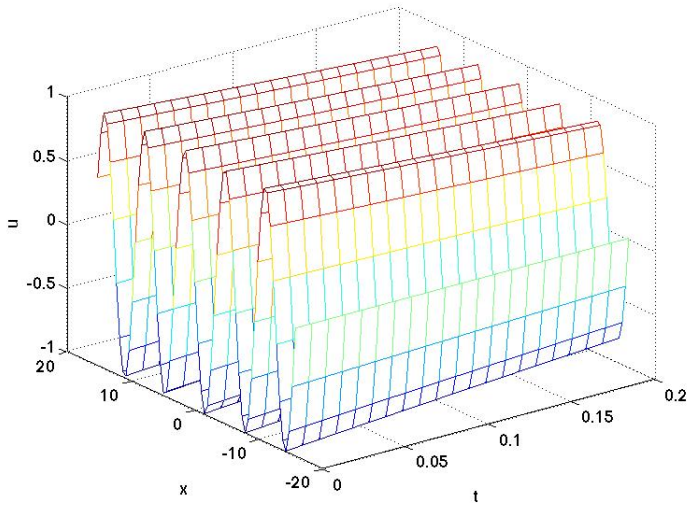

Figure 10.1: The solution to the heat equation computed by Fortran and post-processed by Matlab.

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.

```
1  !
     ------------------------------------------------------------------
2  !
3  !
4  ! PURPOSE
5  !
6  ! This program uses OpenMP to print hello world from all available
7  ! threads
8  !
9  ! .. Parameters ..
10 !
11 ! .. Scalars ..
12 !  id       = thread id
13 !  nthreads     = total number of threads
14 !
15 ! .. Arrays ..
16 !
17 ! .. Vectors ..
18 !
19 ! REFERENCES
20 ! http:// en.wikipedia.org/wiki/OpenMP
21 !
22 ! ACKNOWLEDGEMENTS
23 ! The program below was modified from one available at the internet
24 ! address in the references. This internet address was last checked
25 ! on 30 December 2011
26 !
27 ! ACCURACY
28 !
29 ! ERROR INDICATORS AND WARNINGS
30 !
31 ! FURTHER COMMENTS
32 !
33 !
     ------------------------------------------------------------------
```

```
34   ! External routines required
35   !
36   ! External libraries required
37   ! OpenMP library
38   PROGRAM hello90
39   USE omp_lib
40   IMPLICIT NONE
41   INTEGER:: id, nthreads
42   !$OMP PARALLEL PRIVATE(id)
43   id = omp_get_thread_num()
44   nthreads = omp_get_num_threads()
45   PRINT *, 'Hello World from thread', id
46   !$OMP BARRIER
47   IF ( id == 0 ) THEN
48     PRINT*, 'There are', nthreads, 'threads'
49   END IF
50   !$OMP END PARALLEL
51   END PROGRAM
```

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

```
1  #define the complier
2  COMPILER = ifort
3  # compilation settings, optimization, precision, parallelization
4    FLAGS = -O0 -openmp
5
6  # libraries
7  LIBS =
8  # source list for main program
9  SOURCES =  helloworld.f90
10
11 test: $(SOURCES)
12     ${COMPILER} -o helloworld $(FLAGS) $(SOURCES)
13
14 clean:
15   rm *.o
16
17 clobber:
18   rm helloworld
```

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

```
1  #!/bin/bash
2  #PBS -N helloworld
3  #PBS -l nodes=1:ppn=2,walltime=00:02:00
4  #PBS -q flux
5  #PBS -l qos=math471f11_flux
6  #PBS -A math471f11_flux
7  #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)
```

```
47
48    PRINT*, 'Hello World from process', myid
49    CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
50    IF ( myid == 0 ) THEN
51      PRINT*, 'There are ', numprocs, ' MPI processes'
52    END IF
53    CALL MPI_FINALIZE(ierr)
54    END PROGRAM
```

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

```
1  #define the complier
2  COMPILER = mpif90
3  # compilation settings, optimization, precision, parallelization
4    FLAGS = -O0
5
6  # libraries
7  LIBS =
8  # source list for main program
9  SOURCES =  helloworld.f90
10
11 test: $(SOURCES)
12     ${COMPILER} -o helloworld $(FLAGS) $(SOURCES)
13
14 clean:
15   rm *.o
16
17 clobber:
18   rm helloworld
```

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

```
1  #!/bin/bash
2  #PBS -N helloworld
3  #PBS -l nodes=1:ppn=2,walltime=00:02:00
4  #PBS -q flux
5  #PBS -l qos=math471f11_flux
6  #PBS -A math471f11_flux
7  #PBS -M your_uniqname@umich.edu
8  #PBS -m abe
9  #PBS -V
10 #
11 # Create a local directory to run and copy your files to local.
12 # Let PBS handle your output
13 mkdir /tmp/${PBS_JOBID}
14 cp ${HOME}/ParallelMethods/helloworldMPI/helloworld /tmp/${PBS_JOBID
       }/helloworld
15 cd /tmp/${PBS_JOBID}
16
```

```
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*

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

and the **Y mean** *is*

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

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

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

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

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

[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

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

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

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

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 $R = \{(x, y) : 0 \leq x^2 + y^2 \leq 1\}$ 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 $y^2 + x^3 \leq 1$ 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 $\iint_R e^{xy}\,dA$, 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 $\iiint_S e^{xyz}\,dV$, 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.

```
1
2
3
4
5
6   !-----------------------------------------------------------------------
7   !
8   !
9   ! PURPOSE
10  !
11  ! This program use a monte carlo method to calculate pi
12  !
13  ! .. Parameters ..
14  !  npts       = total number of Monte Carlo points
15  !  xmin       = lower bound for integration region
16  !  xmax             = upper bound for integration region
17  ! .. Scalars ..
18  !  i                = loop counter
19  !  f        = average value from summation
20  !  sum              = total sum
21  !  randnum          = random number generated from (0,1) uniform
22  !                     distribution
23  !  x                = current Monte Carlo location
24  ! .. Arrays ..
25  !
26  ! .. Vectors ..
27  !
28  ! REFERENCES
29  ! http://chpc.wustl.edu/mpi-fortran.html
30  ! 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.

```
31  !
32  ! ACKNOWLEDGEMENTS
33  ! The program below was modified from one available at the internet
34  ! address in the references. This internet address was last checked
35  ! on 30 March 2012
36  !
37  ! ACCURACY
38  !
39  ! ERROR INDICATORS AND WARNINGS
40  !
41  ! FURTHER COMMENTS
42  !
43  !-----------------------------------------------------------------------
44  ! External routines required
45  !
46  ! External libraries required
47  ! None
48  PROGRAM monte_carlo
49    IMPLICIT NONE
50
51    INTEGER(kind=8), PARAMETER      :: npts = 1e10
52    REAL(kind=8), PARAMETER     :: xmin=0.0d0,xmax=1.0d0
53    INTEGER(kind=8)          :: i
54    REAL(kind=8)           :: f,sum, randnum,x
55
56    DO i=1,npts
57      CALL random_number(randnum)
58      x = (xmax-xmin)*randnum + xmin
59      sum = sum + 4.0d0/(1.0d0 + x**2)
60    END DO
61    f = sum/npts
62    PRINT*,'PI calculated with ',npts,' points = ',f
63
64    STOP
65    END
```

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

```
1 #define the complier
2 COMPILER = mpif90
3 # compilation settings, optimization, precision, parallelization
4   FLAGS = -O0
5
6 # libraries
7 LIBS =
8 # source list for main program
9 SOURCES =  montecarloserial.f90
10
11 test: $(SOURCES)
12     ${COMPILER} -o montecarloserial $(FLAGS) $(SOURCES)
```

```
13
14 clean:
15   rm *.o
16
17 clobber:
18   rm  montecarloserial
```

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

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

Parallel

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

```
1
2
3
4
5
6   !------------------------------------------------------------------
7   !
8   !
9   ! PURPOSE
10  !
11  ! 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
```

```
63      INTEGER(kind=8)            :: i
64      REAL(kind=8)             :: f,sum,mysum,randnum
65      REAL(kind=8)             :: x, start, finish
66
67      ! Initialize MPI
68      CALL MPI_INIT(ierr)
69      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
70      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
71      start=MPI_WTIME()
72
73    ! Calculate the number of points each MPI process needs to generate
74      IF (myid .eq. 0) THEN
75        mynpts = npts - (nprocs-1)*(npts/nprocs)
76      ELSE
77        mynpts = npts/nprocs
78      ENDIF
79
80      ! set initial sum to zero
81      mysum = 0.0d0
82    ! use loop on local process to generate portion of Monte Carlo integral
83      DO i=1,mynpts
84        CALL random_number(randnum)
85        x = (xmax-xmin)*randnum + xmin
86        mysum = mysum + 4.0d0/(1.0d0 + x**2)
87      ENDDO
88
89    ! Do a reduction and sum the results from all processes
90      CALL MPI_REDUCE(mysum,sum,1,MPI_DOUBLE_PRECISION,MPI_SUM,&
91            0,MPI_COMM_WORLD,ierr)
92      finish=MPI_WTIME()
93
94      ! Get one process to output the result and running time
95      IF (myid .eq. 0) THEN
96           f = sum/npts
97           PRINT*,'PI calculated with ',npts,' points = ',f
98           PRINT*,'Program took ', finish-start, ' for Time stepping'
99      ENDIF
100
101     CALL MPI_FINALIZE(ierr)
102
103     STOP
104     END PROGRAM
```

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

```
1 #define the complier
2 COMPILER = mpif90
3 # compilation settings, optimization, precision, parallelization
4   FLAGS = -O0
5
```

```
# 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 $\boldsymbol{u} = \exp((\boldsymbol{A}+\boldsymbol{B})t)\,\boldsymbol{u}(t=0)$, 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 $u(t=0)\exp((\boldsymbol{A}+\boldsymbol{B})\delta t)$ 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.

```
% 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
```

```
36  !  planbx      = Backward 1d fft plan in x
37  !  dt        = timestep
38  ! .. Arrays ..
39  !  u         = approximate solution
40  !  v         = Fourier transform of approximate solution
41  ! .. Vectors ..
42  !  una         = temporary field
43  !  unb         = temporary field
44  !  vna         = temporary field
45  !  pot         = potential
46  !  kx        = fourier frequencies in x direction
47  !  x         = x locations
48  !  time        = times at which save data
49  !  name_config     = array to store filename for data to be saved
50  !  fftfx       = array to setup x Fourier transform
51  !  fftbx       = array to setup x Fourier transform
52  ! REFERENCES
53  !
54  ! ACKNOWLEDGEMENTS
55  !
56  ! ACCURACY
57  !
58  ! ERROR INDICATORS AND WARNINGS
59  !
60  ! FURTHER COMMENTS
61  ! Check that the initial iterate is consistent with the
62  ! boundary conditions for the domain specified
63  !-----------------------------------------------------------------------
64  ! External routines required
65  !
66  ! External libraries required
67  ! FFTW3  -- Fast Fourier Transform in the West Library
68  !     (http://www.fftw.org/)
69
70
71  PROGRAM main
72
73  ! Declare variables
74  IMPLICIT NONE
75  INTEGER(kind=4), PARAMETER  :: Nx=8*256
76  INTEGER(kind=4), PARAMETER  :: Nt=200
77  REAL(kind=8), PARAMETER   &
78    ::  pi=3.14159265358979323846264338327950288419716939937510d0
79  REAL(kind=8), PARAMETER :: L=5.0d0
80  REAL(kind=8), PARAMETER :: Es=1.0d0
81  REAL(kind=8)  :: dt=2.0d0/Nt
82  COMPLEX(kind=8),  DIMENSION(:), ALLOCATABLE :: kx
83  REAL(kind=8),  DIMENSION(:), ALLOCATABLE  :: x
84  COMPLEX(kind=8),  DIMENSION(:,:), ALLOCATABLE :: u
85  COMPLEX(kind=8),  DIMENSION(:,:), ALLOCATABLE :: v
86  COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  :: una,vn
```

```
87  COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  :: unb,pot
88  REAL(kind=8), DIMENSION(:), ALLOCATABLE   :: time
89  INTEGER(kind=4) :: i,j,k,n,modes,AllocateStatus
90  INTEGER(kind=4) :: start, finish, count_rate
91  INTEGER(kind=4), PARAMETER  :: FFTW_IN_PLACE = 8, FFTW_MEASURE = 0, &
92    FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32, FFTW_ESTIMATE = 64
93  INTEGER(kind=4), PARAMETER  :: FFTW_FORWARD = -1, FFTW_BACKWARD=1
94  COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  :: fftfx,fftbx
95  INTEGER(kind=8) :: planfx,planbx
96    CHARACTER*100 :: name_config
97
98  CALL system_clock(start,count_rate)
99  ALLOCATE(kx(1:Nx),x(1:Nx),u(1:Nx,1:Nt+1),v(1:Nx,1:Nt+1),&
100   una(1:Nx),vn(1:Nx),unb(1:Nx),pot(1:Nx),time(1:Nt+1),&
101   fftfx(1:Nx),fftbx(1:Nx),stat=AllocateStatus)
102 IF (allocatestatus .ne. 0) STOP
103 ! set up ffts
104 CALL dfftw_plan_dft_1d_(planfx,Nx,fftfx(1:Nx),fftbx(1:Nx),&
105   FFTW_FORWARD,FFTW_PATIENT)
106 CALL dfftw_plan_dft_1d_(planbx,Nx,fftbx(1:Nx),fftfx(1:Nx),&
107   FFTW_BACKWARD,FFTW_PATIENT)
108 PRINT *,'Setup FFTs'
109   ! setup fourier frequencies
110 DO i=1,1+Nx/2
111   kx(i)= cmplx(0.0d0,1.0d0)*(i-1.0d0)/L
112 END DO
113 kx(1+Nx/2)=0.0d0
114 DO i = 1,Nx/2 -1
115   kx(i+1+Nx/2)=-kx(1-i+Nx/2)
116 END DO
117 DO i=1,Nx
118   x(i)=(-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)))*pi*L
119 END DO
120 PRINT *,'Setup grid and fourier frequencies'
121
122 DO i=1,Nx
123   u(i,1)=exp(-1.0d0*(x(i)**2))
124 END DO
125 ! transform initial data
126 CALL dfftw_execute_dft_(planfx,u(1:Nx,1),v(1:Nx,1))
127 PRINT *,'Got initial data, starting timestepping'
128 time(1)=0.0d0
129 DO n=1,Nt
130   time(n+1)=n*dt
131   DO i=1,Nx
132     vn(i)=exp(0.5d0*dt*kx(i)*kx(i)*cmplx(0.0d0,1.0d0))*v(i,n)
133   END DO
134   CALL dfftw_execute_dft_(planbx,vn(1:Nx),una(1:Nx))
135   ! normalize
136   DO i=1,Nx
137     una(i)=una(1:Nx)/REAL(Nx,kind(0d0))
```

```
138       pot(i)=Es*una(i)*conjg(una(i))
139       unb(i)=exp(cmplx(0.0d0,-1.0d0)*dt*pot(i))*una(i)
140     END DO
141     CALL dfftw_execute_dft_(planfx,unb(1:Nx),vn(1:Nx))
142     DO i=1,Nx
143       v(i,n+1)=exp(0.50d0*dt*kx(i)*kx(i)*cmplx(0.0d0,1.0d0))*vn(i)
144     END DO
145     CALL dfftw_execute_dft_(planbx,v(1:Nx,n+1),u(1:Nx,n+1))
146     ! normalize
147     DO i=1,Nx
148       u(i,n+1)=u(i,n+1)/REAL(Nx,kind(0d0))
149     END DO
150   END DO
151   PRINT *,'Finished time stepping'
152   CALL system_clock(finish,count_rate)
153   PRINT*,'Program took ',&
154     REAL(finish-start,kind(0d0))/REAL(count_rate,kind(0d0)),'for execution
        '
155
156   name_config = 'u.dat'
157   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
158   REWIND(11)
159   DO j=1,Nt
160     DO i=1,Nx
161       WRITE(11,*) abs(u(i,j))**2
162     END DO
163   END DO
164   CLOSE(11)
165
166   name_config = 'tdata.dat'
167   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
168   REWIND(11)
169   DO j=1,Nt
170     WRITE(11,*) time(j)
171   END DO
172   CLOSE(11)
173
174   name_config = 'xcoord.dat'
175   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
176   REWIND(11)
177   DO i=1,Nx
178     WRITE(11,*) x(i)
179   END DO
180   CLOSE(11)
181
182   PRINT *,'Saved data'
183
184   CALL dfftw_destroy_plan_(planbx)
185   CALL dfftw_destroy_plan_(planfx)
186   CALL dfftw_cleanup_()
187
```

```
188   DEALLOCATE(kx,x,u,v,una,vn,unb,&
189         pot,time,fftfx,fftbx,&
190           stat=AllocateStatus)
191   IF (allocatestatus .ne. 0) STOP
192   PRINT *,'deallocated memory'
193   PRINT *,'Program execution complete'
194   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.

```
1  % A program to plot the computed results
2
3  clear all; format compact, format short,
4  set(0,'defaultaxesfontsize',18,'defaultaxeslinewidth',.9,...
5      'defaultlinelinewidth',3.5,'defaultpatchlinewidth',5.5);
6
7  % Load data
8  load('./u.dat');
9  load('./tdata.dat');
10 load('./xcoord.dat');
11 Tsteps = length(tdata);
12
13 Nx = length(xcoord); Nt = length(tdata);
14
15 u = reshape(u,Nx,Nt);
16
17 % Plot data
18 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
```

```
51  !  vnbx       = temporary field
52  !  vnay       = temporary field
53  !  vnby       = temporary field
54  !  potx       = potential
55  ! .. Vectors ..
56  !  kx       = fourier frequencies in x direction
57  !  ky       = fourier frequencies in y direction
58  !  x        = x locations
59  !  y        = y locations
60  !  time       = times at which save data
61  !  name_config    = array to store filename for data to be saved
62  !  fftfx      = array to setup x Fourier transform
63  !  fftbx      = array to setup x Fourier transform
64  !  fftfy      = array to setup y Fourier transform
65  !  fftby      = array to setup y Fourier transform
66  !
67  ! REFERENCES
68  !
69  ! ACKNOWLEDGEMENTS
70  !
71  ! ACCURACY
72  !
73  ! ERROR INDICATORS AND WARNINGS
74  !
75  ! FURTHER COMMENTS
76  ! Check that the initial iterate is consistent with the
77  ! boundary conditions for the domain specified
78  !-----------------------------------------------------------------------
79  ! External routines required
80  !
81  ! External libraries required
82  ! FFTW3  -- Fast Fourier Transform in the West Library
83  !     (http://www.fftw.org/)
84  ! OpenMP library
85  PROGRAM main
86  USE omp_lib
87  IMPLICIT NONE
88  ! Declare variables
89  INTEGER(kind=4), PARAMETER  ::  Nx=1024
90  INTEGER(kind=4), PARAMETER  ::  Ny=1024
91  INTEGER(kind=4), PARAMETER  ::  Nt=20
92  INTEGER(kind=4), PARAMETER  ::  plotgap=5
93  REAL(kind=8), PARAMETER   ::  &
94  pi=3.14159265358979323846264338327950288419716939937510d0
95  REAL(kind=8), PARAMETER   ::  Lx=2.0d0
96  REAL(kind=8), PARAMETER   ::  Ly=2.0d0
97  REAL(kind=8), PARAMETER   ::  Es=1.0d0
98  REAL(kind=8)        ::  dt=0.10d0/Nt
99  COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  kx
100 COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  ky
101 REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  x
```

```
102   REAL(kind=8),     DIMENSION(:), ALLOCATABLE  ::  y
103   COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  unax,vnax,vnbx,potx
104   COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  vnay,vnby
105   REAL(kind=8),     DIMENSION(:), ALLOCATABLE  ::  time
106   INTEGER(kind=4)        ::  i,j,k,n,allocatestatus,ierr
107   INTEGER(kind=4)        ::  start, finish, count_rate, numthreads
108   INTEGER(kind=8), PARAMETER  ::  FFTW_IN_PLACE=8, FFTW_MEASURE=0,&
109                   FFTW_EXHAUSTIVE=8, FFTW_PATIENT=32,&
110                            FFTW_ESTIMATE=64
111   INTEGER(kind=8),PARAMETER   ::  FFTW_FORWARD=-1, FFTW_BACKWARD=1
112   INTEGER(kind=8)        ::  planfxy,planbxy
113     CHARACTER*100        ::  name_config,number_file
114
115   numthreads=omp_get_max_threads()
116   PRINT *,'There are ',numthreads,' threads.'
117
118   ALLOCATE(kx(1:Nx),ky(1:Nx),x(1:Nx),y(1:Nx),unax(1:Nx,1:Ny),&
119       vnax(1:Nx,1:Ny),potx(1:Nx,1:Ny),time(1:1+Nt/plotgap),&
120       stat=allocatestatus)
121   IF (allocatestatus .ne. 0) stop
122   PRINT *,'allocated memory'
123
124   ! set up multithreaded ffts
125   CALL dfftw_init_threads_(ierr)
126   PRINT *,'Initiated threaded FFTW'
127   CALL dfftw_plan_with_nthreads_(numthreads)
128   PRINT *,'Inidicated number of threads to be used in planning'
129   CALL dfftw_plan_dft_2d_(planfxy,Nx,Ny,unax(1:Nx,1:Ny),vnax(1:Nx,1:Ny),&
130               FFTW_FORWARD,FFTW_ESTIMATE)
131   CALL dfftw_plan_dft_2d_(planbxy,Nx,Ny,vnax(1:Nx,1:Ny),unax(1:Nx,1:Ny),&
132               FFTW_BACKWARD,FFTW_ESTIMATE)
133   PRINT *,'Setup FFTs'
134
135   ! setup fourier frequencies
136   !$OMP PARALLEL PRIVATE(i,j)
137   !$OMP DO SCHEDULE(static)
138   DO i=1,1+Nx/2
139     kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
140   END DO
141   !$OMP END DO
142   kx(1+Nx/2)=0.0d0
143   !$OMP DO SCHEDULE(static)
144   DO i = 1,Nx/2 -1
145     kx(i+1+Nx/2)=-kx(1-i+Nx/2)
146   END DO
147   !$OMP END DO
148   !$OMP DO SCHEDULE(static)
149     DO i=1,Nx
150     x(i)=(-1.0d0+2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)) )*pi*Lx
151   END DO
152   !$OMP END DO
```

```
153   !$OMP DO SCHEDULE(static)
154   DO j=1,1+Ny/2
155     ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
156   END DO
157   !$OMP END DO
158   ky(1+Ny/2)=0.0d0
159   !$OMP DO SCHEDULE(static)
160   DO j = 1,Ny/2 -1
161     ky(j+1+Ny/2)=-ky(1-j+Ny/2)
162   END DO
163   !$OMP END DO
164   !$OMP DO SCHEDULE(static)
165     DO j=1,Ny
166     y(j)=(-1.0d0+2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)) )*pi*Ly
167   END DO
168   !$OMP END DO
169   PRINT *,'Setup grid and fourier frequencies'
170   !$OMP DO SCHEDULE(static)
171   DO j=1,Ny
172     unax(1:Nx,j)=exp(-1.0d0*(x(1:Nx)**2 +y(j)**2))
173   END DO
174   !$OMP END DO
175   !$OMP END PARALLEL
176   name_config = 'uinitial.dat'
177   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
178   REWIND(11)
179   DO j=1,Ny
180     DO i=1,Nx
181       WRITE(11,*) abs(unax(i,j))**2
182     END DO
183   END DO
184   CLOSE(11)
185   ! transform initial data and do first half time step
186   CALL dfftw_execute_dft_(planfxy,unax(1:Nx,1:Ny),vnax(1:Nx,1:Ny))
187
188   PRINT *,'Got initial data, starting timestepping'
189   time(1)=0.0d0
190   CALL system_clock(start,count_rate)
191   DO n=1,Nt
192     !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
193     DO j=1,Ny
194       DO i=1,Nx
195         vnax(i,j)=exp(0.5d0*dt*(kx(i)*kx(i) + ky(j)*ky(j))&
196           *cmplx(0.0d0,1.0d0))*vnax(i,j)
197       END DO
198     END DO
199     !$OMP END PARALLEL DO
200     CALL dfftw_execute_dft_(planbxy,vnax(1:Nx,1:Ny),unax(1:Nx,1:Ny))
201     !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
202     DO j=1,Ny
203       DO i=1,Nx
```

```
204         unax(i,j)=unax(i,j)/REAL(Nx*Ny,kind(0d0))
205         potx(i,j)=Es*unax(i,j)*conjg(unax(i,j))
206         unax(i,j)=exp(cmplx(0.0d0,-1.0d0)*dt*potx(i,j))&
207             *unax(i,j)
208       END DO
209     END DO
210     !$OMP END PARALLEL DO
211     CALL dfftw_execute_dft_(planfxy,unax(1:Nx,1:Ny),vnax(1:Nx,1:Ny))
212     !$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
213     DO j=1,Ny
214       DO i=1,Nx
215         vnax(i,j)=exp(0.5d0*dt*(kx(i)*kx(i) + ky(j)*ky(j))&
216             *cmplx(0.0d0,1.0d0))*vnax(i,j)
217       END DO
218     END DO
219     !$OMP END PARALLEL DO
220     IF (mod(n,plotgap)==0) then
221       time(1+n/plotgap)=n*dt
222       PRINT *,'time',n*dt
223       CALL dfftw_execute_dft_(planbxy,vnax(1:Nx,1:Ny),unax(1:Nx,1:Ny))
224       !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
225       DO j=1,Ny
226         DO i=1,Nx
227           unax(i,j)=unax(i,j)/REAL(Nx*Ny,kind(0d0))
228         END DO
229       END DO
230       !$OMP END PARALLEL DO
231       name_config='./data/u'
232       WRITE(number_file,'(i0)') 10000000+1+n/plotgap
233       ind=index(name_config,' ') -1
234       name_config=name_config(1:ind)//numberfile
235       ind=index(name_config,' ') -1
236       name_config=name_config(1:ind)//'.dat'
237       OPEN(unit=11,FILE=name_config,status="UNKNOWN")
238       REWIND(11)
239       DO j=1,Ny
240         DO i=1,Nx
241           WRITE(11,*) abs(unax(i,j))**2
242         END DO
243       END DO
244       CLOSE(11)
245     END IF
246   END DO
247   PRINT *,'Finished time stepping'
248   CALL system_clock(finish,count_rate)
249   PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),&
250     'for Time stepping'
251
252
253   name_config = 'tdata.dat'
254   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
```

```
255   REWIND(11)
256   DO j=1,1+Nt/plotgap
257     WRITE(11,*) time(j)
258   END DO
259   CLOSE(11)
260
261   name_config = 'xcoord.dat'
262   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
263   REWIND(11)
264   DO i=1,Nx
265     WRITE(11,*) x(i)
266   END DO
267   CLOSE(11)
268
269   name_config = 'ycoord.dat'
270   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
271   REWIND(11)
272   DO j=1,Ny
273     WRITE(11,*) y(j)
274   END DO
275   CLOSE(11)
276   PRINT *,'Saved data'
277
278   CALL dfftw_destroy_plan_(planbxy)
279   CALL dfftw_destroy_plan_(planfxy)
280   CALL dfftw_cleanup_threads_()
281
282   DEALLOCATE(unax,vnax,potx,stat=allocatestatus)
283   IF (allocatestatus .ne. 0) STOP
284   PRINT *,'Deallocated memory'
285
286     PRINT *,'Program execution complete'
287   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 $-mp$.

```
1 #define the complier
2 COMPILER = gfortran
3 # compilation settings, optimization, precision, parallelization
4 FLAGS = -O3  -fopenmp
5
6
7 # libraries
8 LIBS = -L/usr/local/lib -lfftw3 -lm
9 # 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
```

```
47  !  unax       = temporary field
48  !  vnax       = temporary field
49  !  vnbx       = temporary field
50  !  vnay       = temporary field
51  !  vnby       = temporary field
52  !  potx       = potential
53  ! .. Vectors ..
54  !  kx       = fourier frequencies in x direction
55  !  ky       = fourier frequencies in y direction
56  !  x        = x locations
57  !  y        = y locations
58  !  time       = times at which save data
59  !  name_config    = array to store filename for data to be saved
60  !  fftfx      = array to setup x Fourier transform
61  !  fftbx      = array to setup x Fourier transform
62  !  fftfy      = array to setup y Fourier transform
63  !  fftby      = array to setup y Fourier transform
64  !
65  ! REFERENCES
66  !
67  ! ACKNOWLEDGEMENTS
68  !
69  ! ACCURACY
70  !
71  ! ERROR INDICATORS AND WARNINGS
72  !
73  ! FURTHER COMMENTS
74  ! Check that the initial iterate is consistent with the
75  ! boundary conditions for the domain specified
76  !
    ----------------------------------------------------------------------

77  ! External routines required
78  !
79  ! External libraries required
80  ! FFTW3  -- Fast Fourier Transform in the West Library
81  !     (http://www.fftw.org/)
82  ! OpenMP library
83
84  PROGRAM main
85  USE omp_lib
86  IMPLICIT NONE
87  ! Declare variables
88  INTEGER(kind=4), PARAMETER  ::  Nx=2**8
89  INTEGER(kind=4), PARAMETER  ::  Ny=2**8
90  INTEGER(kind=4), PARAMETER  ::  Nt=20
91  INTEGER(kind=4), PARAMETER  ::  plotgap=5
92  REAL(kind=8), PARAMETER   :: &
93    pi=3.14159265358979323846264338327950288419716939937510d0
94  REAL(kind=8), PARAMETER   ::  Lx=2.0d0
95  REAL(kind=8), PARAMETER   ::  Ly=2.0d0
```

```
96  REAL(kind=8), PARAMETER    ::  Es=0.0d0
97  REAL(kind=8)          ::  dt=0.10d0/Nt
98  COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  kx,ky
99  REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  x,y
100 COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  unax,vnax,vnbx,potx
101 COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  vnay,vnby
102 REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  time
103 INTEGER(kind=4)        ::  i,j,k,n,allocatestatus
104 INTEGER(kind=4)        ::  start, finish, count_rate
105 INTEGER(kind=8), PARAMETER  ::  FFTW_IN_PLACE=8, FFTW_MEASURE=0,&
106                 FFTW_EXHAUSTIVE=8, FFTW_PATIENT=32,&
107                           FFTW_ESTIMATE=64
108 INTEGER(kind=8),PARAMETER   ::  FFTW_FORWARD=-1, FFTW_BACKWARD=1
109 COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE :: fftfx,fftbx,fftfy,
       fftby
110 INTEGER(kind=8)        ::  planfx,planbx,planfy,planby
111   CHARACTER*100        ::  name_config
112
113 ALLOCATE(kx(1:Nx),ky(1:Nx),x(1:Nx),y(1:Nx),unax(1:Nx,1:Ny),&
114     vnax(1:Nx,1:Ny),vnbx(1:Nx,1:Ny),potx(1:Nx,1:Ny),fftfx(1:Nx),&
115     fftbx(1:Nx),fftfy(1:Nx),fftby(1:Nx),vnay(1:Ny,1:Nx),&
116     vnby(1:Ny,1:Nx),time(1:1+Nt/plotgap),stat=allocatestatus)
117 IF (allocatestatus .ne. 0) stop
118 PRINT *,'allocated memory'
119   ! set up ffts
120 CALL dfftw_plan_dft_1d_(planfx,Nx,fftfx(1:Nx),fftbx(1:Nx),&
121     FFTW_FORWARD,FFTW_ESTIMATE)
122 CALL dfftw_plan_dft_1d_(planbx,Nx,fftbx(1:Nx),fftfx(1:Nx),&
123     FFTW_BACKWARD,FFTW_ESTIMATE)
124 CALL dfftw_plan_dft_1d_(planfy,Ny,fftfy(1:Ny),fftby(1:Ny),&
125     FFTW_FORWARD,FFTW_ESTIMATE)
126 CALL dfftw_plan_dft_1d_(planby,Ny,fftby(1:Ny),fftfy(1:Ny),&
127     FFTW_BACKWARD,FFTW_ESTIMATE)
128 PRINT *,'Setup FFTs'
129
130 ! setup fourier frequencies
131 !$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
132 DO i=1,1+Nx/2
133   kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
134 END DO
135 !$OMP END PARALLEL DO
136 kx(1+Nx/2)=0.0d0
137 !$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
138 DO i = 1,Nx/2 -1
139   kx(i+1+Nx/2)=-kx(1-i+Nx/2)
140 END DO
141 !$OMP END PARALLEL DO
142 !$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
143   DO i=1,Nx
144   x(i)=(-1.0d0+2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)) )*pi*Lx
145 END DO
```

```
146   !$OMP END PARALLEL DO
147   !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
148   DO j=1,1+Ny/2
149     ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
150   END DO
151   !$OMP END PARALLEL DO
152   ky(1+Ny/2)=0.0d0
153   !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
154   DO j = 1,Ny/2 -1
155     ky(j+1+Ny/2)=-ky(1-j+Ny/2)
156   END DO
157   !$OMP END PARALLEL DO
158   !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
159     DO j=1,Ny
160     y(j)=(-1.0d0+2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)) )*pi*Ly
161   END DO
162   !$OMP END PARALLEL DO
163   PRINT *,'Setup grid and fourier frequencies'
164   !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
165   DO j=1,Ny
166     DO i=1,Nx
167       unax(i,j)=exp(-1.0d0*(x(i)**2 +y(j)**2))
168     END DO
169   END DO
170   !$OMP END PARALLEL DO
171   name_config = 'uinitial.dat'
172   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
173   REWIND(11)
174   DO j=1,Ny
175     DO i=1,Nx
176       WRITE(11,*) abs(unax(i,j))**2
177     END DO
178   END DO
179   CLOSE(11)
180   !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
181   DO j=1,Ny
182     DO i=1,Nx
183       CALL dfftw_execute_dft_(planfx,unax(i,j),vnax(i,j))
184     END DO
185   END DO
186   !$OMP END PARALLEL DO
187   vnay(1:Ny,1:Nx)=TRANSPOSE(vnax(1:Nx,1:Ny))
188   ! transform initial data and do first half time step
189   !$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
190   DO i=1,Nx
191     CALL dfftw_execute_dft_(planfy,vnay(1:Ny,i),vnby(1:Ny,i))
192     DO j=1,Ny
193       vnby(j,i)=exp(0.5d0*dt*(kx(i)*kx(i) + ky(j)*ky(j))&
194           *cmplx(0.0d0,1.0d0))*vnby(j,i)
195     END DO
196     CALL dfftw_execute_dft_(planby,vnby(j,i),vnay(j,i))
```

```
197   END DO
198   !$OMP END PARALLEL DO
199   PRINT *,'Got initial data, starting timestepping'
200   time(1)=0.0d0
201   CALL system_clock(start,count_rate)
202   DO n=1,Nt
203     vnbx(1:Nx,1:Ny)=TRANSPOSE(vnay(1:Ny,1:Nx))/REAL(Ny,kind(0d0))
204     !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
205     DO j=1,Ny
206       CALL dfftw_execute_dft_(planbx,vnbx(1:Nx,j),unax(1:Nx,j))
207       DO i=1,Nx
208         unax(i,j)=unax(1:Nx,j)/REAL(Nx,kind(0d0))
209         potx(i,j)=Es*unax(i,j)*conjg(unax(i,j))
210         unax(i,j)=exp(cmplx(0.0d0,-1.0d0)*dt*potx(i,j))&
211               *unax(i,j)
212       END DO
213       CALL dfftw_execute_dft_(planfx,unax(1:Nx,j),vnax(1:Nx,j))
214     END DO
215     !$OMP END PARALLEL DO
216     vnby(1:Ny,1:Nx)=TRANSPOSE(vnax(1:Nx,1:Ny))
217     !$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
218     DO i=1,Nx
219       CALL dfftw_execute_dft_(planfy,vnby(1:Ny,i),vnay(1:Ny,i))
220       DO j=1,Ny
221         vnby(j,i)=exp(dt*(kx(i)*kx(i) + ky(j)*ky(j))&
222             *cmplx(0.0d0,1.0d0))*vnay(j,i)
223       END DO
224       CALL dfftw_execute_dft_(planby,vnby(1:Ny,i),vnay(1:Ny,i))
225     END DO
226     !$OMP END PARALLEL DO
227     IF (mod(n,plotgap)==0) then
228       time(1+n/plotgap)=n*dt
229       PRINT *,'time',n*dt
230     END IF
231   END DO
232   PRINT *,'Finished time stepping'
233   CALL system_clock(finish,count_rate)
234   PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),&
235     'for Time stepping'
236
237   ! transform back final data and do another half time step
238   vnbx(1:Nx,1:Ny)=transpose(vnay(1:Ny,1:Nx))/REAL(Ny,kind(0d0))
239   !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
240   DO j=1,Ny
241     CALL dfftw_execute_dft_(planbx,vnbx(1:Nx,j),unax(1:Nx,j))
242     unax(1:Nx,j)=unax(1:Nx,j)/REAL(Nx,kind(0d0))
243     potx(1:Nx,j)=Es*unax(1:Nx,j)*conjg(unax(1:Nx,j))
244     unax(1:Nx,j)=exp(cmplx(0,-1)*dt*potx(1:Nx,j))*unax(1:Nx,j)
245     CALL dfftw_execute_dft_(planfx,unax(1:Nx,j),vnax(1:Nx,j))
246   END DO
247   !$OMP END PARALLEL DO
```

```
248   vnby(1:Ny,1:Nx)=TRANSPOSE(vnax(1:Nx,1:Ny))
249   !$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
250   DO i=1,Nx
251     CALL dfftw_execute_dft_(planfy,vnby(1:Ny,i),vnay(1:Ny,i))
252     vnby(1:Ny,i)=exp(0.5d0*dt*(kx(i)*kx(i) + ky(1:Ny)*ky(1:Ny))&
253           *cmplx(0,1))*vnay(1:Ny,i)
254     CALL dfftw_execute_dft_(planby,vnby(1:Ny,i),vnay(1:Ny,i))
255   END DO
256   !$OMP END PARALLEL DO
257   vnbx(1:Nx,1:Ny)=TRANSPOSE(vnay(1:Ny,1:Nx))/REAL(Ny,kind(0d0))
258   !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
259   DO j=1,Ny
260     CALL dfftw_execute_dft_(planbx,vnbx(1:Nx,j),unax(1:Nx,j))
261     unax(1:Nx,j)=unax(1:Nx,j)/REAL(Nx,kind(0d0))
262   END DO
263   !$OMP END PARALLEL DO
264   name_config = 'ufinal.dat'
265   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
266   REWIND(11)
267   DO j=1,Ny
268     DO i=1,Nx
269       WRITE(11,*) abs(unax(i,j))**2
270     END DO
271   END DO
272   CLOSE(11)
273
274   name_config = 'tdata.dat'
275   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
276   REWIND(11)
277   DO j=1,1+Nt/plotgap
278     WRITE(11,*) time(j)
279   END DO
280   CLOSE(11)
281
282   name_config = 'xcoord.dat'
283   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
284   REWIND(11)
285   DO i=1,Nx
286     WRITE(11,*) x(i)
287   END DO
288   CLOSE(11)
289
290   name_config = 'ycoord.dat'
291   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
292   REWIND(11)
293   DO j=1,Ny
294     WRITE(11,*) y(j)
295   END DO
296   CLOSE(11)
297   PRINT *,'Saved data'
298
```

```
299   CALL dfftw_destroy_plan_(planbx)
300   CALL dfftw_destroy_plan_(planfx)
301   CALL dfftw_destroy_plan_(planby)
302   CALL dfftw_destroy_plan_(planfy)
303   CALL dfftw_cleanup_()
304
305   DEALLOCATE(unax,vnax,vnbx,potx, vnay,vnby,stat=allocatestatus)
306   IF (allocatestatus .ne. 0) STOP
307   PRINT *,'Deallocated memory'
308
309     PRINT *,'Program execution complete'
310   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 $-fopenmp$.

```
1  #define the complier
2  COMPILER = gfortran
3  # compilation settings, optimization, precision, parallelization
4  FLAGS = -O0 -fopenmp
5
6
7  # libraries
8  LIBS = -L/usr/local/lib -lfftw3 -lm
9  # source list for main program
10 SOURCES =  NLSsplitting.f90
11
12 test: $(SOURCES)
13     ${COMPILER} -o NLSsplitting $(FLAGS) $(SOURCES) $(LIBS)
14
15 clean:
16   rm *.o
17
18 clobber:
19   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

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

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

and

$$\Delta\phi = |\Psi|^2 - 1,$$

where Ψ is the, ϕ the 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

$$i\Psi_t + \Delta\Psi = 0$$

using the Fourier transform so that

$$\Psi(\delta t) = \exp\left(-i\Delta^2\delta t\right)\Psi(0)$$

Then solve

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

using the Fourier transform. Finally, solve

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

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

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

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

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

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

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

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

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
```

```
53  ! .. Vectors ..
54  !  kx       = fourier frequencies in x direction
55  !  ky       = fourier frequencies in y direction
56  !  x        = x locations
57  !  y        = y locations
58  !  time       = times at which save data
59  !  name_config    = array to store filename for data to be saved
60  !  fftfxy     = array to setup 2D Fourier transform
61  !  fftbxy     = array to setup 2D Fourier transform
62  !
63  ! REFERENCES
64  !
65  ! ACKNOWLEDGEMENTS
66  !
67  ! ACCURACY
68  !
69  ! ERROR INDICATORS AND WARNINGS
70  !
71  ! FURTHER COMMENTS
72  ! Check that the initial iterate is consistent with the
73  ! boundary conditions for the domain specified
74  !-----------------------------------------------------------------------
75  ! External routines required
76  !
77  ! External libraries required
78  ! FFTW3  -- Fast Fourier Transform in the West Library
79  !     (http://www.fftw.org/)
80  PROGRAM main
81  ! Declare variables
82  IMPLICIT NONE
83  INTEGER(kind=4), PARAMETER  ::  Nx=2**5
84  INTEGER(kind=4), PARAMETER  ::  Ny=2**5
85  INTEGER(kind=4), PARAMETER  ::  Nz=2**5
86  INTEGER(kind=4), PARAMETER  ::  Nt=50
87  INTEGER(kind=4), PARAMETER  ::  plotgap=10
88  REAL(kind=8), PARAMETER ::&
89    pi=3.14159265358979323846264338327950288419716939937510d0
90  REAL(kind=8), PARAMETER ::  Lx=2.0d0,Ly=2.0d0,Lz=2.0d0
91  REAL(kind=8), PARAMETER ::  Es=1.0d0
92  REAL(kind=8)  ::  dt=0.10d0/Nt
93  REAL(kind=8) :: modescalereal
94  COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  kx,ky,kz
95  REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  x,y,z
96  COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE  ::  unax,vnax,potx
97  REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  time
98  INTEGER(kind=4) ::  i,j,k,n,AllocateStatus,count,iol
99    ! timing
100 INTEGER(kind=4) ::  start, finish, count_rate
101     ! fftw variables
102 INTEGER(kind=8), PARAMETER  ::  FFTW_IN_PLACE = 8, FFTW_MEASURE = 0, &
103   FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32, FFTW_ESTIMATE = 64
```

```
104   INTEGER(kind=8),PARAMETER ::  FFTW_FORWARD = -1, FFTW_BACKWARD=1
105   INTEGER(kind=8) ::  planfxyz,planbxyz
106   CHARACTER*100 ::  name_config, number_file
107
108   CALL system_clock(start,count_rate)
109   ALLOCATE(unax(1:Nx,1:Ny,1:Nz),vnax(1:Nx,1:Ny,1:Nz),potx(1:Nx,1:Ny,1:Nz)
          ,&
110         kx(1:Nx),ky(1:Ny),kz(1:Nz),x(1:Nx),y(1:Ny),z(1:Nz),&
111         time(1:1+Nt/plotgap),stat=AllocateStatus)
112   IF (AllocateStatus .ne. 0) STOP
113   PRINT *,'allocated space'
114   modescalereal=1.0d0/REAL(Nx,KIND(0d0))
115   modescalereal=modescalereal/REAL(Ny,KIND(0d0))
116   modescalereal=modescalereal/REAL(Nz,KIND(0d0))
117
118   ! set up ffts
119   CALL dfftw_plan_dft_3d_(planfxyz,Nx,Ny,Nz,unax(1:Nx,1:Ny,1:Nz),&
120     vnax(1:Nx,1:Ny,1:Nz),FFTW_FORWARD,FFTW_ESTIMATE)
121   CALL dfftw_plan_dft_3d_(planbxyz,Nx,Ny,Nz,vnax(1:Nx,1:Ny,1:Nz),&
122     unax(1:Nx,1:Ny,1:Nz),FFTW_BACKWARD,FFTW_ESTIMATE)
123
124   PRINT *,'Setup FFTs'
125
126   ! setup fourier frequencies and grid points
127   DO i=1,1+Nx/2
128     kx(i)= cmplx(0.0d0,1.0)*REAL(i-1,kind(0d0))/Lx
129   END DO
130   kx(1+Nx/2)=0.0d0
131   DO i = 1,Nx/2 -1
132     kx(i+1+Nx/2)=-kx(1-i+Nx/2)
133   END DO
134     DO i=1,Nx
135     x(i)=(-1.0d0+2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)) )*pi*Lx
136   END DO
137   DO j=1,1+Ny/2
138     ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
139   END DO
140   ky(1+Ny/2)=0.0d0
141   DO j = 1,Ny/2 -1
142     ky(j+1+Ny/2)=-ky(1-j+Ny/2)
143   END DO
144     DO j=1,Ny
145     y(j)=(-1.0d0+2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)) )*pi*Ly
146   END DO
147   DO k=1,1+Nz/2
148     kz(k)= cmplx(0.0d0,1.0d0)*REAL(k-1,kind(0d0))/Lz
149   END DO
150   kz(1+Nz/2)=0.0d0
151   DO k = 1,Nz/2 -1
152     kz(k+1+Nz/2)=-kz(1-k+Nz/2)
153   END DO
```

```
154     DO k=1,Nz
155     z(k)=(-1.0d0+2.0d0*REAL(k-1,kind(0d0))/REAL(Nz,kind(0d0)) )*pi*Lz
156   END DO
157
158   PRINT *,'Setup grid and fourier frequencies'
159
160   DO k=1,Nz;  DO j=1,Ny; DO i=1,Nx
161     unax(i,j,k)=exp(-1.0d0*(x(i)**2 +y(j)**2+z(k)**2))
162   END DO; END DO; END DO
163
164   name_config = 'uinitial.dat'
165   INQUIRE(iolength=iol) unax(1,1,1)
166   OPEN(unit=11,FILE=name_config,form="unformatted", &
167        access="direct",recl=iol)
168   count=1
169   DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
170     WRITE(11,rec=count) unax(i,j,k)
171         count=count+1
172   END DO; END DO; END DO
173   CLOSE(11)
174
175   CALL dfftw_execute_dft_(planfxyz,unax(1:Nx,1:Ny,1:Nz),vnax(1:Nx,1:Ny,1:
        Nz))
176
177   PRINT *,'Got initial data, starting timestepping'
178   time(1)=0
179   DO n=1,Nt
180     DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
181         vnax(i,j,k)=exp(0.50d0*dt*&
182           (kz(k)*kz(k) + kx(i)*kx(i) + ky(j)*ky(j))&
183           *cmplx(0.0d0,1.0d0))*vnax(i,j,k)
184     END DO; END DO; END DO
185     CALL dfftw_execute_dft_(planbxyz,vnax(1:Nx,1:Ny,1:Nz),&
186       unax(1:Nx,1:Ny,1:Nz))
187
188     DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
189       unax(i,j,k)=unax(i,j,k)*modescalereal
190       potx(i,j,k)=Es*unax(i,j,k)*conjg(unax(i,j,k))
191       unax(i,j,k)=exp(cmplx(0.0d0,-1.0d0)*dt*potx(i,j,k))&
192           *unax(i,j,k)
193     END DO; END DO; END DO
194     CALL dfftw_execute_dft_(planfxyz,unax(1:Nx,1:Ny,1:Nz),&
195       vnax(1:Nx,1:Ny,1:Nz))
196
197     DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
198       vnax(i,j,k)=exp(0.5d0*dt*&
199           (kx(i)*kx(i) + ky(j)*ky(j)+ kz(k)*kz(k))&
200             *cmplx(0.0d0,1.0d0))*vnax(i,j,k)
201     END DO; END DO; END DO
202     IF (mod(n,plotgap)==0) THEN
203       time(1+n/plotgap)=n*dt
```

```
204       PRINT *,'time',n*dt
205       CALL dfftw_execute_dft_(planbxyz,vnax(1:Nx,1:Ny,1:Nz),unax(1:Nx,1:Ny
              ,1:Nz))
206       DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
207         unax(i,j,k)=unax(i,j,k)*modescalereal
208       END DO; END DO; END DO
209       name_config='./data/u'
210       WRITE(number_file,'(i0)') 10000000+1+n/plotgap
211       ind=index(name_config,' ') -1
212       name_config=name_config(1:ind)//numberfile
213       ind=index(name_config,' ') -1
214       name_config=name_config(1:ind)//'.dat'
215       OPEN(unit=11,FILE=name_config,status="UNKNOWN")
216       REWIND(11)
217       DO j=1,Ny
218         DO i=1,Nx
219           WRITE(11,*) abs(unax(i,j))**2
220         END DO
221       END DO
222       CLOSE(11)
223
224     END IF
225   END DO
226   PRINT *,'Finished time stepping'
227
228     ! transform back final data and do another half time step
229   CALL system_clock(finish,count_rate)
230   PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),'for
          execution'
231
232   name_config = 'tdata.dat'
233   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
234   REWIND(11)
235   DO j=1,1+Nt/plotgap
236     WRITE(11,*) time(j)
237   END DO
238   CLOSE(11)
239
240   name_config = 'xcoord.dat'
241   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
242   REWIND(11)
243   DO i=1,Nx
244     WRITE(11,*) x(i)
245   END DO
246   CLOSE(11)
247
248   name_config = 'ycoord.dat'
249   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
250   REWIND(11)
251   DO j=1,Ny
252     WRITE(11,*) y(j)
```

```
253   END DO
254   CLOSE(11)
255
256   name_config = 'zcoord.dat'
257   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
258   REWIND(11)
259   DO k=1,Nz
260     WRITE(11,*) z(k)
261   END DO
262   CLOSE(11)
263   PRINT *,'Saved data'
264
265   CALL dfftw_destroy_plan_(planbxyz)
266   CALL dfftw_destroy_plan_(planfxyz)
267   CALL dfftw_cleanup_()
268
269   DEALLOCATE(unax,vnax,potx,&
270         kx,ky,kz,x,y,z,&
271         time,stat=AllocateStatus)
272   IF (AllocateStatus .ne. 0) STOP
273
274   PRINT *,'Program execution complete'
275   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.

```
1  % A program to plot the computed results
2
3  clear all; format compact, format short,
4  set(0,'defaultaxesfontsize',18,'defaultaxeslinewidth',.9,...
5      'defaultlinelinewidth',3.5,'defaultpatchlinewidth',5.5);
6
7  % Load data
8  tdata=load('./tdata.dat');
9  x=load('./xcoord.dat');
10 y=load('./ycoord.dat');
11 z=load('./zcoord.dat');
12 Tsteps = length(tdata);
13
14 Nx = length(x); Nt = length(tdata);
15 Ny = length(y); Nz = length(z);
16 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.

```


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

```
7   !
8   !
9   ! PURPOSE
10  !
11  ! This program solves nonlinear Schrodinger equation in 3 dimensions
12  ! i*u_t+Es*|u|^2u+u_{xx}+u_{yy}+u_{zz}=0
13  ! using a second order time spectral splitting scheme
14  !
15  ! The boundary conditions are u(x=0,y,z)=u(2*Lx*\pi,y,z),
16  ! u(x,y=0,z)=u(x,y=2*Ly*\pi,z), u(x,y,z=0)=u(x,y,z=2*Lz*\pi)
17  ! The initial condition is u=exp(-x^2-y^2)
18  !
19  ! .. Parameters ..
20  !  Nx       = number of modes in x - power of 2 for FFT
21  !  Ny       = number of modes in y - power of 2 for FFT
22  !  Nz       = number of modes in z - power of 2 for FFT
23  !  Nt       = number of timesteps to take
24  !  Tmax       = maximum simulation time
25  !  plotgap      = number of timesteps between plots
26  !  pi = 3.14159265358979323846264338327950288419716939937510d0
27  !  Lx       = width of box in x direction
28  !  Ly       = width of box in y direction
29  !  Lz       = width of box in z direction
30  !  ES       = +1 for focusing and -1 for defocusing
31  ! .. Scalars ..
32  !  i        = loop counter in x direction
33  !  j        = loop counter in y direction
34  !  k        = loop counter in z direction
35  !  n        = loop counter for timesteps direction
36  !  allocatestatus = error indicator during allocation
37  !  start      = variable to record start time of program
38  !  finish     = variable to record end time of program
39  !  count_rate   = variable for clock count rate
40  !  dt       = timestep
41  !  modescalereal  = Number to scale after backward FFT
42  !  myid       = Process id
43  !  ierr       = error code
44  !  p_row      = number of rows for domain decomposition
45  !  p_col      = number of columns for domain decomposition
46  !  filesize     = total filesize
47  !  disp       = displacement to start writing data from
48  !  ind        = index in array to write
49  !  plotnum      = number of plot to save
50  !  numberfile   = number of the file to be saved to disk
51  !  stat       = error indicator when reading inputfile
52  ! .. Arrays ..
53  !  u        = approximate solution
54  !  v        = Fourier transform of approximate solution
55  !  pot        = potential
56  ! .. Vectors ..
57  !  kx       = fourier frequencies in x direction
```

```
58  !  ky       = fourier frequencies in y direction
59  !  kz       = fourier frequencies in z direction
60  !  x        = x locations
61  !  y        = y locations
62  !  z        = z locations
63  !  time       = times at which save data
64  !  nameconfig   = array to store filename for data to be saved
65  !  InputFileName  = name of the Input File
66  ! .. Special Structures ..
67  !  decomp     = contains information on domain decomposition
68  !         see http://www.2decomp.org/ for more information
69  !
70  ! REFERENCES
71  !
72  ! ACKNOWLEDGEMENTS
73  !
74  ! ACCURACY
75  !
76  ! ERROR INDICATORS AND WARNINGS
77  !
78  ! FURTHER COMMENTS
79  ! Check that the initial iterate is consistent with the
80  ! boundary conditions for the domain specified
81  !-----------------------------------------------------------------------
82  ! External routines required
83  !
84  ! External libraries required
85  ! 2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
86  !     (http://www.2decomp.org/index.html)
87  ! MPI library
88
89  PROGRAM main
90  USE decomp_2d
91  USE decomp_2d_fft
92  USE decomp_2d_io
93  USE MPI
94  ! Declare variables
95  IMPLICIT NONE
96  INTEGER(kind=4)    ::  Nx=2**5
97  INTEGER(kind=4)    ::  Ny=2**5
98  INTEGER(kind=4)    ::  Nz=2**5
99  INTEGER(kind=4)    ::  Nt=50
100 INTEGER(kind=4)    ::  plotgap=10
101 REAL(kind=8), PARAMETER ::&
102   pi=3.14159265358979323846264338327950288419716939937510d0
103 REAL(kind=8)     ::  Lx=2.0d0,Ly=2.0d0,Lz=2.0d0
104 REAL(kind=8)     ::  Es=1.0d0
105 REAL(kind=8)     ::  dt=0.0010d0
106 COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  kx,ky,kz
107 REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  x,y,z
108 COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE  ::  u,v,pot
```

```
109   REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  time
110   INTEGER(KIND=4), DIMENSION(1:5) ::  intcomm
111   REAL(KIND=8), DIMENSION(1:5)  ::  dpcomm
112   REAL(kind=8) :: modescalereal
113   INTEGER(kind=4) ::  i,j,k,n,AllocateStatus,stat
114   INTEGER(kind=4) :: myid,numprocs,ierr
115   TYPE(DECOMP_INFO) ::  decomp
116   INTEGER(kind=MPI_OFFSET_KIND) :: filesize, disp
117   INTEGER(kind=4) ::  p_row=0, p_col=0
118   INTEGER(kind=4) ::  start, finish, count_rate, ind, plotnum
119   CHARACTER*50  ::  nameconfig
120   CHARACTER*20  ::  numberfile, InputFileName
121       ! initialisation of MPI
122   CALL MPI_INIT(ierr)
123   CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
124   CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
125
126   IF(myid.eq.0) THEN
127     CALL GET_ENVIRONMENT_VARIABLE(NAME='inputfile',VALUE=InputFileName,
          STATUS=stat)
128   END IF
129   CALL MPI_BCAST(stat,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
130
131   IF(stat.NE.0) THEN
132     IF(myid.eq.0) THEN
133         PRINT*,"Need to set environment variable inputfile to the name of
              the &
134              file where the simulation parameters are set"
135     END IF
136     STOP
137   END IF
138   IF(myid.eq.0) THEN
139     InputFileName='./INPUTFILE'
140     OPEN(unit=11,FILE=trim(InputFileName),status="OLD")
141     REWIND(11)
142     READ(11,*) intcomm(1), intcomm(2), intcomm(3), intcomm(4), intcomm(5),
          &
143           dpcomm(1), dpcomm(2),  dpcomm(3),  dpcomm(4),  dpcomm(5)
144     CLOSE(11)
145     PRINT *,"NX ",intcomm(1)
146     PRINT *,"NY ",intcomm(2)
147     PRINT *,"NZ ",intcomm(3)
148     PRINT *,"NT ",intcomm(4)
149     PRINT *,"plotgap ",intcomm(5)
150     PRINT *,"Lx ",dpcomm(1)
151     PRINT *,"Ly ",dpcomm(2)
152     PRINT *,"Lz ",dpcomm(3)
153     PRINT *,"Es ",dpcomm(4)
154     PRINT *,"Dt ",dpcomm(5)
155     PRINT *,"Read inputfile"
156   END IF
```

```
157   CALL MPI_BCAST(dpcomm,5,MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD,ierr)
158   CALL MPI_BCAST(intcomm,5,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
159
160   Nx=intcomm(1)
161   Ny=intcomm(2)
162   Nz=intcomm(3)
163   Nt=intcomm(4)
164   plotgap=intcomm(5)
165   Lx=dpcomm(1)
166   Ly=dpcomm(2)
167   Lz=dpcomm(3)
168   Es=dpcomm(4)
169   DT=dpcomm(5)
170
171   ! initialisation of 2decomp
172   ! do automatic domain decomposition
173   CALL decomp_2d_init(Nx,Ny,Nz,p_row,p_col)
174   ! get information about domain decomposition choosen
175   CALL decomp_info_init(Nx,Ny,Nz,decomp)
176   ! initialise FFT library
177   CALL decomp_2d_fft_init
178   ALLOCATE(u(decomp%xst(1):decomp%xen(1),&
179           decomp%xst(2):decomp%xen(2),&
180           decomp%xst(3):decomp%xen(3)),&
181             v(decomp%zst(1):decomp%zen(1),&
182               decomp%zst(2):decomp%zen(2),&
183               decomp%zst(3):decomp%zen(3)),&
184             pot(decomp%xst(1):decomp%xen(1),&
185               decomp%xst(2):decomp%xen(2),&
186               decomp%xst(3):decomp%xen(3)),&
187         kx(1:Nx),ky(1:Ny),kz(1:Nz),&
188         x(1:Nx),y(1:Ny),z(1:Nz),&
189         time(1:1+Nt/plotgap),stat=AllocateStatus)
190   IF (AllocateStatus .ne. 0) STOP
191
192   IF (myid.eq.0) THEN
193     PRINT *,'allocated space'
194   END IF
195
196   modescalereal=1.0d0/REAL(Nx,KIND(0d0))
197   modescalereal=modescalereal/REAL(Ny,KIND(0d0))
198   modescalereal=modescalereal/REAL(Nz,KIND(0d0))
199
200   ! setup fourier frequencies and grid points
201   DO i=1,1+Nx/2
202     kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
203   END DO
204   kx(1+Nx/2)=0.0d0
205   DO i = 1,Nx/2 -1
206     kx(i+1+Nx/2)=-kx(1-i+Nx/2)
207   END DO
```

```
208     DO i=1,Nx
209     x(i)=(-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)))*pi*Lx
210   END DO
211   DO j=1,1+Ny/2
212     ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
213   END DO
214   ky(1+Ny/2)=0.0d0
215   DO j = 1,Ny/2 -1
216     ky(j+1+Ny/2)=-ky(1-j+Ny/2)
217   END DO
218     DO j=1,Ny
219     y(j)=(-1.0d0 + 2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)))*pi*Ly
220   END DO
221   DO k=1,1+Nz/2
222     kz(k)= cmplx(0.0d0,1.0d0)*REAL(k-1,kind(0d0))/Lz
223   END DO
224   kz(1+Nz/2)=0.0d0
225   DO k = 1,Nz/2 -1
226     kz(k+1+Nz/2)=-kz(1-k+Nz/2)
227   END DO
228     DO k=1,Nz
229     z(k)=(-1.0d0 + 2.0d0*REAL(k-1,kind(0d0))/REAL(Nz,kind(0d0)))*pi*Lz
230   END DO
231
232   IF (myid.eq.0) THEN
233     PRINT *,'Setup grid and fourier frequencies'
234   END IF
235
236   DO k=decomp%xst(3),decomp%xen(3)
237     DO j=decomp%xst(2),decomp%xen(2)
238       DO i=decomp%xst(1),decomp%xen(1)
239         u(i,j,k)=exp(-1.0d0*(x(i)**2 +y(j)**2+z(k)**2))
240       END DO
241     END DO
242   END DO
243
244   ! write out using 2DECOMP&FFT MPI-IO routines
245   nameconfig='./data/u'
246   plotnum=0
247   WRITE(numberfile,'(i0)') 10000000+plotnum
248   ind=index(nameconfig,' ') -1
249   nameconfig=nameconfig(1:ind)//numberfile
250   ind=index(nameconfig,' ') -1
251   nameconfig=nameconfig(1:ind)//'.datbin'
252   CALL decomp_2d_write_one(1,u,nameconfig)
253
254   CALL decomp_2d_fft_3d(u,v,DECOMP_2D_FFT_FORWARD)
255   IF (myid.eq.0) THEN
256     PRINT *,'Got initial data, starting timestepping'
257   END IF
258   CALL system_clock(start,count_rate)
```

```
259   time(1)=0
260   DO n=1,Nt
261     ! Use Strang splitting
262     DO k=decomp%zst(3),decomp%zen(3)
263       DO j=decomp%zst(2),decomp%zen(2)
264         DO i=decomp%zst(1),decomp%zen(1)
265           v(i,j,k)=exp(0.50d0*dt*&
266             (kz(k)*kz(k) + kx(i)*kx(i) + ky(j)*ky(j))&
267             *cmplx(0.0d0,1.0d0))*v(i,j,k)
268         END DO
269       END DO
270     END DO
271
272     CALL decomp_2d_fft_3d(v,u,DECOMP_2D_FFT_BACKWARD)
273
274     DO k=decomp%xst(3),decomp%xen(3)
275       DO j=decomp%xst(2),decomp%xen(2)
276         DO i=decomp%xst(1),decomp%xen(1)
277           u(i,j,k)=u(i,j,k)*modescalereal
278           pot(i,j,k)=Es*u(i,j,k)*conjg(u(i,j,k))
279           u(i,j,k)=exp(cmplx(0.0d0,-1.0d0)*dt*pot(i,j,k))*u(i,j,k)
280         END DO
281       END DO
282     END DO
283     CALL decomp_2d_fft_3d(u,v,DECOMP_2D_FFT_FORWARD)
284
285     DO k=decomp%zst(3),decomp%zen(3)
286       DO j=decomp%zst(2),decomp%zen(2)
287         DO i=decomp%zst(1),decomp%zen(1)
288           v(i,j,k)=exp(dt*0.5d0*&
289             (kx(i)*kx(i) +ky(j)*ky(j) +kz(k)*kz(k))&
290             *cmplx(0.0d0,1.0d0))*v(i,j,k)
291         END DO
292       END DO
293     END DO
294     IF (mod(n,plotgap)==0) THEN
295       time(1+n/plotgap)=n*dt
296       IF (myid.eq.0) THEN
297         PRINT *,'time',n*dt
298       END IF
299       CALL decomp_2d_fft_3d(v,u,DECOMP_2D_FFT_BACKWARD)
300       u=u*modescalereal
301       nameconfig='./data/u'
302       plotnum=plotnum+1
303       WRITE(numberfile,'(i0)') 10000000+plotnum
304       ind=index(nameconfig,' ') -1
305       nameconfig=nameconfig(1:ind)//numberfile
306       ind=index(nameconfig,' ') -1
307       nameconfig=nameconfig(1:ind)//'.datbin'
308       ! write out using 2DECOMP&FFT MPI-IO routines
309       CALL decomp_2d_write_one(1,u,nameconfig)
```

```
310       END IF
311     END DO
312     IF (myid.eq.0) THEN
313       PRINT *,'Finished time stepping'
314     END IF
315
316     CALL system_clock(finish,count_rate)
317
318     IF (myid.eq.0) THEN
319       PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),'for
            execution'
320     END IF
321
322     IF (myid.eq.0) THEN
323       ! Save times at which output was made in text format
324       nameconfig = './data/tdata.dat'
325       OPEN(unit=11,FILE=nameconfig,status="UNKNOWN")
326       REWIND(11)
327       DO j=1,1+Nt/plotgap
328         WRITE(11,*) time(j)
329       END DO
330       CLOSE(11)
331       ! Save x grid points in text format
332       nameconfig = './data/xcoord.dat'
333       OPEN(unit=11,FILE=nameconfig,status="UNKNOWN")
334       REWIND(11)
335       DO i=1,Nx
336         WRITE(11,*) x(i)
337       END DO
338       CLOSE(11)
339       ! Save y grid points in text format
340       nameconfig = './data/ycoord.dat'
341       OPEN(unit=11,FILE=nameconfig,status="UNKNOWN")
342       REWIND(11)
343       DO j=1,Ny
344         WRITE(11,*) y(j)
345       END DO
346       CLOSE(11)
347       ! Save z grid points in text format
348       nameconfig = './data/zcoord.dat'
349       OPEN(unit=11,FILE=nameconfig,status="UNKNOWN")
350       REWIND(11)
351       DO k=1,Nz
352         WRITE(11,*) z(k)
353       END DO
354       CLOSE(11)
355       PRINT *,'Saved data'
356     END IF
357
358     ! clean up
359       CALL decomp_2d_fft_finalize
```

```
360     CALL decomp_2d_finalize
361   DEALLOCATE(u,v,pot,&
362         kx,ky,kz,x,y,z,&
363         time,stat=AllocateStatus)
364   IF (AllocateStatus .ne. 0) STOP
365   IF (myid.eq.0) THEN
366       PRINT *,'Program execution complete'
367   END IF
368   CALL MPI_FINALIZE(ierr)
369   END PROGRAM main
```

12.9 Exercises

1) Write an MPI code using 2DECOMP&FFT to solve the Gross-Pitaevskii equation in three dimensions.

2) 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

$$\begin{aligned}&\rho\left[\frac{\omega^{n+1,k+1}-\omega^n}{\delta t}\right.\\&\left.+\frac{1}{2}\left(u^{n+1,k}\frac{\partial\omega^{n+1,k}}{\partial x}+v^{n+1,k}\frac{\partial\omega^{n+1,k}}{\partial y}+u^n\frac{\partial\omega^n}{\partial x}+v^n\frac{\partial\omega^n}{\partial y}\right)\right]\\&=\frac{\mu}{2}\Delta\left(\omega^{n+1,k+1}+\omega^n\right)+\left(\frac{\partial fx}{\partial y}-\frac{\partial fy}{\partial x}\right)\Big|_{t=(n+0.5)\delta t},\end{aligned} \tag{13.5}$$

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,

$$\begin{aligned}&\rho\left[\frac{\omega^{n+1,k+1}-\omega^n}{\delta t}\right.\\&\left.+\left(\frac{u^{n+1,k}+u^n}{2}\right)\frac{\partial}{\partial x}\left(\frac{\omega^{n+1,k}+\omega^n}{2}\right)+\left(\frac{v^{n+1,k}+v^n}{2}\right)\frac{\partial}{\partial y}\left(\frac{\omega^{n+1,k}+\omega^n}{2}\right)\right]\\&=\frac{\mu}{2}\Delta\left(\omega^{n+1,k+1}+\omega^n\right)+\left(\frac{\partial fx}{\partial y}-\frac{\partial fy}{\partial x}\right)\Big|_{t=(n+0.5)\delta t},\end{aligned} \tag{13.7}$$

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

$$u(x, y, t) = \sin(2\pi x)\cos(2\pi y)\exp(-8\pi^2\mu t) \tag{13.13}$$
$$v(x, y, t) = -\cos(2\pi x)\sin(2\pi y)\exp(-8\pi^2\mu t). \tag{13.14}$$

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
```

```
15
16 %grid
17 Nx=64; h=1/Nx; x=h*(1:Nx);
18 Ny=64; h=1/Ny; y=h*(1:Ny)';
19 [xx,yy]=meshgrid(x,y);
20
21 %initial conditions
22 u=sin(2*pi*xx).*cos(2*pi*yy);
23 v=-cos(2*pi*xx).*sin(2*pi*yy);
24 u_y=-2*pi*sin(2*pi*xx).*sin(2*pi*yy);
25 v_x=2*pi*sin(2*pi*xx).*sin(2*pi*yy);
26 omega=v_x-u_y;
27
28 dt=0.0025; t(1)=0; tmax=.1;
29 nplots=ceil(tmax/dt);
30
31 %wave numbers for derivatives
32 k_x=2*pi*(1i*[(0:Nx/2-1)  0 1-Nx/2:-1]');
33 k_y=2*pi*(1i*[(0:Ny/2-1)  0 1-Ny/2:-1]);
34 k2x=k_x.^2;
35 k2y=k_y.^2;
36
37 %wave number grid for multiplying matricies
38 [kxx,kyy]=meshgrid(k2x,k2y);
39 [kx,ky]=meshgrid(k_x,k_y);
40
41 % use a high tolerance so time stepping errors
42 % are not dominated by errors in solution to nonlinear
43 % system
44 tol=10^(-10);
45
46 %compute \hat{\omega}^{n+1,k}
47 omegahat=fft2(omega);
48 %nonlinear term
49 nonlinhat=fft2(u.*ifft2(omegahat.*kx)+v.*ifft2(omegahat.*ky));
50 for i=1:nplots
51     chg=1;
52     % save old values
53     uold=u; vold=v;  omegaold=omega; omegacheck=omega;
54     omegahatold=omegahat; nonlinhatold=nonlinhat;
55     while chg>tol
56         %nonlinear {n+1,k}
57         nonlinhat=fft2(u.*ifft2(omegahat.*kx)+v.*ifft2(omegahat.*ky));
58
59         %CrankNicolson timestepping
60         omegahat=((1/dt + 0.5*(1/Re)*(kxx+kyy)).*omegahatold...
61             -.5*(nonlinhatold+nonlinhat))...
62             ./(1/dt -0.5*(1/Re)*(kxx+kyy));
63
64         %compute \hat{\psi}^{n+1,k+1}
65         psihat=-omegahat./(kxx+kyy);
```

```
66
67          %NOTE: kxx+kyy has to be zero at the following points to avoid a
68          % discontinuity. However, we suppose that the streamfunction has
69          % mean value zero, so we set them equal to zero
70          psihat(1,1)=0;
71          psihat(Nx/2+1,Ny/2+1)=0;
72          psihat(Nx/2+1,1)=0;
73          psihat(1,Ny/2+1)=0;
74
75          %computes {\psi}_x by differentiation via FFT
76          dpsix = real(ifft2(psihat.*kx));
77          %computes {\psi}_y by differentiation via FFT
78          dpsiy = real(ifft2(psihat.*ky));
79
80          u=dpsiy;    %u^{n+1,k+1}
81          v=-dpsix;   %v^{n+1,k+1}
82
83          %\omega^{n+1,k+1}
84          omega=ifft2(omegahat);
85          % check for convergence
86          chg=max(max(abs(omega-omegacheck)))
87          % store omega to check for convergence of next iteration
88          omegacheck=omega;
89      end
90       t(i+1)=t(i)+dt;
91       uexact_y=-2*pi*sin(2*pi*xx).*sin(2*pi*yy).*exp(-8*pi^2*t(i+1)/Re);
92       vexact_x=2*pi*sin(2*pi*xx).*sin(2*pi*yy).*exp(-8*pi^2*t(i+1)/Re);
93       omegaexact=vexact_x-uexact_y;
94       figure(1); pcolor(omega);  xlabel x; ylabel y;
95       title Numerical; colorbar; drawnow;
96       figure(2); pcolor(omegaexact); xlabel x; ylabel y;
97       title Exact; colorbar; drawnow;
98       figure(3); pcolor(omega-omegaexact); xlabel x; ylabel y;
99       title Error; colorbar; drawnow;
100 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.

```
1 % A program to solve the 3D Navier stokes equations with periodic boundary
2 % conditions. The program is based on the Orszag-Patterson algorithm as
3 % documented on pg. 98 of C. Canuto, M.Y. Hussaini, A. Quarteroni and
4 % T.A. Zhang "Spectral Methods: Evolution to Complex Geometries and
5 % Applications to Fluid Dynamics" Springer (2007)
6 %
```

```
% 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);
```

```
58
59 uhat=fftn(u);
60 vhat=fftn(v);
61 what=fftn(w);
62
63 ux=ifftn(uhat.*kxm);uy=ifftn(uhat.*kym);uz=ifftn(uhat.*kzm);
64 vx=ifftn(vhat.*kxm);vy=ifftn(vhat.*kym);vz=ifftn(vhat.*kzm);
65 wx=ifftn(what.*kxm);wy=ifftn(what.*kym);wz=ifftn(what.*kzm);
66
67 % calculate vorticity for plotting
68 omegax=wy-vz; omegay=uz-wx; omegaz=vx-uy;
69 omegatot=omegax.^2+omegay.^2+omegaz.^2;
70 figure(1); clf; n=0;
71 subplot(2,2,1); title(['omega x ',num2str(n*dt)]);
72 p1 = patch(isosurface(x,y,z,omegax,.0025),...
73             'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
74 p2 = patch(isocaps(x,y,z,omegax,.0025),...
75             'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
76         isonormals(omegax,p1); lighting phong;
77 xlabel('x'); ylabel('y'); zlabel('z');
78 axis equal; axis square; view(3); colorbar;
79 subplot(2,2,2); title(['omega y ',num2str(n*dt)]);
80 p1 = patch(isosurface(x,y,z,omegay,.0025),...
81             'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
82 p2 = patch(isocaps(x,y,z,omegay,.0025),...
83             'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
84         isonormals(omegay,p1); lighting phong;
85 xlabel('x'); ylabel('y'); zlabel('z');
86 axis equal; axis square; view(3); colorbar;
87 subplot(2,2,3); title(['omega z ',num2str(n*dt)]);
88 p1 = patch(isosurface(x,y,z,omegaz,.0025),...
89             'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
90 p2 = patch(isocaps(x,y,z,omegaz,.0025),...
91             'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
92         isonormals(omegaz,p1); lighting phong;
93 xlabel('x'); ylabel('y'); zlabel('z');
94 axis equal; axis square; view(3); colorbar;
95 subplot(2,2,4); title(['|omega|^2 ',num2str(n*dt)]);
96 p1 = patch(isosurface(x,y,z,omegatot,.0025),...
97             'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
98 p2 = patch(isocaps(x,y,z,omegatot,.0025),...
99             'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
100         isonormals(omegatot,p1); lighting phong;
101 xlabel('x'); ylabel('y'); zlabel('z'); colorbar;
102 axis equal; axis square; view(3);
103
104
105 for n=1:Nt
106     uold=u; uxold=ux; uyold=uy; uzold=uz;
107     vold=v; vxold=vx; vyold=vy; vzold=vz;
108     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
```

```
45  !  planbx     = Backward 1d fft plan in x
46  !  planfy     = Forward 1d fft plan in y
47  !  planby     = Backward 1d fft plan in y
48  !  dt       = timestep
49  ! .. Arrays ..
50  !  u        = velocity in x direction
51  !  uold       = velocity in x direction at previous timestep
52  !  v        = velocity in y direction
53  !  vold       = velocity in y direction at previous timestep
54  !  u_y        = y derivative of velocity in x direction
55  !  v_x        = x derivative of velocity in y direction
56  !  omeg       = vorticity in real space
57  !  omegold      = vorticity in real space at previous
58  !           iterate
59  !  omegcheck    = store of vorticity at previous iterate
60  !  omegoldhat   = 2D Fourier transform of vorticity at previous
61  !           iterate
62  !  omegoldhat_x   = x-derivative of vorticity in Fourier space
63  !           at previous iterate
64  !  omegold_x    = x-derivative of vorticity in real space
65  !           at previous iterate
66  !  omegoldhat_y   = y-derivative of vorticity in Fourier space
67  !           at previous iterate
68  !  omegold_y    = y-derivative of vorticity in real space
69  !            at previous iterate
70  !  nlold      = nonlinear term in real space
71  !           at previous iterate
72  !  nloldhat     = nonlinear term in Fourier space
73  !           at previous iterate
74  !  omeghat      = 2D Fourier transform of vorticity
75  !           at next iterate
76  !  omeghat_x    = x-derivative of vorticity in Fourier space
77  !           at next timestep
78  !  omeghat_y    = y-derivative of vorticity in Fourier space
79  !           at next timestep
80  !  omeg_x     = x-derivative of vorticity in real space
81  !           at next timestep
82  !  omeg_y     = y-derivative of vorticity in real space
83  !           at next timestep
84  ! .. Vectors ..
85  !  kx       = fourier frequencies in x direction
86  !  ky       = fourier frequencies in y direction
87  !  kxx        = square of fourier frequencies in x direction
88  !  kyy        = square of fourier frequencies in y direction
89  !  x        = x locations
90  !  y        = y locations
91  !  time       = times at which save data
92  !  name_config    = array to store filename for data to be saved
93  !  fftfx      = array to setup x Fourier transform
94  !  fftbx      = array to setup y Fourier transform
95  ! REFERENCES
```

```
96  !
97  ! ACKNOWLEDGEMENTS
98  !
99  ! ACCURACY
100 !
101 ! ERROR INDICATORS AND WARNINGS
102 !
103 ! FURTHER COMMENTS
104 ! This program has not been optimized to use the least amount of memory
105 ! but is intended as an example only for which all states can be saved
106 !---------------------------------------------------------------------
107 ! External routines required
108 !
109 ! External libraries required
110 ! FFTW3  -- Fast Fourier Transform in the West Library
111 !     (http://www.fftw.org/)
112 ! declare variables
113
114 IMPLICIT NONE
115 INTEGER(kind=4), PARAMETER  ::  Nx=256
116 INTEGER(kind=4), PARAMETER  ::  Ny=256
117 REAL(kind=8), PARAMETER   ::  dt=0.00125
118 REAL(kind=8), PARAMETER &
119   ::  pi=3.14159265358979323846264338327950288419716939937510
120 REAL(kind=8), PARAMETER   ::  rho=1.0d0
121 REAL(kind=8), PARAMETER   ::  mu=1.0d0
122 REAL(kind=8), PARAMETER   ::  tol=0.1d0**10
123 REAL(kind=8)         ::  chg
124 INTEGER(kind=4), PARAMETER  ::  nplots=50
125 REAL(kind=8), DIMENSION(:), ALLOCATABLE     ::  time
126 COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE    ::  kx,kxx
127 COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE    ::  ky,kyy
128 REAL(kind=8), DIMENSION(:), ALLOCATABLE     ::  x
129 REAL(kind=8), DIMENSION(:), ALLOCATABLE     ::  y
130 COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE  :: &
131   u,uold,v,vold,u_y,v_x,omegold, omegcheck, omeg,&
132   omegoldhat, omegoldhat_x, omegold_x,&
133     omegoldhat_y, omegold_y, nlold, nloldhat,&
134   omeghat, omeghat_x, omeghat_y, omeg_x, omeg_y,&
135   nl, nlhat, psihat, psihat_x, psi_x, psihat_y, psi_y
136 REAL(kind=8),DIMENSION(:,:), ALLOCATABLE  ::  uexact_y,vexact_x,
      omegexact
137 INTEGER(kind=4)               ::  i,j,k,n, allocatestatus, count, iol
138 INTEGER(kind=4)               ::  start, finish, count_rate
139 INTEGER(kind=4), PARAMETER            ::  FFTW_IN_PLACE = 8,
      FFTW_MEASURE = 0, &
140                           FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32,    &
141                                   FFTW_ESTIMATE = 64
142   INTEGER(kind=4),PARAMETER              ::  FFTW_FORWARD = -1,
        FFTW_BACKWARD=1
143 COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE  ::  fftfx,fftbx
```

```
144  INTEGER(kind=8)                ::  planfxy,planbxy
145  CHARACTER*100                  ::  name_config
146
147  CALL system_clock(start,count_rate)
148  ALLOCATE(time(1:nplots),kx(1:Nx),kxx(1:Nx),ky(1:Ny),kyy(1:Ny),x(1:Nx),y
         (1:Ny),&
149       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),&
150       v_x(1:Nx,1:Ny),omegold(1:Nx,1:Ny),omegcheck(1:Nx,1:Ny), omeg(1:Nx,1:
             Ny),&
151       omegoldhat(1:Nx,1:Ny),omegoldhat_x(1:Nx,1:Ny), omegold_x(1:Nx,1:Ny),
              &
152       omegoldhat_y(1:Nx,1:Ny),omegold_y(1:Nx,1:Ny), nlold(1:Nx,1:Ny),
             nloldhat(1:Nx,1:Ny),&
153       omeghat(1:Nx,1:Ny), omeghat_x(1:Nx,1:Ny), omeghat_y(1:Nx,1:Ny),
             omeg_x(1:Nx,1:Ny),&
154       omeg_y(1:Nx,1:Ny), nl(1:Nx,1:Ny), nlhat(1:Nx,1:Ny), psihat(1:Nx,1:Ny
             ), &
155       psihat_x(1:Nx,1:Ny), psi_x(1:Nx,1:Ny), psihat_y(1:Nx,1:Ny), psi_y(1:
             Nx,1:Ny),&
156       uexact_y(1:Nx,1:Ny), vexact_x(1:Nx,1:Ny), omegexact(1:Nx,1:Ny),fftfx
             (1:Nx,1:Ny),&
157       fftbx(1:Nx,1:Ny),stat=AllocateStatus)
158  IF (AllocateStatus .ne. 0) STOP
159  PRINT *,'allocated space'
160
161  ! set up ffts
162  CALL dfftw_plan_dft_2d_(planfxy,Nx,Ny,fftfx(1:Nx,1:Ny),fftbx(1:Nx,1:Ny)
         ,&
163          FFTW_FORWARD,FFTW_EXHAUSTIVE)
164  CALL dfftw_plan_dft_2d_(planbxy,Nx,Ny,fftbx(1:Nx,1:Ny),fftfx(1:Nx,1:Ny)
         ,&
165       FFTW_BACKWARD,FFTW_EXHAUSTIVE)
166
167  ! setup fourier frequencies in x-direction
168  DO i=1,1+Nx/2
169    kx(i)= 2.0d0*pi*cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))
170  END DO
171  kx(1+Nx/2)=0.0d0
172  DO i = 1,Nx/2 -1
173    kx(i+1+Nx/2)=-kx(1-i+Nx/2)
174  END DO
175  DO i=1,Nx
176    kxx(i)=kx(i)*kx(i)
177  END DO
178  DO i=1,Nx
179    x(i)=REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0))
180  END DO
181
182  ! setup fourier frequencies in y-direction
183  DO j=1,1+Ny/2
```

```
184     ky(j)= 2.0d0*pi*cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))
185   END DO
186   ky(1+Ny/2)=0.0d0
187   DO j = 1,Ny/2 -1
188     ky(j+1+Ny/2)=-ky(1-j+Ny/2)
189   END DO
190   DO j=1,Ny
191     kyy(j)=ky(j)*ky(j)
192   END DO
193   DO j=1,Ny
194     y(j)=REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0))
195   END DO
196   PRINT *,'Setup grid and fourier frequencies'
197
198
199   DO j=1,Ny
200     DO i=1,Nx
201       u(i,j)=sin(2.0d0*pi*x(i))*cos(2.0d0*pi*y(j))
202       v(i,j)=-cos(2.0d0*pi*x(i))*sin(2.0d0*pi*y(j))
203       u_y(i,j)=-2.0d0*pi*sin(2.0d0*pi*x(i))*sin(2.0d0*pi*y(j))
204       v_x(i,j)=2.0d0*pi*sin(2.0d0*pi*x(i))*sin(2.0d0*pi*y(j))
205       omeg(i,j)=v_x(i,j)-u_y(i,j)
206     END DO
207   END DO
208
209   ! Vorticity to Fourier Space
210   CALL dfftw_execute_dft_(planfxy,omeg(1:Nx,1:Ny),omeghat(1:Nx,1:Ny))
211
212   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
213   !!!!!!!!!!!!!!!Initial nonlinear term !!!!!!!!!!!!!!!!!!!!!!!!!!
214     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
215   ! obtain \hat{\omega}_x^{n,k}
216   DO j=1,Ny
217     omeghat_x(1:Nx,j)=omeghat(1:Nx,j)*kx(1:Nx)
218   END DO
219   ! obtain \hat{\omega}_y^{n,k}
220   DO i=1,Nx
221     omeghat_y(i,1:Ny)=omeghat(i,1:Ny)*ky(1:Ny)
222   END DO
223   ! convert to real space
224   CALL dfftw_execute_dft_(planbxy,omeghat_x(1:Nx,1:Ny),omeg_x(1:Nx,1:Ny))
225   CALL dfftw_execute_dft_(planbxy,omeghat_y(1:Nx,1:Ny),omeg_y(1:Nx,1:Ny))
226   ! compute nonlinear term in real space
227   DO j=1,Ny
228     nl(1:Nx,j)=u(1:Nx,j)*omeg_x(1:Nx,j)/REAL(Nx*Ny,kind(0d0))+&
229                v(1:Nx,j)*omeg_y(1:Nx,j)/REAL(Nx*Ny,kind(0d0))
230   END DO
231   CALL dfftw_execute_dft_(planfxy,nl(1:Nx,1:Ny),nlhat(1:Nx,1:Ny))
232   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
233   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
234   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
235   time(1)=0.0d0
236   PRINT *,'Got initial data, starting timestepping'
237   DO n=1,nplots
238     chg=1
239     ! save old values
240     uold(1:Nx,1:Ny)=u(1:Nx,1:Ny)
241     vold(1:Nx,1:Ny)=v(1:Nx,1:Ny)
242     omegold(1:Nx,1:Ny)=omeg(1:Nx,1:Ny)
243     omegcheck(1:Nx,1:Ny)=omeg(1:Nx,1:Ny)
244     omegoldhat(1:Nx,1:Ny)=omeghat(1:Nx,1:Ny)
245     nloldhat(1:Nx,1:Ny)=nlhat(1:Nx,1:Ny)
246     DO WHILE (chg>tol)
247       !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
248       !!!!!!!!!!!!!!nonlinear fixed (n,k+1)!!!!!!!!!!!!!!!!!!!!!!!!!!!!
249       !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
250       ! obtain \hat{\omega}_x^{n+1,k}
251       DO j=1,Ny
252         omeghat_x(1:Nx,j)=omeghat(1:Nx,j)*kx(1:Nx)
253       END DO
254       ! obtain \hat{\omega}_y^{n+1,k}
255       DO i=1,Nx
256         omeghat_y(i,1:Ny)=omeghat(i,1:Ny)*ky(1:Ny)
257       END DO
258       ! convert back to real space
259       CALL dfftw_execute_dft_(planbxy,omeghat_x(1:Nx,1:Ny),omeg_x(1:Nx,1:
              Ny))
260       CALL dfftw_execute_dft_(planbxy,omeghat_y(1:Nx,1:Ny),omeg_y(1:Nx,1:
              Ny))
261       ! calculate nonlinear term in real space
262       DO j=1,Ny
263         nl(1:Nx,j)=u(1:Nx,j)*omeg_x(1:Nx,j)/REAL(Nx*Ny,kind(0d0))+&
264           v(1:Nx,j)*omeg_y(1:Nx,j)/REAL(Nx*Ny,kind(0d0))
265       END DO
266       ! convert back to fourier
267       CALL dfftw_execute_dft_(planfxy,nl(1:Nx,1:Ny),nlhat(1:Nx,1:Ny))
268       !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
269       !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
270       !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
271
272       ! obtain \hat{\omega}^{n+1,k+1} with Crank Nicolson timestepping
273       DO j=1,Ny
274         omeghat(1:Nx,j)=( (1.0d0/dt+0.5d0*(mu/rho)*(kxx(1:Nx)+kyy(j)))&
275             *omegoldhat(1:Nx,j) - 0.5d0*(nloldhat(1:Nx,j)+nlhat(1:Nx,j)))
                /&
276             (1.0d0/dt-0.5d0*(mu/rho)*(kxx(1:Nx)+kyy(j)))
277       END DO
278
279       ! calculate \hat{\psi}^{n+1,k+1}
280       DO j=1,Ny
281         psihat(1:Nx,j)=-omeghat(1:Nx,j)/(kxx(1:Nx)+kyy(j))
282       END DO
```

```
283       psihat(1,1)=0.0d0
284         psihat(Nx/2+1,Ny/2+1)=0.0d0
285       psihat(Nx/2+1,1)=0.0d0
286       psihat(1,Ny/2+1)=0.0d0
287
288       ! obtain \psi_x^{n+1,k+1} and \psi_y^{n+1,k+1}
289       DO j=1,Ny
290         psihat_x(1:Nx,j)=psihat(1:Nx,j)*kx(1:Nx)
291       END DO
292       CALL dfftw_execute_dft_(planbxy,psihat_x(1:Nx,1:Ny),psi_x(1:Nx,1:Ny)
              )
293       DO i=1,Nx
294         psihat_y(i,1:Ny)=psihat(i,1:Ny)*ky(1:Ny)
295       END DO
296       CALL dfftw_execute_dft_(planbxy,psihat_y(1:Ny,1:Ny),psi_y(1:Ny,1:Ny)
              )
297       DO j=1,Ny
298         psi_x(1:Nx,j)=psi_x(1:Nx,j)/REAL(Nx*Ny,kind(0d0))
299         psi_y(1:Nx,j)=psi_y(1:Nx,j)/REAL(Nx*Ny,kind(0d0))
300       END DO
301
302       ! obtain \omega^{n+1,k+1}
303       CALL dfftw_execute_dft_(planbxy,omeghat(1:Nx,1:Ny),omeg(1:Nx,1:Ny))
304       DO j=1,Ny
305         omeg(1:Nx,j)=omeg(1:Nx,j)/REAL(Nx*Ny,kind(0d0))
306       END DO
307
308       ! obtain u^{n+1,k+1} and v^{n+1,k+1} using stream function (\psi) in
              real space
309       DO j=1,Ny
310         u(1:Nx,j)=psi_y(1:Nx,j)
311         v(1:Nx,j)=-psi_x(1:Nx,j)
312       END DO
313
314       ! check for convergence
315       chg=maxval(abs(omeg-omegcheck))
316       ! saves {n+1,k+1} to {n,k} for next iteration
317       omegcheck=omeg
318     END DO
319     time(n+1)=time(n)+dt
320     PRINT *,'TIME ',time(n+1)
321   END DO
322
323   DO j=1,Ny
324     DO i=1,Nx
325       uexact_y(i,j)=-2.0d0*pi*sin(2.0d0*pi*x(i))*sin(2.0d0*pi*y(j))*&
326                 exp(-8.0d0*mu*(pi**2)*nplots*dt)
327       vexact_x(i,j)=2.0d0*pi*sin(2.0d0*pi*x(i))*sin(2.0d0*pi*y(j))*&
328                 exp(-8.0d0*mu*(pi**2)*nplots*dt)
329       omegexact(i,j)=vexact_x(i,j)-uexact_y(i,j)
330     END DO
```

```
331   END DO
332
333   name_config = 'omegafinal.datbin'
334   INQUIRE(iolength=iol) omegexact(1,1)
335   OPEN(unit=11,FILE=name_config,form="unformatted", access="direct",recl=
          iol)
336   count = 1
337   DO j=1,Ny
338     DO i=1,Nx
339       WRITE(11,rec=count) REAL(omeg(i,j),KIND(0d0))
340       count=count+1
341     END DO
342   END DO
343   CLOSE(11)
344
345   name_config = 'omegaexactfinal.datbin'
346   OPEN(unit=11,FILE=name_config,form="unformatted", access="direct",recl=
          iol)
347   count = 1
348   DO j=1,Ny
349     DO i=1,Nx
350       WRITE(11,rec=count) omegexact(i,j)
351       count=count+1
352     END DO
353   END DO
354   CLOSE(11)
355
356   name_config = 'xcoord.dat'
357   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
358   REWIND(11)
359   DO i=1,Nx
360     WRITE(11,*) x(i)
361   END DO
362   CLOSE(11)
363
364   name_config = 'ycoord.dat'
365   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
366   REWIND(11)
367   DO j=1,Ny
368     WRITE(11,*) y(j)
369   END DO
370   CLOSE(11)
371
372   CALL dfftw_destroy_plan_(planfxy)
373   CALL dfftw_destroy_plan_(planbxy)
374   CALL dfftw_cleanup_()
375
376   DEALLOCATE(time,kx,kxx,ky,kyy,x,y,&
377       u,uold,v,vold,u_y,v_x,omegold, omegcheck, omeg, &
378       omegoldhat, omegoldhat_x, omegold_x,&
379       omegoldhat_y, omegold_y, nlold, nloldhat,&
```

```
380       omeghat, omeghat_x, omeghat_y, omeg_x, omeg_y,&
381       nl, nlhat, psihat, psihat_x, psi_x, psihat_y, psi_y,&
382       uexact_y,vexact_x,omegexact, &
383       fftfx,fftbx,stat=AllocateStatus)
384   IF (AllocateStatus .ne. 0) STOP
385   PRINT *,'Program execution complete'
386   END PROGRAM main
```

Listing 13.4: A Matlab program to plot the vorticity fields and error produced by listing 13.3.

```
1  % A program to create a plot of the computed results
2  % from the 2D Matlab Navier-Stokes solver
3
4  clear all; format compact, format short,
5  set(0,'defaultaxesfontsize',14,'defaultaxeslinewidth',.7,...
6      'defaultlinelinewidth',2,'defaultpatchlinewidth',3.5);
7
8  % Load data
9  % Get coordinates
10 X=load('xcoord.dat');
11 Y=load('ycoord.dat');
12 % find number of grid points
13 Nx=length(X);
14 Ny=length(Y);
15
16 % reshape coordinates to allow easy plotting
17 [xx,yy]=ndgrid(X,Y);
18
19 %
20 % Open file and dataset using the default properties.
21 %
22 FILENUM=['omegafinal.datbin'];
23 FILEEXA=['omegaexactfinal.datbin'];
24 fidnum=fopen(FILENUM,'r');
25 [fnamenum,modenum,mformatnum]=fopen(fidnum);
26 fidexa=fopen(FILEEXA,'r');
27 [fnameexa,modeexa,mformatexa]=fopen(fidexa);
28 Num=fread(fidnum,Nx*Ny,'double',mformatnum);
29 Exa=fread(fidexa,Nx*Ny,'double',mformatexa);
30 Num=reshape(Num,Nx,Ny);
31 Exa=reshape(Exa,Nx,Ny);
32 % close files
33 fclose(fidnum);
34 fclose(fidexa);
35 %
36 % Plot data on the screen.
37 %
38 figure(2);clf;
39 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
```

```
35  !  k        = loop counter in z direction
36  !  n        = loop counter for timesteps direction
37  !  allocatestatus = error indicator during allocation
38  !  count      = keep track of information written to disk
39  !  iol        = size of array to write to disk
40  !  start      = variable to record start time of program
41  !  finish     = variable to record end time of program
42  !  count_rate   = variable for clock count rate
43  !  planfxyz     = Forward 3d fft plan
44  !  planbxyz     = Backward 3d fft plan
45  !  dt       = timestep
46  ! .. Arrays ..
47  !  u        = velocity in x direction
48  !  v        = velocity in y direction
49  !  w        = velocity in z direction
50  !  uold       = velocity in x direction at previous timestep
51  !  vold       = velocity in y direction at previous timestep
52  !  wold       = velocity in z direction at previous timestep
53  !  ux       = x derivative of velocity in x direction
54  !  uy       = y derivative of velocity in x direction
55  !  uz       = z derivative of velocity in x direction
56  !  vx       = x derivative of velocity in y direction
57  !  vy       = y derivative of velocity in y direction
58  !  vz       = z derivative of velocity in y direction
59  !  wx       = x derivative of velocity in z direction
60  !  wy       = y derivative of velocity in z direction
61  !  wz       = z derivative of velocity in z direction
62  !  uxold      = x derivative of velocity in x direction
63  !  uyold      = y derivative of velocity in x direction
64  !  uzold      = z derivative of velocity in x direction
65  !  vxold      = x derivative of velocity in y direction
66  !  vyold      = y derivative of velocity in y direction
67  !  vzold      = z derivative of velocity in y direction
68  !  wxold      = x derivative of velocity in z direction
69  !  wyold      = y derivative of velocity in z direction
70  !  wzold      = z derivative of velocity in z direction
71  !  omeg       = vorticity in real space
72  !  omegold      = vorticity in real space at previous
73  !           iterate
74  !  omegcheck    = store of vorticity at previous iterate
75  !  omegoldhat   = 2D Fourier transform of vorticity at previous
76  !           iterate
77  !  omegoldhat_x   = x-derivative of vorticity in Fourier space
78  !           at previous iterate
79  !  omegold_x    = x-derivative of vorticity in real space
80  !           at previous iterate
81  !  omegoldhat_y   = y-derivative of vorticity in Fourier space
82  !           at previous iterate
83  !  omegold_y    = y-derivative of vorticity in real space
84  !            at previous iterate
85  !  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
```

```
133     !declare variables
134     INTEGER(kind=4), PARAMETER     :: Nx=64
135   INTEGER(kind=4), PARAMETER     :: Ny=64
136   INTEGER(kind=4), PARAMETER     :: Nz=64
137     INTEGER(kind=4), PARAMETER     :: Lx=1
138   INTEGER(kind=4), PARAMETER     :: Ly=1
139   INTEGER(kind=4), PARAMETER     :: Lz=1
140   INTEGER(kind=4), PARAMETER     :: Nt=20
141   REAL(kind=8), PARAMETER      :: dt=0.2d0/Nt
142   REAL(kind=8), PARAMETER      :: Re=1.0d0
143   REAL(kind=8), PARAMETER      :: tol=0.1d0**10
144   REAL(kind=8), PARAMETER      :: theta=0.0d0
145
146   REAL(kind=8), PARAMETER &
147     ::  pi=3.14159265358979323846264338327950288419716939937510d0
148   REAL(kind=8), PARAMETER    ::  ReInv=1.0d0/REAL(Re,kind(0d0))
149   REAL(kind=8), PARAMETER    :: dtInv=1.0d0/REAL(dt,kind(0d0))
150   REAL(kind=8)                    :: scalemodes,chg,factor
151   REAL(kind=8), DIMENSION(:), ALLOCATABLE     :: x, y, z, time
152   COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE  :: u, v, w,&
153                            ux, uy, uz,&
154                            vx, vy, vz,&
155                            wx, wy, wz,&
156                            uold, uxold, uyold, uzold,&
157                            vold, vxold, vyold, vzold,&
158                            wold, wxold, wyold, wzold,&
159                            utemp, vtemp, wtemp, temp_r
160
161   COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE     :: kx, ky, kz
162   COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE  :: uhat, vhat, what,&
163                          rhsuhatfix, rhsvhatfix,&
164                          rhswhatfix, nonlinuhat,&
165                          nonlinvhat, nonlinwhat,&
166                          phat,temp_c
167   REAL(kind=8), DIMENSION(:,:,:), ALLOCATABLE    :: realtemp
168   !FFTW variables
169   INTEGER(kind=4)              :: ierr
170   INTEGER(kind=4), PARAMETER          :: FFTW_IN_PLACE = 8,&
171                        FFTW_MEASURE = 0,&
172                          FFTW_EXHAUSTIVE = 8,&
173                        FFTW_PATIENT = 32,&
174                                    FFTW_ESTIMATE = 64
175     INTEGER(kind=4),PARAMETER           :: FFTW_FORWARD = -1,&
176                        FFTW_BACKWARD=1
177   INTEGER(kind=8)              :: planfxyz,planbxyz
178
179   !variables used for saving data and timing
180   INTEGER(kind=4)              :: count, iol
181   INTEGER(kind=4)              :: i,j,k,n,t,allocatestatus
182   INTEGER(kind=4)              :: ind, numberfile
183   CHARACTER*100              :: name_config
```

```
184   INTEGER(kind=4)              ::  start, finish, count_rate
185
186     PRINT *,'Grid:',Nx,'X',Ny,'Y',Nz,'Z'
187   PRINT *,'dt:',dt
188   ALLOCATE(x(1:Nx),y(1:Ny),z(1:Nz),time(1:Nt+1),u(1:Nx,1:Ny,1:Nz),&
189         v(1:Nx,1:Ny,1:Nz), w(1:Nx,1:Ny,1:Nz), ux(1:Nx,1:Ny,1:Nz),&
190         uy(1:Nx,1:Ny,1:Nz), uz(1:Nx,1:Ny,1:Nz), vx(1:Nx,1:Ny,1:Nz),&
191         vy(1:Nx,1:Ny,1:Nz), vz(1:Nx,1:Ny,1:Nz), wx(1:Nx,1:Ny,1:Nz),&
192         wy(1:Nx,1:Ny,1:Nz), wz(1:Nx,1:Ny,1:Nz), uold(1:Nx,1:Ny,1:Nz),&
193         uxold(1:Nx,1:Ny,1:Nz), uyold(1:Nx,1:Ny,1:Nz), uzold(1:Nx,1:Ny,1:Nz
              ),&
194         vold(1:Nx,1:Ny,1:Nz), vxold(1:Nx,1:Ny,1:Nz), vyold(1:Nx,1:Ny,1:Nz)
              ,&
195         vzold(1:Nx,1:Ny,1:Nz), wold(1:Nx,1:Ny,1:Nz), wxold(1:Nx,1:Ny,1:Nz)
              ,&
196         wyold(1:Nx,1:Ny,1:Nz), wzold(1:Nx,1:Ny,1:Nz), utemp(1:Nx,1:Ny,1:Nz
              ),&
197         vtemp(1:Nx,1:Ny,1:Nz), wtemp(1:Nx,1:Ny,1:Nz), temp_r(1:Nx,1:Ny,1:
              Nz),&
198         kx(1:Nx),ky(1:Ny),kz(1:Nz),uhat(1:Nx,1:Ny,1:Nz), vhat(1:Nx,1:Ny,1:
              Nz),&
199         what(1:Nx,1:Ny,1:Nz), rhsuhatfix(1:Nx,1:Ny,1:Nz),&
200         rhsvhatfix(1:Nx,1:Ny,1:Nz), rhswhatfix(1:Nx,1:Ny,1:Nz),&
201         nonlinuhat(1:Nx,1:Ny,1:Nz), nonlinvhat(1:Nx,1:Ny,1:Nz),&
202         nonlinwhat(1:Nx,1:Ny,1:Nz), phat(1:Nx,1:Ny,1:Nz),temp_c(1:Nx,1:Ny
              ,1:Nz),&
203         realtemp(1:Nx,1:Ny,1:Nz), stat=AllocateStatus)
204   IF (AllocateStatus .ne. 0) STOP
205   PRINT *,'allocated space'
206
207   CALL dfftw_plan_dft_3d_(planfxyz,Nx,Ny,Nz,temp_r(1:Nx,1:Ny,1:Nz),&
208       temp_c(1:Nx,1:Ny,1:Nz),FFTW_FORWARD,FFTW_ESTIMATE)
209   CALL dfftw_plan_dft_3d_(planbxyz,Nx,Ny,Nz,temp_c(1:Nx,1:Ny,1:Nz),&
210       temp_r(1:Nx,1:Ny,1:Nz),FFTW_BACKWARD,FFTW_ESTIMATE)
211   PRINT *,'Setup 3D FFTs'
212
213   ! setup fourier frequencies in x-direction
214   DO i=1,Nx/2+1
215     kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
216   END DO
217   kx(1+Nx/2)=0.0d0
218   DO i = 1,Nx/2 -1
219     kx(i+1+Nx/2)=-kx(1-i+Nx/2)
220   END DO
221   ind=1
222   DO i=-Nx/2,Nx/2-1
223     x(ind)=2.0d0*pi*REAL(i,kind(0d0))*Lx/REAL(Nx,kind(0d0))
224     ind=ind+1
225   END DO
226   ! setup fourier frequencies in y-direction
227   DO j=1,Ny/2+1
```

```
228       ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
229     END DO
230     ky(1+Ny/2)=0.0d0
231     DO j = 1,Ny/2 -1
232       ky(j+1+Ny/2)=-ky(1-j+Ny/2)
233     END DO
234     ind=1
235     DO j=-Ny/2,Ny/2-1
236       y(ind)=2.0d0*pi*REAL(j,kind(0d0))*Ly/REAL(Ny,kind(0d0))
237       ind=ind+1
238     END DO
239     ! setup fourier frequencies in z-direction
240     DO k=1,Nz/2+1
241       kz(k)= cmplx(0.0d0,1.0d0)*REAL(k-1,kind(0d0))/Lz
242     END DO
243     kz(1+Nz/2)=0.0d0
244     DO k = 1,Nz/2 -1
245       kz(k+1+Nz/2)=-kz(1-k+Nz/2)
246     END DO
247     ind=1
248     DO k=-Nz/2,Nz/2-1
249       z(ind)=2.0d0*pi*REAL(k,kind(0d0))*Lz/REAL(Nz,kind(0d0))
250       ind=ind+1
251     END DO
252     scalemodes=1.0d0/REAL(Nx*Ny*Nz,kind(0d0))
253     PRINT *,'Setup grid and fourier frequencies'
254
255     !initial conditions for Taylor-Green vortex
256   ! factor=2.0d0/sqrt(3.0d0)
257   ! DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
258   !   u(i,j,k)=factor*sin(theta+2.0d0*pi/3.0d0)*sin(x(i))*cos(y(j))*cos(z(k)
      )
259   ! END DO; END DO; END DO
260   ! DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
261   !   v(i,j,k)=factor*sin(theta-2.0d0*pi/3.0d0)*cos(x(i))*sin(y(j))*cos(z(k)
      )
262   ! END DO ; END DO ; END DO
263   ! DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
264   !   w(i,j,k)=factor*sin(theta)*cos(x(i))*cos(y(j))*sin(z(k))
265   ! END DO ; END DO ; END DO
266
267     ! Initial conditions for exact solution
268     time(1)=0.0d0
269     factor=sqrt(3.0d0)
270     DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
271       u(i,j,k)=-0.5*( factor*cos(x(i))*sin(y(j))*sin(z(k))&
272                +sin(x(i))*cos(y(j))*cos(z(k)) )*exp(-(factor**2)*time(1)/Re)
273     END DO; END DO; END DO
274     DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
275       v(i,j,k)=0.5*(  factor*sin(x(i))*cos(y(j))*sin(z(k))&
276                -cos(x(i))*sin(y(j))*cos(z(k)) )*exp(-(factor**2)*time(1)/Re)
```

```
277   END DO ; END DO ; END DO
278   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
279     w(i,j,k)=cos(x(i))*cos(y(j))*sin(z(k))*exp(-(factor**2)*time(1)/Re)
280   END DO ; END DO ; END DO
281
282   CALL dfftw_execute_dft_(planfxyz,u(1:Nx,1:Ny,1:Nz),uhat(1:Nx,1:Ny,1:Nz))
283   CALL dfftw_execute_dft_(planfxyz,v(1:Nx,1:Ny,1:Nz),vhat(1:Nx,1:Ny,1:Nz))
284   CALL dfftw_execute_dft_(planfxyz,w(1:Nx,1:Ny,1:Nz),what(1:Nx,1:Ny,1:Nz))
285
286   ! derivative of u with respect to x, y, and z
287   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
288     temp_c(i,j,k)=uhat(i,j,k)*kx(i)*scalemodes
289   END DO ; END DO ; END DO
290   CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),ux(1:Nx,1:Ny,1:
        Nz))
291   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
292     temp_c(i,j,k)=uhat(i,j,k)*ky(j)*scalemodes
293   END DO ; END DO ; END DO
294   CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),uy(1:Nx,1:Ny,1:
        Nz))
295   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
296     temp_c(i,j,k)=uhat(i,j,k)*kz(k)*scalemodes
297   END DO ; END DO ; END DO
298   CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),uz(1:Nx,1:Ny,1:
        Nz))
299
300   ! derivative of v with respect to x, y, and z
301   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
302     temp_c(i,j,k)=vhat(i,j,k)*kx(i)*scalemodes
303   END DO ; END DO ; END DO
304   CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),vx(1:Nx,1:Ny,1:
        Nz))
305   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
306     temp_c(i,j,k)=vhat(i,j,k)*ky(j)*scalemodes
307   END DO ; END DO ; END DO
308   CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),vy(1:Nx,1:Ny,1:
        Nz))
309   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
310     temp_c(i,j,k)=vhat(i,j,k)*kz(k)*scalemodes
311   END DO ; END DO ; END DO
312   CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),vz(1:Nx,1:Ny,1:
        Nz))
313
314   ! derivative of w with respect to x, y, and z
315   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
316     temp_c(i,j,k)=what(i,j,k)*kx(i)*scalemodes
317   END DO ; END DO ; END DO
318   CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),wx(1:Nx,1:Ny,1:
        Nz))
319   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
320     temp_c(i,j,k)=what(i,j,k)*ky(j)*scalemodes
```

```
321   END DO ; END DO ; END DO
322   CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),wy(1:Nx,1:Ny,1:
          Nz))
323   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
324     temp_c(i,j,k)=what(i,j,k)*kz(k)*scalemodes
325   END DO ; END DO ; END DO
326   CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),wz(1:Nx,1:Ny,1:
          Nz))
327   ! save initial data
328   time(1)=0.0
329   n=0
330   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
331     realtemp(i,j,k)=REAL(wy(i,j,k)-vz(i,j,k),KIND=8)
332   END DO ; END DO ; END DO
333   name_config='./data/omegax'
334   CALL savedata(Nx,Ny,Nz,n,name_config,realtemp)
335   !omegay
336   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
337     realtemp(i,j,k)=REAL(uz(i,j,k)-wx(i,j,k),KIND=8)
338   END DO ; END DO ; END DO
339   name_config='./data/omegay'
340   CALL savedata(Nx,Ny,Nz,n,name_config,realtemp)
341   !omegaz
342   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
343     realtemp(i,j,k)=REAL(vx(i,j,k)-uy(i,j,k),KIND=8)
344   END DO ; END DO ; END DO
345   name_config='./data/omegaz'
346   CALL savedata(Nx,Ny,Nz,n,name_config,realtemp)
347
348   DO n=1,Nt
349     !fixed point
350     DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
351       uold(i,j,k)=u(i,j,k)
352       uxold(i,j,k)=ux(i,j,k)
353       uyold(i,j,k)=uy(i,j,k)
354       uzold(i,j,k)=uz(i,j,k)
355     END DO ; END DO ; END DO
356     DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
357       vold(i,j,k)=v(i,j,k)
358       vxold(i,j,k)=vx(i,j,k)
359       vyold(i,j,k)=vy(i,j,k)
360       vzold(i,j,k)=vz(i,j,k)
361     END DO ; END DO ; END DO
362     DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
363       wold(i,j,k)=w(i,j,k)
364       wxold(i,j,k)=wx(i,j,k)
365       wyold(i,j,k)=wy(i,j,k)
366       wzold(i,j,k)=wz(i,j,k)
367     END DO ; END DO ; END DO
368     DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
369       rhsuhatfix(i,j,k) = (dtInv+(0.5d0*ReInv)*&
```

```
370       (kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))*uhat(i,j,k)
371     END DO ; END DO ; END DO
372     DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
373       rhsvhatfix(i,j,k) = (dtInv+(0.5d0*ReInv)*&
374       (kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))*vhat(i,j,k)
375     END DO ; END DO ; END DO
376     DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
377       rhswhatfix(i,j,k) = (dtInv+(0.5d0*ReInv)*&
378       (kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))*what(i,j,k)
379     END DO ; END DO ; END DO
380
381     chg=1
382     DO WHILE (chg .gt. tol)
383       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
384         temp_r(i,j,k)=0.25d0*((u(i,j,k)+uold(i,j,k))*(ux(i,j,k)+uxold(i,j,
              k))&
385                   +(v(i,j,k)+vold(i,j,k))*(uy(i,j,k)+uyold(i,j,k))&
386                   +(w(i,j,k)+wold(i,j,k))*(uz(i,j,k)+uzold(i,j,k)))
387       END DO ; END DO ; END DO
388       CALL dfftw_execute_dft_(planfxyz,temp_r(1:Nx,1:Ny,1:Nz),nonlinuhat
            (1:Nx,1:Ny,1:Nz))
389       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
390         temp_r(i,j,k)=0.25d0*((u(i,j,k)+uold(i,j,k))*(vx(i,j,k)+vxold(i,j,
              k))&
391                   +(v(i,j,k)+vold(i,j,k))*(vy(i,j,k)+vyold(i,j,k))&
392                   +(w(i,j,k)+wold(i,j,k))*(vz(i,j,k)+vzold(i,j,k)))
393       END DO ; END DO ; END DO
394       CALL dfftw_execute_dft_(planfxyz,temp_r(1:Nx,1:Ny,1:Nz),nonlinvhat
            (1:Nx,1:Ny,1:Nz))
395       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
396         temp_r(i,j,k)=0.25d0*((u(i,j,k)+uold(i,j,k))*(wx(i,j,k)+wxold(i,j,
              k))&
397                   +(v(i,j,k)+vold(i,j,k))*(wy(i,j,k)+wyold(i,j,k))&
398                   +(w(i,j,k)+wold(i,j,k))*(wz(i,j,k)+wzold(i,j,k)))
399       END DO ; END DO ; END DO
400       CALL dfftw_execute_dft_(planfxyz,temp_r(1:Nx,1:Ny,1:Nz),nonlinwhat
            (1:Nx,1:Ny,1:Nz))
401       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
402         phat(i,j,k)=-1.0d0*( kx(i)*nonlinuhat(i,j,k)&
403                   +ky(j)*nonlinvhat(i,j,k)&
404                   +kz(k)*nonlinwhat(i,j,k))&
405                   /(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)+0.1d0**13)
406       END DO ; END DO ; END DO
407
408       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
409         uhat(i,j,k)=(rhsuhatfix(i,j,k)-nonlinuhat(i,j,k)-kx(i)*phat(i,j,k)
              )/&
410               (dtInv-(0.5d0*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))
                    !*scalemodes
411       END DO ; END DO ; END DO
412       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
```

```
413         vhat(i,j,k)=(rhsvhatfix(i,j,k)-nonlinvhat(i,j,k)-ky(j)*phat(i,j,k)
               )/&
414            (dtInv-(0.5d0*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))
                  !*scalemodes
415       END DO ; END DO ; END DO
416       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
417         what(i,j,k)=(rhswhatfix(i,j,k)-nonlinwhat(i,j,k)-kz(k)*phat(i,j,k)
               )/&
418            (dtInv-(0.5d0*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))
                  !*scalemodes
419       END DO ; END DO ; END DO
420
421       ! derivative of u with respect to x, y, and z
422       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
423         temp_c(i,j,k)=uhat(i,j,k)*kx(i)*scalemodes
424       END DO ; END DO ; END DO
425       CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),ux(1:Nx,1:Ny
              ,1:Nz))
426       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
427         temp_c(i,j,k)=uhat(i,j,k)*ky(j)*scalemodes
428       END DO ; END DO ; END DO
429       CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),uy(1:Nx,1:Ny
              ,1:Nz))
430       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
431         temp_c(i,j,k)=uhat(i,j,k)*kz(k)*scalemodes
432       END DO ; END DO ; END DO
433       CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),uz(1:Nx,1:Ny
              ,1:Nz))
434
435       ! derivative of v with respect to x, y, and z
436       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
437         temp_c(i,j,k)=vhat(i,j,k)*kx(i)*scalemodes
438       END DO ; END DO ; END DO
439       CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),vx(1:Nx,1:Ny
              ,1:Nz))
440       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
441         temp_c(i,j,k)=vhat(i,j,k)*ky(j)*scalemodes
442       END DO ; END DO ; END DO
443       CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),vy(1:Nx,1:Ny
              ,1:Nz))
444       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
445         temp_c(i,j,k)=vhat(i,j,k)*kz(k)*scalemodes
446       END DO ; END DO ; END DO
447       CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),vz(1:Nx,1:Ny
              ,1:Nz))
448
449       ! derivative of w with respect to x, y, and z
450       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
451         temp_c(i,j,k)=what(i,j,k)*kx(i)*scalemodes
452       END DO ; END DO ; END DO
```

```
453       CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),wx(1:Nx,1:Ny
              ,1:Nz))
454       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
455         temp_c(i,j,k)=what(i,j,k)*ky(j)*scalemodes
456       END DO ; END DO ; END DO
457       CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),wy(1:Nx,1:Ny
              ,1:Nz))
458       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
459         temp_c(i,j,k)=what(i,j,k)*kz(k)*scalemodes
460       END DO ; END DO ; END DO
461       CALL dfftw_execute_dft_(planbxyz,temp_c(1:Nx,1:Ny,1:Nz),wz(1:Nx,1:Ny
              ,1:Nz))
462
463       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
464         utemp(i,j,k)=u(i,j,k)
465       END DO ; END DO ; END DO
466       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
467         vtemp(i,j,k)=v(i,j,k)
468       END DO ; END DO ; END DO
469       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
470         wtemp(i,j,k)=w(i,j,k)
471       END DO ; END DO ; END DO
472
473       CALL dfftw_execute_dft_(planbxyz,uhat(1:Nx,1:Ny,1:Nz),u(1:Nx,1:Ny,1:
              Nz))
474       CALL dfftw_execute_dft_(planbxyz,vhat(1:Nx,1:Ny,1:Nz),v(1:Nx,1:Ny,1:
              Nz))
475       CALL dfftw_execute_dft_(planbxyz,what(1:Nx,1:Ny,1:Nz),w(1:Nx,1:Ny,1:
              Nz))
476
477       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
478         u(i,j,k)=u(i,j,k)*scalemodes
479       END DO ; END DO ; END DO
480       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
481         v(i,j,k)=v(i,j,k)*scalemodes
482       END DO ; END DO ; END DO
483       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
484         w(i,j,k)=w(i,j,k)*scalemodes
485       END DO ; END DO ; END DO
486
487       chg =maxval(abs(utemp-u))+maxval(abs(vtemp-v))+maxval(abs(wtemp-w))
488       PRINT *,'chg:',chg
489     END DO
490     time(n+1)=n*dt
491     PRINT *,'time',n*dt
492     !NOTE: utemp, vtemp, and wtemp are just temporary space that can be
            used
493     !     instead of creating new arrays.
494     !omegax
495     DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
496       realtemp(i,j,k)=REAL(wy(i,j,k)-vz(i,j,k),KIND=8)
```

```
497       END DO ; END DO ; END DO
498       name_config='./data/omegax'
499       CALL savedata(Nx,Ny,Nz,n,name_config,realtemp)
500       !omegay
501       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
502         realtemp(i,j,k)=REAL(uz(i,j,k)-wx(i,j,k),KIND=8)
503       END DO ; END DO ; END DO
504       name_config='./data/omegay'
505       CALL savedata(Nx,Ny,Nz,n,name_config,realtemp)
506       !omegaz
507       DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
508         realtemp(i,j,k)=REAL(vx(i,j,k)-uy(i,j,k),KIND=8)
509       END DO ; END DO ; END DO
510       name_config='./data/omegaz'
511       CALL savedata(Nx,Ny,Nz,n,name_config,realtemp)
512     END DO
513
514     name_config = './data/tdata.dat'
515     OPEN(unit=11,FILE=name_config,status="UNKNOWN")
516     REWIND(11)
517     DO n=1,1+Nt
518       WRITE(11,*) time(n)
519     END DO
520     CLOSE(11)
521
522     name_config = './data/xcoord.dat'
523     OPEN(unit=11,FILE=name_config,status="UNKNOWN")
524     REWIND(11)
525     DO i=1,Nx
526       WRITE(11,*) x(i)
527     END DO
528     CLOSE(11)
529
530     name_config = './data/ycoord.dat'
531     OPEN(unit=11,FILE=name_config,status="UNKNOWN")
532     REWIND(11)
533     DO j=1,Ny
534       WRITE(11,*) y(j)
535     END DO
536     CLOSE(11)
537
538     name_config = './data/zcoord.dat'
539     OPEN(unit=11,FILE=name_config,status="UNKNOWN")
540     REWIND(11)
541     DO k=1,Nz
542       WRITE(11,*) z(k)
543     END DO
544     CLOSE(11)
545     PRINT *,'Saved data'
546
547     ! Calculate error in final numerical solution
```

```
548   DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
549     utemp(i,j,k)=u(i,j,k) -&
550             (-0.5*( factor*cos(x(i))*sin(y(j))*sin(z(k))&
551             +sin(x(i))*cos(y(j))*cos(z(k)) )*exp(-(factor**2)*time(Nt+1)/
                   Re))
552   END DO; END DO; END DO
553   DO k=1,Nz; DO j=1,Ny; DO i=1,Nx
554     vtemp(i,j,k)=v(i,j,k) -&
555           (0.5*(  factor*sin(x(i))*cos(y(j))*sin(z(k))&
556             -cos(x(i))*sin(y(j))*cos(z(k)) )*exp(-(factor**2)*time(Nt+1)/
                   Re))
557   END DO ; END DO ; END DO
558   DO k=1,Nz ; DO j=1,Ny ; DO i=1,Nx
559     wtemp(i,j,k)=w(i,j,k)-&
560           (cos(x(i))*cos(y(j))*sin(z(k))*exp(-(factor**2)*time(Nt+1)/Re))
561   END DO ; END DO ; END DO
562   chg=maxval(abs(utemp))+maxval(abs(vtemp))+maxval(abs(wtemp))
563   PRINT*,'The error at the final timestep is',chg
564
565   CALL dfftw_destroy_plan_(planfxyz)
566   CALL dfftw_destroy_plan_(planbxyz)
567   DEALLOCATE(x,y,z,time,u,v,w,ux,uy,uz,vx,vy,vz,wx,wy,wz,uold,uxold,uyold,
         uzold,&
568           vold,vxold,vyold,vzold,wold,wxold,wyold,wzold,utemp,vtemp,wtemp
                ,&
569           temp_r,kx,ky,kz,uhat,vhat,what,rhsuhatfix,rhsvhatfix,&
570           rhswhatfix,phat,nonlinuhat,nonlinvhat,nonlinwhat,temp_c,&
571           realtemp,stat=AllocateStatus)
572   IF (AllocateStatus .ne. 0) STOP
573   PRINT *,'Program execution complete'
574   END PROGRAM main
```

Listing 13.6: A Matlab program to plot the vorticity fields produced by listing 13.5.

```
1  % A program to create a plot of the computed results
2  % from the 3D Fortran Navier-Stokes solver
3  clear all; format compact; format short;
4  set(0,'defaultaxesfontsize',30,'defaultaxeslinewidth',.7,...
5      'defaultlinelinewidth',6,'defaultpatchlinewidth',3.7,...
6      'defaultaxesfontweight','bold')
7  % Load data
8  % Get coordinates
9  tdata=load('./data/tdata.dat');
10 x=load('./data/xcoord.dat');
11 y=load('./data/ycoord.dat');
12 z=load('./data/zcoord.dat');
13 nplots = length(tdata);
14
15 Nx = length(x); Nt = length(tdata);
16 Ny = length(y); Nz = length(z);
```

```
17
18 % reshape coordinates to allow easy plotting
19 [yy,xx,zz]=meshgrid(x,y,z);
20
21 for i =1:nplots
22     %
23     % Open file and dataset using the default properties.
24     %
25     FILEX=['./data/omegax',num2str(9999999+i),'.datbin'];
26     FILEY=['./data/omegay',num2str(9999999+i),'.datbin'];
27     FILEZ=['./data/omegaz',num2str(9999999+i),'.datbin'];
28     FILEPIC=['./data/pic',num2str(9999999+i),'.jpg'];
29     fid=fopen(FILEX,'r');
30     [fname,mode,mformat]=fopen(fid);
31     omegax=fread(fid,Nx*Ny*Nz,'real*8');
32     omegax=reshape(omegax,Nx,Ny,Nz);
33     fclose(fid);
34     fid=fopen(FILEY,'r');
35     [fname,mode,mformat]=fopen(fid);
36     omegay=fread(fid,Nx*Ny*Nz,'real*8');
37     omegay=reshape(omegay,Nx,Ny,Nz);
38     fclose(fid);
39     fid=fopen(FILEZ,'r');
40     [fname,mode,mformat]=fopen(fid);
41     omegaz=fread(fid,Nx*Ny*Nz,'real*8');
42     omegaz=reshape(omegaz,Nx,Ny,Nz);
43     fclose(fid);
44     %
45     % Plot data on the screen.
46     %
47     omegatot=omegax.^2+omegay.^2+omegaz.^2;
48     figure(100); clf;
49     subplot(2,2,1); title(['omega x ',num2str(tdata(i))]);
50     p1 = patch(isosurface(xx,yy,zz,omegax,.0025),...
51             'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
52     p2 = patch(isocaps(xx,yy,zz,omegax,.0025),...
53             'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
54         isonormals(omegax,p1); lighting phong;
55     xlabel('x'); ylabel('y'); zlabel('z');
56     axis equal; axis square; view(3); colorbar;
57     subplot(2,2,2); title(['omega y ',num2str(tdata(i))]);
58     p1 = patch(isosurface(xx,yy,zz,omegay,.0025),...
59             'FaceColor','interp','EdgeColor','none','FaceAlpha',0.3);
60     p2 = patch(isocaps(xx,yy,zz,omegay,.0025),...
61             'FaceColor','interp','EdgeColor','none','FaceAlpha',0.1);
62         isonormals(omegay,p1); lighting phong;
63     xlabel('x'); ylabel('y'); zlabel('z');
64     axis equal; axis square; view(3); colorbar;
65     subplot(2,2,3); title(['omega z ',num2str(tdata(i))]);
66     p1 = patch(isosurface(xx,yy,zz,omegaz,.0025),...
67             '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.

```
1  PROGRAM main
2  !
       ------------------------------------------------------------------------------
3  !
4  !
5  ! PURPOSE
6  !
7  ! This program numerically solves the 3D incompressible Navier-Stokes
8  ! on a Cubic Domain [0,2pi]x[0,2pi]x[0,2pi] using pseudo-spectral
       methods and
9  ! Implicit Midpoint rule timestepping. The numerical solution is
       compared to
10 ! an exact solution reported by Shapiro
11 !
12 ! Analytical Solution:
13 ! u(x,y,z,t)=-0.25*(cos(x)sin(y)sin(z)+sin(x)cos(y)cos(z))exp(-t/Re)
14 ! v(x,y,z,t)= 0.25*(sin(x)cos(y)sin(z)-cos(x)sin(y)cos(z))exp(-t/Re)
15 ! w(x,y,z,t)= 0.5*cos(x)cos(y)sin(z)exp(-t/Re)
16 !
17 ! .. Parameters ..
18 !  Nx        = number of modes in x - power of 2 for FFT
19 !  Ny        = number of modes in y - power of 2 for FFT
20 !  Nz        = number of modes in z - power of 2 for FFT
21 !  Nt        = number of timesteps to take
22 !  Tmax        = maximum simulation time
23 !  FFTW_IN_PLACE  = value for FFTW input
24 !  FFTW_MEASURE   = value for FFTW input
25 !  FFTW_EXHAUSTIVE  = value for FFTW input
26 !  FFTW_PATIENT   = value for FFTW input
27 !  FFTW_ESTIMATE  = value for FFTW input
28 !  FFTW_FORWARD     = value for FFTW input
29 !  FFTW_BACKWARD  = value for FFTW input
30 !  pi = 3.14159265358979323846264338327950288419716939937510d0
31 !  Re        = Reynolds number
32 ! .. Scalars ..
33 !  i         = loop counter in x direction
34 !  j         = loop counter in y direction
35 !  k         = loop counter in z direction
36 !  n         = loop counter for timesteps direction
37 !  allocatestatus = error indicator during allocation
38 !  count       = keep track of information written to disk
39 !  iol         = size of array to write to disk
40 !  start       = variable to record start time of program
41 !  finish      = variable to record end time of program
42 !  count_rate    = variable for clock count rate
43 !  planfxyz      = Forward 3d fft plan
44 !  planbxyz      = Backward 3d fft plan
45 !  dt        = timestep
46 ! .. Arrays ..
```

```
47  !  u         = velocity in x direction
48  !  v         = velocity in y direction
49  !  w         = velocity in z direction
50  !  uold        = velocity in x direction at previous timestep
51  !  vold        = velocity in y direction at previous timestep
52  !  wold        = velocity in z direction at previous timestep
53  !  ux        = x derivative of velocity in x direction
54  !  uy        = y derivative of velocity in x direction
55  !  uz        = z derivative of velocity in x direction
56  !  vx        = x derivative of velocity in y direction
57  !  vy        = y derivative of velocity in y direction
58  !  vz        = z derivative of velocity in y direction
59  !  wx        = x derivative of velocity in z direction
60  !  wy        = y derivative of velocity in z direction
61  !  wz        = z derivative of velocity in z direction
62  !  uxold       = x derivative of velocity in x direction
63  !  uyold       = y derivative of velocity in x direction
64  !  uzold       = z derivative of velocity in x direction
65  !  vxold       = x derivative of velocity in y direction
66  !  vyold       = y derivative of velocity in y direction
67  !  vzold       = z derivative of velocity in y direction
68  !  wxold       = x derivative of velocity in z direction
69  !  wyold       = y derivative of velocity in z direction
70  !  wzold       = z derivative of velocity in z direction
71  !  utemp       = temporary storage of u to check convergence
72  !  vtemp       = temporary storage of u to check convergence
73  !  wtemp       = temporary storage of u to check convergence
74  !  temp_r      = temporary storage for untransformed variables
75  !  uhat        = Fourier transform of u
76  !  vhat        = Fourier transform of v
77  !  what        = Fourier transform of w
78  !  rhsuhatfix    = Fourier transform of righthand side for u for
        timestepping
79  !  rhsvhatfix    = Fourier transform of righthand side for v for
        timestepping
80  !  rhswhatfix    = Fourier transform of righthand side for w for
        timestepping
81  !  nonlinuhat    = Fourier transform of nonlinear term for u
82  !  nonlinvhat      = Fourier transform of nonlinear term for u
83  !  nonlinwhat    = Fourier transform of nonlinear term for u
84  !  phat        = Fourier transform of nonlinear term for pressure, p
85  !  temp_c      = temporary storage for Fourier transforms
86  !  realtemp      = Real storage
87  !
88  ! .. Vectors ..
89  !  kx        = fourier frequencies in x direction
90  !  ky        = fourier frequencies in y direction
91  !  kz        = fourier frequencies in z direction
92  !  x         = x locations
93  !  y         = y locations
94  !  z         = y locations
```

```
95  !  time        = times at which save data
96  !  name_config    = array to store filename for data to be saved
97  !
98  ! REFERENCES
99  !
100 ! A. Shapiro " The use of an exact solution of the Navier-Stokes
       equations
101 ! in a validation test of a three-dimensional nonhydrostatic numerical
       model"
102 ! Monthly Weather Review vol. 121, 2420-2425, (1993).
103 !
104 ! ACKNOWLEDGEMENTS
105 !
106 ! ACCURACY
107 !
108 ! ERROR INDICATORS AND WARNINGS
109 !
110 ! FURTHER COMMENTS
111 !
112 ! This program has not been optimized to use the least amount of memory
113 ! but is intended as an example only for which all states can be saved
114 !
115 !
       ---------------------------------------------------------------------------

116 ! External routines required
117 !
118 ! External libraries required
119 ! 2DECOMP&FFT -- Fast Fourier Transform in the West Library
120 !     (http://2decomp.org/)
121
122 USE decomp_2d
123 USE decomp_2d_fft
124 USE decomp_2d_io
125 USE MPI
126 IMPLICIT NONE
127 ! declare variables
128   INTEGER(kind=4), PARAMETER    :: Nx=256
129 INTEGER(kind=4), PARAMETER    :: Ny=256
130 INTEGER(kind=4), PARAMETER    :: Nz=256
131   INTEGER(kind=4), PARAMETER    :: Lx=1
132 INTEGER(kind=4), PARAMETER    :: Ly=1
133 INTEGER(kind=4), PARAMETER    :: Lz=1
134 INTEGER(kind=4), PARAMETER    :: Nt=20
135 REAL(kind=8), PARAMETER     :: dt=0.05d0/Nt
136 REAL(kind=8), PARAMETER     :: Re=1.0d0
137 REAL(kind=8), PARAMETER     :: tol=0.1d0**10
138 REAL(kind=8), PARAMETER     :: theta=0.0d0
139
140 REAL(kind=8), PARAMETER &
141   ::  pi=3.14159265358979323846264338327950288419716939937510d0
```

```
142   REAL(kind=8), PARAMETER    ::  ReInv=1.0d0/REAL(Re,kind(0d0))
143   REAL(kind=8), PARAMETER    ::  dtInv=1.0d0/REAL(dt,kind(0d0))
144   REAL(kind=8)         :: scalemodes,chg,factor
145   REAL(kind=8), DIMENSION(:), ALLOCATABLE     :: x, y, z, time,mychg,
          allchg
146   COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE  :: u, v, w,&
147                             ux, uy, uz,&
148                             vx, vy, vz,&
149                             wx, wy, wz,&
150                             uold, uxold, uyold, uzold,&
151                             vold, vxold, vyold, vzold,&
152                             wold, wxold, wyold, wzold,&
153                             utemp, vtemp, wtemp, temp_r
154
155   COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE    ::  kx, ky, kz
156   COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE  ::  uhat, vhat, what,&
157                               rhsuhatfix, rhsvhatfix,&
158                             rhswhatfix, nonlinuhat,&
159                             nonlinvhat, nonlinwhat,&
160                             phat,temp_c
161   REAL(kind=8), DIMENSION(:,:,:), ALLOCATABLE   ::  realtemp
162   ! MPI and 2DECOMP variables
163   TYPE(DECOMP_INFO)                 ::  decomp
164   INTEGER(kind=MPI_OFFSET_KIND)           ::  filesize, disp
165   INTEGER(kind=4)                   ::  p_row=0, p_col=0, numprocs, myid,
          ierr
166
167   ! variables used for saving data and timing
168   INTEGER(kind=4)              :: count, iol
169   INTEGER(kind=4)              :: i,j,k,n,t,allocatestatus
170   INTEGER(kind=4)              :: ind, numberfile
171   CHARACTER*100              :: name_config
172   INTEGER(kind=4)              :: start, finish, count_rate
173
174   ! initialisation of 2DECOMP&FFT
175   CALL MPI_INIT(ierr)
176   CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
177   CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
178   ! do automatic domain decomposition
179   CALL decomp_2d_init(Nx,Ny,Nz,p_row,p_col)
180   ! get information about domain decomposition choosen
181   CALL decomp_info_init(Nx,Ny,Nz,decomp)
182   ! initialise FFT library
183   CALL decomp_2d_fft_init
184   IF (myid.eq.0) THEN
185       PRINT *,'Grid:',Nx,'X',Ny,'Y',Nz,'Z'
186     PRINT *,'dt:',dt
187   END IF
188   ALLOCATE(x(1:Nx),y(1:Ny),z(1:Nz),time(1:Nt+1),mychg(1:3),allchg(1:3),&
189         u(decomp%xst(1):decomp%xen(1),&
190             decomp%xst(2):decomp%xen(2),&
```

```
191             decomp%xst(3):decomp%xen(3)),&
192         v(decomp%xst(1):decomp%xen(1),&
193             decomp%xst(2):decomp%xen(2),&
194             decomp%xst(3):decomp%xen(3)),&
195         w(decomp%xst(1):decomp%xen(1),&
196             decomp%xst(2):decomp%xen(2),&
197             decomp%xst(3):decomp%xen(3)),&
198         ux(decomp%xst(1):decomp%xen(1),&
199             decomp%xst(2):decomp%xen(2),&
200             decomp%xst(3):decomp%xen(3)),&
201         uy(decomp%xst(1):decomp%xen(1),&
202             decomp%xst(2):decomp%xen(2),&
203             decomp%xst(3):decomp%xen(3)),&
204         uz(decomp%xst(1):decomp%xen(1),&
205             decomp%xst(2):decomp%xen(2),&
206             decomp%xst(3):decomp%xen(3)),&
207         vx(decomp%xst(1):decomp%xen(1),&
208             decomp%xst(2):decomp%xen(2),&
209             decomp%xst(3):decomp%xen(3)),&
210         vy(decomp%xst(1):decomp%xen(1),&
211             decomp%xst(2):decomp%xen(2),&
212             decomp%xst(3):decomp%xen(3)),&
213         vz(decomp%xst(1):decomp%xen(1),&
214             decomp%xst(2):decomp%xen(2),&
215             decomp%xst(3):decomp%xen(3)),&
216         wx(decomp%xst(1):decomp%xen(1),&
217             decomp%xst(2):decomp%xen(2),&
218             decomp%xst(3):decomp%xen(3)),&
219         wy(decomp%xst(1):decomp%xen(1),&
220             decomp%xst(2):decomp%xen(2),&
221             decomp%xst(3):decomp%xen(3)),&
222         wz(decomp%xst(1):decomp%xen(1),&
223             decomp%xst(2):decomp%xen(2),&
224             decomp%xst(3):decomp%xen(3)),&
225         uold(decomp%xst(1):decomp%xen(1),&
226             decomp%xst(2):decomp%xen(2),&
227             decomp%xst(3):decomp%xen(3)),&
228         uxold(decomp%xst(1):decomp%xen(1),&
229             decomp%xst(2):decomp%xen(2),&
230             decomp%xst(3):decomp%xen(3)),&
231         uyold(decomp%xst(1):decomp%xen(1),&
232             decomp%xst(2):decomp%xen(2),&
233             decomp%xst(3):decomp%xen(3)),&
234         uzold(decomp%xst(1):decomp%xen(1),&
235             decomp%xst(2):decomp%xen(2),&
236             decomp%xst(3):decomp%xen(3)),&
237         vold(decomp%xst(1):decomp%xen(1),&
238             decomp%xst(2):decomp%xen(2),&
239             decomp%xst(3):decomp%xen(3)),&
240         vxold(decomp%xst(1):decomp%xen(1),&
241             decomp%xst(2):decomp%xen(2),&
```

```
242             decomp%xst(3):decomp%xen(3)),&
243         vyold(decomp%xst(1):decomp%xen(1),&
244             decomp%xst(2):decomp%xen(2),&
245             decomp%xst(3):decomp%xen(3)),&
246         vzold(decomp%xst(1):decomp%xen(1),&
247             decomp%xst(2):decomp%xen(2),&
248             decomp%xst(3):decomp%xen(3)),&
249         wold(decomp%xst(1):decomp%xen(1),&
250             decomp%xst(2):decomp%xen(2),&
251             decomp%xst(3):decomp%xen(3)),&
252         wxold(decomp%xst(1):decomp%xen(1),&
253             decomp%xst(2):decomp%xen(2),&
254             decomp%xst(3):decomp%xen(3)),&
255         wyold(decomp%xst(1):decomp%xen(1),&
256             decomp%xst(2):decomp%xen(2),&
257             decomp%xst(3):decomp%xen(3)),&
258         wzold(decomp%xst(1):decomp%xen(1),&
259             decomp%xst(2):decomp%xen(2),&
260             decomp%xst(3):decomp%xen(3)),&
261         utemp(decomp%xst(1):decomp%xen(1),&
262             decomp%xst(2):decomp%xen(2),&
263             decomp%xst(3):decomp%xen(3)),&
264         vtemp(decomp%xst(1):decomp%xen(1),&
265             decomp%xst(2):decomp%xen(2),&
266             decomp%xst(3):decomp%xen(3)),&
267         wtemp(decomp%xst(1):decomp%xen(1),&
268             decomp%xst(2):decomp%xen(2),&
269             decomp%xst(3):decomp%xen(3)),&
270         temp_r(decomp%xst(1):decomp%xen(1),&
271             decomp%xst(2):decomp%xen(2),&
272             decomp%xst(3):decomp%xen(3)),&
273         kx(1:Nx),ky(1:Ny),kz(1:Nz),&
274         uhat(decomp%zst(1):decomp%zen(1),&
275             decomp%zst(2):decomp%zen(2),&
276             decomp%zst(3):decomp%zen(3)),&
277         vhat(decomp%zst(1):decomp%zen(1),&
278             decomp%zst(2):decomp%zen(2),&
279             decomp%zst(3):decomp%zen(3)),&
280         what(decomp%zst(1):decomp%zen(1),&
281             decomp%zst(2):decomp%zen(2),&
282             decomp%zst(3):decomp%zen(3)),&
283         rhsuhatfix(decomp%zst(1):decomp%zen(1),&
284             decomp%zst(2):decomp%zen(2),&
285             decomp%zst(3):decomp%zen(3)),&
286         rhsvhatfix(decomp%zst(1):decomp%zen(1),&
287             decomp%zst(2):decomp%zen(2),&
288             decomp%zst(3):decomp%zen(3)),&
289         rhswhatfix(decomp%zst(1):decomp%zen(1),&
290             decomp%zst(2):decomp%zen(2),&
291             decomp%zst(3):decomp%zen(3)),&
292         nonlinuhat(decomp%zst(1):decomp%zen(1),&
```

```
293             decomp%zst(2):decomp%zen(2),&
294             decomp%zst(3):decomp%zen(3)),&
295         nonlinvhat(decomp%zst(1):decomp%zen(1),&
296             decomp%zst(2):decomp%zen(2),&
297             decomp%zst(3):decomp%zen(3)),&
298         nonlinwhat(decomp%zst(1):decomp%zen(1),&
299             decomp%zst(2):decomp%zen(2),&
300             decomp%zst(3):decomp%zen(3)),&
301         phat(decomp%zst(1):decomp%zen(1),&
302             decomp%zst(2):decomp%zen(2),&
303             decomp%zst(3):decomp%zen(3)),&
304         temp_c(decomp%zst(1):decomp%zen(1),&
305             decomp%zst(2):decomp%zen(2),&
306             decomp%zst(3):decomp%zen(3)),&
307         realtemp(decomp%xst(1):decomp%xen(1),&
308             decomp%xst(2):decomp%xen(2),&
309             decomp%xst(3):decomp%xen(3)), stat=AllocateStatus)
310   IF (AllocateStatus .ne. 0) STOP
311   IF (myid.eq.0) THEN
312     PRINT *,'allocated space'
313   END IF
314
315   ! setup fourier frequencies in x-direction
316   DO i=1,Nx/2+1
317     kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
318   END DO
319   kx(1+Nx/2)=0.0d0
320   DO i = 1,Nx/2 -1
321     kx(i+1+Nx/2)=-kx(1-i+Nx/2)
322   END DO
323   ind=1
324   DO i=-Nx/2,Nx/2-1
325     x(ind)=2.0d0*pi*REAL(i,kind(0d0))*Lx/REAL(Nx,kind(0d0))
326     ind=ind+1
327   END DO
328   ! setup fourier frequencies in y-direction
329   DO j=1,Ny/2+1
330     ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
331   END DO
332   ky(1+Ny/2)=0.0d0
333   DO j = 1,Ny/2 -1
334     ky(j+1+Ny/2)=-ky(1-j+Ny/2)
335   END DO
336   ind=1
337   DO j=-Ny/2,Ny/2-1
338     y(ind)=2.0d0*pi*REAL(j,kind(0d0))*Ly/REAL(Ny,kind(0d0))
339     ind=ind+1
340   END DO
341   ! setup fourier frequencies in z-direction
342   DO k=1,Nz/2+1
343     kz(k)= cmplx(0.0d0,1.0d0)*REAL(k-1,kind(0d0))/Lz
```

```
344   END DO
345   kz(1+Nz/2)=0.0d0
346   DO k = 1,Nz/2 -1
347     kz(k+1+Nz/2)=-kz(1-k+Nz/2)
348   END DO
349   ind=1
350   DO k=-Nz/2,Nz/2-1
351     z(ind)=2.0d0*pi*REAL(k,kind(0d0))*Lz/REAL(Nz,kind(0d0))
352     ind=ind+1
353   END DO
354   scalemodes=1.0d0/REAL(Nx*Ny*Nz,kind(0d0))
355   IF (myid.eq.0) THEN
356     PRINT *,'Setup grid and fourier frequencies'
357   END IF
358
359   !initial conditions for Taylor-Green vortex
360 ! factor=2.0d0/sqrt(3.0d0)
361 ! DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
      =decomp%xst(1),decomp%xen(1)
362 !   u(i,j,k)=factor*sin(theta+2.0d0*pi/3.0d0)*sin(x(i))*cos(y(j))*cos(z(k)
      )
363 ! END DO; END DO; END DO
364 ! DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
      =decomp%xst(1),decomp%xen(1)
365 !   v(i,j,k)=factor*sin(theta-2.0d0*pi/3.0d0)*cos(x(i))*sin(y(j))*cos(z(k)
      )
366 ! END DO ; END DO ; END DO
367 ! DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
      =decomp%xst(1),decomp%xen(1)
368 !   w(i,j,k)=factor*sin(theta)*cos(x(i))*cos(y(j))*sin(z(k))
369 ! END DO ; END DO ; END DO
370
371   time(1)=0.0d0
372   factor=sqrt(3.0d0)
373   DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
        =decomp%xst(1),decomp%xen(1)
374     u(i,j,k)=-0.5*( factor*cos(x(i))*sin(y(j))*sin(z(k))&
375             +sin(x(i))*cos(y(j))*cos(z(k)) )*exp(-(factor**2)*time(1)/Re)
376   END DO; END DO; END DO
377   DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
        =decomp%xst(1),decomp%xen(1)
378     v(i,j,k)=0.5*(  factor*sin(x(i))*cos(y(j))*sin(z(k))&
379             -cos(x(i))*sin(y(j))*cos(z(k)) )*exp(-(factor**2)*time(1)/Re)
380   END DO ; END DO ; END DO
381   DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
        =decomp%xst(1),decomp%xen(1)
382     w(i,j,k)=cos(x(i))*cos(y(j))*sin(z(k))*exp(-(factor**2)*time(1)/Re)
383   END DO ; END DO ; END DO
384
385   CALL decomp_2d_fft_3d(u,uhat,DECOMP_2D_FFT_FORWARD)
386   CALL decomp_2d_fft_3d(v,vhat,DECOMP_2D_FFT_FORWARD)
```

```
387   CALL decomp_2d_fft_3d(w,what,DECOMP_2D_FFT_FORWARD)
388
389   ! derivative of u with respect to x, y, and z
390   DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
          i=decomp%zst(1),decomp%zen(1)
391     temp_c(i,j,k)=uhat(i,j,k)*kx(i)*scalemodes
392   END DO ; END DO ; END DO
393   CALL decomp_2d_fft_3d(temp_c,ux,DECOMP_2D_FFT_BACKWARD)
394   DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
          i=decomp%zst(1),decomp%zen(1)
395     temp_c(i,j,k)=uhat(i,j,k)*ky(j)*scalemodes
396   END DO ; END DO ; END DO
397   CALL decomp_2d_fft_3d(temp_c,uy,DECOMP_2D_FFT_BACKWARD)
398   DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
          i=decomp%zst(1),decomp%zen(1)
399     temp_c(i,j,k)=uhat(i,j,k)*kz(k)*scalemodes
400   END DO ; END DO ; END DO
401   CALL decomp_2d_fft_3d(temp_c,uz,DECOMP_2D_FFT_BACKWARD)
402
403   ! derivative of v with respect to x, y, and z
404   DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
          i=decomp%zst(1),decomp%zen(1)
405     temp_c(i,j,k)=vhat(i,j,k)*kx(i)*scalemodes
406   END DO ; END DO ; END DO
407   CALL decomp_2d_fft_3d(temp_c,vx,DECOMP_2D_FFT_BACKWARD)
408   DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
          i=decomp%zst(1),decomp%zen(1)
409     temp_c(i,j,k)=vhat(i,j,k)*ky(j)*scalemodes
410   END DO ; END DO ; END DO
411   CALL decomp_2d_fft_3d(temp_c,vy,DECOMP_2D_FFT_BACKWARD)
412   DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
          i=decomp%zst(1),decomp%zen(1)
413     temp_c(i,j,k)=vhat(i,j,k)*kz(k)*scalemodes
414   END DO ; END DO ; END DO
415   CALL decomp_2d_fft_3d(temp_c,vz,DECOMP_2D_FFT_BACKWARD)
416
417   ! derivative of w with respect to x, y, and z
418   DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
          i=decomp%zst(1),decomp%zen(1)
419     temp_c(i,j,k)=what(i,j,k)*kx(i)*scalemodes
420   END DO ; END DO ; END DO
421   CALL decomp_2d_fft_3d(temp_c,wx,DECOMP_2D_FFT_BACKWARD)
422   DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
          i=decomp%zst(1),decomp%zen(1)
423     temp_c(i,j,k)=what(i,j,k)*ky(j)*scalemodes
424   END DO ; END DO ; END DO
425   CALL decomp_2d_fft_3d(temp_c,wy,DECOMP_2D_FFT_BACKWARD)
426   DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ; DO
          i=decomp%zst(1),decomp%zen(1)
427     temp_c(i,j,k)=what(i,j,k)*kz(k)*scalemodes
428   END DO ; END DO ; END DO
```

```
429   CALL decomp_2d_fft_3d(temp_c,wz,DECOMP_2D_FFT_BACKWARD)
430   ! save initial data
431   n=0
432   DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
          =decomp%xst(1),decomp%xen(1)
433     realtemp(i,j,k)=REAL(wy(i,j,k)-vz(i,j,k),KIND=8)
434   END DO ; END DO ; END DO
435   name_config='./data/omegax'
436   CALL savedata(Nx,Ny,Nz,n,name_config,realtemp,decomp)
437   !omegay
438   DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
          =decomp%xst(1),decomp%xen(1)
439     realtemp(i,j,k)=REAL(uz(i,j,k)-wx(i,j,k),KIND=8)
440   END DO ; END DO ; END DO
441   name_config='./data/omegay'
442   CALL savedata(Nx,Ny,Nz,n,name_config,realtemp,decomp)
443   !omegaz
444   DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
          =decomp%xst(1),decomp%xen(1)
445     realtemp(i,j,k)=REAL(vx(i,j,k)-uy(i,j,k),KIND=8)
446   END DO ; END DO ; END DO
447   name_config='./data/omegaz'
448   CALL savedata(Nx,Ny,Nz,n,name_config,realtemp,decomp)
449
450         !start timer
451         CALL system_clock(start,count_rate)
452   DO n=1,Nt
453     !fixed point
454     DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO
            i=decomp%xst(1),decomp%xen(1)
455       uold(i,j,k)=u(i,j,k)
456       uxold(i,j,k)=ux(i,j,k)
457       uyold(i,j,k)=uy(i,j,k)
458       uzold(i,j,k)=uz(i,j,k)
459     END DO ; END DO ; END DO
460     DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO
            i=decomp%xst(1),decomp%xen(1)
461       vold(i,j,k)=v(i,j,k)
462       vxold(i,j,k)=vx(i,j,k)
463       vyold(i,j,k)=vy(i,j,k)
464       vzold(i,j,k)=vz(i,j,k)
465     END DO ; END DO ; END DO
466     DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO
            i=decomp%xst(1),decomp%xen(1)
467       wold(i,j,k)=w(i,j,k)
468       wxold(i,j,k)=wx(i,j,k)
469       wyold(i,j,k)=wy(i,j,k)
470       wzold(i,j,k)=wz(i,j,k)
471     END DO ; END DO ; END DO
472     DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ;
          DO i=decomp%zst(1),decomp%zen(1)
```

```
473                         rhsuhatfix(i,j,k) = (dtInv+(0.5*ReInv)*(kx(i)*kx(i
                               )+ky(j)*ky(j)+kz(k)*kz(k)))*uhat(i,j,k)
474     END DO ; END DO ; END DO
475     DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ;
           DO i=decomp%zst(1),decomp%zen(1)
476       rhsvhatfix(i,j,k) = (dtInv+(0.5*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k
             )*kz(k)))*vhat(i,j,k)
477     END DO ; END DO ; END DO
478     DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2) ;
           DO i=decomp%zst(1),decomp%zen(1)
479       rhswhatfix(i,j,k) = (dtInv+(0.5*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k
             )*kz(k)))*what(i,j,k)
480     END DO ; END DO ; END DO
481
482     chg=1
483     DO WHILE (chg .gt. tol)
484       DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
             DO i=decomp%xst(1),decomp%xen(1)
485         temp_r(i,j,k)=0.25d0*((u(i,j,k)+uold(i,j,k))*(ux(i,j,k)+uxold(i,j,
              k))&
486                     +(v(i,j,k)+vold(i,j,k))*(uy(i,j,k)+uyold(i,j,k))&
487                     +(w(i,j,k)+wold(i,j,k))*(uz(i,j,k)+uzold(i,j,k)))
488       END DO ; END DO ; END DO
489       CALL decomp_2d_fft_3d(temp_r,nonlinuhat,DECOMP_2D_FFT_FORWARD)
490       DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
             DO i=decomp%xst(1),decomp%xen(1)
491         temp_r(i,j,k)=0.25d0*((u(i,j,k)+uold(i,j,k))*(vx(i,j,k)+vxold(i,j,
              k))&
492                     +(v(i,j,k)+vold(i,j,k))*(vy(i,j,k)+vyold(i,j,k))&
493                     +(w(i,j,k)+wold(i,j,k))*(vz(i,j,k)+vzold(i,j,k)))
494       END DO ; END DO ; END DO
495       CALL decomp_2d_fft_3d(temp_r,nonlinvhat,DECOMP_2D_FFT_FORWARD)
496       DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
             DO i=decomp%xst(1),decomp%xen(1)
497         temp_r(i,j,k)=0.25d0*((u(i,j,k)+uold(i,j,k))*(wx(i,j,k)+wxold(i,j,
              k))&
498                     +(v(i,j,k)+vold(i,j,k))*(wy(i,j,k)+wyold(i,j,k))&
499                     +(w(i,j,k)+wold(i,j,k))*(wz(i,j,k)+wzold(i,j,k)))
500       END DO ; END DO ; END DO
501       CALL decomp_2d_fft_3d(temp_r,nonlinwhat,DECOMP_2D_FFT_FORWARD)
502       DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
             ; DO i=decomp%zst(1),decomp%zen(1)
503         phat(i,j,k)=-1.0d0*( kx(i)*nonlinuhat(i,j,k)&
504                    +ky(j)*nonlinvhat(i,j,k)&
505                    +kz(k)*nonlinwhat(i,j,k))&
506                    /(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)+0.1d0**13)
507       END DO ; END DO ; END DO
508
509       DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
             ; DO i=decomp%zst(1),decomp%zen(1)
510         uhat(i,j,k)=(rhsuhatfix(i,j,k)-nonlinuhat(i,j,k)-kx(i)*phat(i,j,k)
```

```
          )/&
511          (dtInv-(0.5d0*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))
                !*scalemodes
512       END DO ; END DO ; END DO
513       DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
              ; DO i=decomp%zst(1),decomp%zen(1)
514         vhat(i,j,k)=(rhsvhatfix(i,j,k)-nonlinvhat(i,j,k)-ky(j)*phat(i,j,k)
              )/&
515          (dtInv-(0.5d0*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))
                !*scalemodes
516       END DO ; END DO ; END DO
517       DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
              ; DO i=decomp%zst(1),decomp%zen(1)
518         what(i,j,k)=(rhswhatfix(i,j,k)-nonlinwhat(i,j,k)-kz(k)*phat(i,j,k)
              )/&
519          (dtInv-(0.5d0*ReInv)*(kx(i)*kx(i)+ky(j)*ky(j)+kz(k)*kz(k)))
                !*scalemodes
520       END DO ; END DO ; END DO
521
522       ! derivative of u with respect to x, y, and z
523       DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
              ; DO i=decomp%zst(1),decomp%zen(1)
524         temp_c(i,j,k)=uhat(i,j,k)*kx(i)*scalemodes
525       END DO ; END DO ; END DO
526       CALL decomp_2d_fft_3d(temp_c,ux,DECOMP_2D_FFT_BACKWARD)
527       DO k=decomp%zst(3),decomp%zen(3); DO j=decomp%zst(2),decomp%zen(2) ;
              DO i=decomp%zst(1),decomp%zen(1)
528         temp_c(i,j,k)=uhat(i,j,k)*ky(j)*scalemodes
529       END DO ; END DO ; END DO
530       CALL decomp_2d_fft_3d(temp_c,uy,DECOMP_2D_FFT_BACKWARD)
531       DO k=decomp%zst(3),decomp%zen(3); DO j=decomp%zst(2),decomp%zen(2) ;
              DO i=decomp%zst(1),decomp%zen(1)
532         temp_c(i,j,k)=uhat(i,j,k)*kz(k)*scalemodes
533       END DO ; END DO ; END DO
534       CALL decomp_2d_fft_3d(temp_c,uz,DECOMP_2D_FFT_BACKWARD)
535
536       ! derivative of v with respect to x, y, and z
537       DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
              ; DO i=decomp%zst(1),decomp%zen(1)
538         temp_c(i,j,k)=vhat(i,j,k)*kx(i)*scalemodes
539       END DO ; END DO ; END DO
540       CALL decomp_2d_fft_3d(temp_c,vx,DECOMP_2D_FFT_BACKWARD)
541       DO k=decomp%zst(3),decomp%zen(3); DO j=decomp%zst(2),decomp%zen(2) ;
              DO i=decomp%zst(1),decomp%zen(1)
542         temp_c(i,j,k)=vhat(i,j,k)*ky(j)*scalemodes
543       END DO ; END DO ; END DO
544       CALL decomp_2d_fft_3d(temp_c,vy,DECOMP_2D_FFT_BACKWARD)
545       DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
              ; DO i=decomp%zst(1),decomp%zen(1)
546         temp_c(i,j,k)=vhat(i,j,k)*kz(k)*scalemodes
547       END DO ; END DO ; END DO
```

```
548       CALL decomp_2d_fft_3d(temp_c,vz,DECOMP_2D_FFT_BACKWARD)
549
550       ! derivative of w with respect to x, y, and z
551       DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
              ; DO i=decomp%zst(1),decomp%zen(1)
552         temp_c(i,j,k)=what(i,j,k)*kx(i)*scalemodes
553       END DO ; END DO ; END DO
554       CALL decomp_2d_fft_3d(temp_c,wx,DECOMP_2D_FFT_BACKWARD)
555       DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
              ; DO i=decomp%zst(1),decomp%zen(1)
556         temp_c(i,j,k)=what(i,j,k)*ky(j)*scalemodes
557       END DO ; END DO ; END DO
558       CALL decomp_2d_fft_3d(temp_c,wy,DECOMP_2D_FFT_BACKWARD)
559       DO k=decomp%zst(3),decomp%zen(3) ; DO j=decomp%zst(2),decomp%zen(2)
              ; DO i=decomp%zst(1),decomp%zen(1)
560         temp_c(i,j,k)=what(i,j,k)*kz(k)*scalemodes
561       END DO ; END DO ; END DO
562       CALL decomp_2d_fft_3d(temp_c,wz,DECOMP_2D_FFT_BACKWARD)
563
564       DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
              DO i=decomp%xst(1),decomp%xen(1)
565         utemp(i,j,k)=u(i,j,k)
566       END DO ; END DO ; END DO
567       DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
              DO i=decomp%xst(1),decomp%xen(1)
568         vtemp(i,j,k)=v(i,j,k)
569       END DO ; END DO ; END DO
570       DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
              DO i=decomp%xst(1),decomp%xen(1)
571         wtemp(i,j,k)=w(i,j,k)
572       END DO ; END DO ; END DO
573
574       CALL decomp_2d_fft_3d(uhat,u,DECOMP_2D_FFT_BACKWARD)
575       CALL decomp_2d_fft_3d(vhat,v,DECOMP_2D_FFT_BACKWARD)
576       CALL decomp_2d_fft_3d(what,w,DECOMP_2D_FFT_BACKWARD)
577
578       DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
              DO i=decomp%xst(1),decomp%xen(1)
579         u(i,j,k)=u(i,j,k)*scalemodes
580       END DO ; END DO ; END DO
581       DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
              DO i=decomp%xst(1),decomp%xen(1)
582         v(i,j,k)=v(i,j,k)*scalemodes
583       END DO ; END DO ; END DO
584       DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2);
              DO i=decomp%xst(1),decomp%xen(1)
585         w(i,j,k)=w(i,j,k)*scalemodes
586       END DO ; END DO ; END DO
587
588       mychg(1) =maxval(abs(utemp-u))
589       mychg(2) =maxval(abs(vtemp-v))
```

```
590       mychg(3) =maxval(abs(wtemp-w))
591       CALL MPI_ALLREDUCE(mychg,allchg,3,MPI_DOUBLE_PRECISION,MPI_MAX,
              MPI_COMM_WORLD,ierr)
592       chg=allchg(1)+allchg(2)+allchg(3)
593       IF (myid.eq.0) THEN
594         PRINT *,'chg:',chg
595       END IF
596     END DO
597     time(n+1)=n*dt
598
599                 !goto 5100
600     IF (myid.eq.0) THEN
601       PRINT *,'time',n*dt
602     END IF
603
604                 !save omegax, omegay, and omegaz
605     !omegax
606     DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO
            i=decomp%xst(1),decomp%xen(1)
607       realtemp(i,j,k)=REAL(wy(i,j,k)-vz(i,j,k),KIND=8)
608     END DO ; END DO ; END DO
609     name_config='./data/omegax'
610     CALL savedata(Nx,Ny,Nz,n,name_config,realtemp,decomp)
611     !omegay
612     DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO
            i=decomp%xst(1),decomp%xen(1)
613       realtemp(i,j,k)=REAL(uz(i,j,k)-wx(i,j,k),KIND=8)
614     END DO ; END DO ; END DO
615     name_config='./data/omegay'
616     CALL savedata(Nx,Ny,Nz,n,name_config,realtemp,decomp)
617     !omegaz
618     DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO
            i=decomp%xst(1),decomp%xen(1)
619       realtemp(i,j,k)=REAL(vx(i,j,k)-uy(i,j,k),KIND=8)
620     END DO ; END DO ; END DO
621     name_config='./data/omegaz'
622     CALL savedata(Nx,Ny,Nz,n,name_config,realtemp,decomp)
623                 !5100 continue
624   END DO
625
626         CALL system_clock(finish,count_rate)
627
628         IF (myid.eq.0) then
629            PRINT *, 'Program took', REAL(finish-start)/REAL(count_rate), '
                for main timestepping loop'
630         END IF
631
632   IF (myid.eq.0) THEN
633     name_config = './data/tdata.dat'
634     OPEN(unit=11,FILE=name_config,status="UNKNOWN")
635     REWIND(11)
```

```
636     DO n=1,1+Nt
637       WRITE(11,*) time(n)
638     END DO
639     CLOSE(11)
640
641     name_config = './data/xcoord.dat'
642     OPEN(unit=11,FILE=name_config,status="UNKNOWN")
643     REWIND(11)
644     DO i=1,Nx
645       WRITE(11,*) x(i)
646     END DO
647     CLOSE(11)
648
649     name_config = './data/ycoord.dat'
650     OPEN(unit=11,FILE=name_config,status="UNKNOWN")
651     REWIND(11)
652     DO j=1,Ny
653       WRITE(11,*) y(j)
654     END DO
655     CLOSE(11)
656
657     name_config = './data/zcoord.dat'
658     OPEN(unit=11,FILE=name_config,status="UNKNOWN")
659     REWIND(11)
660     DO k=1,Nz
661       WRITE(11,*) z(k)
662     END DO
663     CLOSE(11)
664     PRINT *,'Saved data'
665   END IF
666
667     ! Calculate error in final numerical solution
668   DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
        =decomp%xst(1),decomp%xen(1)
669     utemp(i,j,k)=u(i,j,k) -&
670             (-0.5*( factor*cos(x(i))*sin(y(j))*sin(z(k))&
671             +sin(x(i))*cos(y(j))*cos(z(k)) )*exp(-(factor**2)*time(Nt+1)/
                Re))
672   END DO; END DO; END DO
673   DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
        =decomp%xst(1),decomp%xen(1)
674     vtemp(i,j,k)=v(i,j,k) -&
675           (0.5*(  factor*sin(x(i))*cos(y(j))*sin(z(k))&
676             -cos(x(i))*sin(y(j))*cos(z(k)) )*exp(-(factor**2)*time(Nt+1)/
                Re))
677   END DO ; END DO ; END DO
678   DO k=decomp%xst(3),decomp%xen(3); DO j=decomp%xst(2),decomp%xen(2); DO i
        =decomp%xst(1),decomp%xen(1)
679     wtemp(i,j,k)=w(i,j,k)-&
680           (cos(x(i))*cos(y(j))*sin(z(k))*exp(-(factor**2)*time(Nt+1)/Re))
681   END DO ; END DO ; END DO
```

```
682   mychg(1) = maxval(abs(utemp))
683   mychg(2) = maxval(abs(vtemp))
684   mychg(3) = maxval(abs(wtemp))
685   CALL MPI_ALLREDUCE(mychg,allchg,3,MPI_DOUBLE_PRECISION,MPI_MAX,
          MPI_COMM_WORLD,ierr)
686   chg=allchg(1)+allchg(2)+allchg(3)
687   IF (myid.eq.0) THEN
688     PRINT*,'The error at the final timestep is',chg
689   END IF
690
691         ! clean up
692     CALL decomp_2d_fft_finalize
693     CALL decomp_2d_finalize
694
695   DEALLOCATE(x,y,z,time,mychg,allchg,u,v,w,ux,uy,uz,vx,vy,vz,wx,wy,wz,uold
          ,uxold,uyold,uzold,&
696           vold,vxold,vyold,vzold,wold,wxold,wyold,wzold,utemp,vtemp,wtemp
                ,&
697           temp_r,kx,ky,kz,uhat,vhat,what,rhsuhatfix,rhsvhatfix,&
698           rhswhatfix,phat,nonlinuhat,nonlinvhat,nonlinwhat,temp_c,&
699           realtemp,stat=AllocateStatus)
700   IF (AllocateStatus .ne. 0) STOP
701   IF (myid.eq.0) THEN
702     PRINT *,'Program execution complete'
703   END IF
704   CALL MPI_FINALIZE(ierr)
705
706   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.

```
1   SUBROUTINE savedata(Nx,Ny,Nz,plotnum,name_config,field,decomp)
2   !-----------------------------------------------------------------------
3   !
4   !
5   ! PURPOSE
6   !
7   ! This subroutine saves a three dimensional real array in binary
8   ! format
9   !
10  ! INPUT
11  !
12  ! .. Scalars ..
13  !  Nx       = number of modes in x - power of 2 for FFT
14  !  Ny       = number of modes in y - power of 2 for FFT
15  !  Nz       = number of modes in z - power of 2 for FFT
16  !  plotnum      = number of plot to be made
17  ! .. Arrays ..
18  !  field      = real data to be saved
```

```
19  !  name_config    = root of filename to save to
20  !
21  ! .. Output ..
22  ! plotnum     = number of plot to be saved
23  ! .. Special Structures ..
24  !  decomp     = contains information on domain decomposition
25  !         see http://www.2decomp.org/ for more information
26  ! LOCAL VARIABLES
27  !
28  ! .. Scalars ..
29  !  i         = loop counter in x direction
30  !  j         = loop counter in y direction
31  !  k         = loop counter in z direction
32  !  count       = counter
33  !  iol         = size of file
34  ! .. Arrays ..
35  !   number_file    = array to hold the number of the plot
36  !
37  ! REFERENCES
38  !
39  ! ACKNOWLEDGEMENTS
40  !
41  ! ACCURACY
42  !
43  ! ERROR INDICATORS AND WARNINGS
44  !
45  ! FURTHER COMMENTS
46  !-----------------------------------------------------------------------
47  ! External routines required
48  !
49  ! External libraries required
50  ! 2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
51  !     (http://www.2decomp.org/index.html)
52  ! MPI library
53  USE decomp_2d
54  USE decomp_2d_fft
55  USE decomp_2d_io
56  IMPLICIT NONE
57  INCLUDE 'mpif.h'
58  ! Declare variables
59  INTEGER(KIND=4), INTENT(IN)           :: Nx,Ny,Nz
60  INTEGER(KIND=4), INTENT(IN)           :: plotnum
61  TYPE(DECOMP_INFO), INTENT(IN)         ::  decomp
62  REAL(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
63                 decomp%xst(2):decomp%xen(2),&
64                 decomp%xst(3):decomp%xen(3)), &
65                       INTENT(IN) :: field
66  CHARACTER*100, INTENT(IN)           :: name_config
67  INTEGER(kind=4)                  :: i,j,k,iol,count,ind
68  CHARACTER*100                 :: number_file
69
```

```
  ! 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 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 the 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

$$\int \frac{1}{2}\left(\frac{\partial u}{\partial t}\right)^2 + \frac{u^2}{2} + \frac{1}{2}\left|\nabla u\right|^2 \mp \frac{\left|u\right|^4}{4} \mathrm{d}\boldsymbol{x}. \tag{14.2}$$

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

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{(\delta t)^2} - \Delta\frac{u^{n+1} + 2u^n + u^{n-1}}{4} + \frac{u^{n+1} + 2u^n + u^{n-1}}{4} = \pm\left|u^n\right|^2 u^n \tag{14.3}$$

and second, a fully implicit time stepping scheme with fixed point iteration

$$\begin{aligned}&\frac{u^{n+1,k+1} - 2u^n + u^{n-1}}{(\delta t)^2} - \Delta\frac{u^{n+1,k+1} + 2u^n + u^{n-1}}{4} + \frac{u^{n+1,k+1} + 2u^n + u^{n-1}}{4} \\ &= \pm\frac{\left|u^{n+1,k}\right|^4 - \left|u^{n-1}\right|^4}{u^{n+1,k} - u^{n-1}}\end{aligned} \tag{14.4}$$

which conserves a discrete energy exactly

$$\int \frac{1}{2}\left(\frac{u^{n+1} - u^n}{\delta t}\right)^2 + \frac{1}{2}\left(\frac{u^{n+1} + u^n}{2}\right)^2 + \frac{1}{2}\left|\nabla\frac{u^{n+1} + u^n}{2}\right|^2 \mp \frac{\left|u^{n+1}\right|^4 + \left|u^n\right|^4}{8}. \tag{14.5}$$

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
```

```
19
20 Es = 1.0; % focusing (+1) or defocusing (-1) parameter
21
22
23 % initialise variables
24 x = (2*pi/Nx)*(-Nx/2:Nx/2 -1)'*Lx;          % x coordinate
25 kx = 1i*[0:Nx/2-1 0 -Nx/2+1:-1]'/Lx;        % wave vector
26 y = (2*pi/Ny)*(-Ny/2:Ny/2 -1)'*Ly;          % y coordinate
27 ky = 1i*[0:Ny/2-1 0 -Ny/2+1:-1]'/Ly;        % wave vector
28 [xx,yy]=meshgrid(x,y);
29 [kxm,kym]=meshgrid(kx,ky);
30
31 % initial conditions
32 u = (0.5*exp(-(xx.^2+yy.^2))).*sin(10*xx+12*yy);
33 uold=u;
34 v=fft2(u);
35 vold=fft2(uold);
36 figure(1); clf; mesh(xx,yy,u); drawnow;
37 t=0; tdata(1)=t;
38
39 % initial energy
40 vx=0.5*kxm.*(v+vold);
41 vy=0.5*kym.*(v+vold);
42 ux=ifft2(vx);
43 uy=ifft2(vy);
44 ux=ifft2(vx);
45 uy=ifft2(vy);
46 Kineticenergy=0.5*abs( (u-uold)/dt).^2;
47 Strainenergy=0.5*abs(ux).^2 +0.5*abs(uy).^2;
48 Potentialenergy=0.5*abs(0.5*(u+uold)).^2 ...
49                     -Es*0.25*((u+uold)*0.5).^4;
50 Kineticenergy=fft2(Kineticenergy);
51 Potentialenergy=fft2(Potentialenergy);
52 Strainenergy=fft2(Strainenergy);
53 EnKin(1)=Kineticenergy(1,1);
54 EnPot(1)=Potentialenergy(1,1);
55 EnStr(1)=Strainenergy(1,1);
56 En(1)=EnStr(1)+EnKin(1)+EnPot(1);
57 En0=En(1)
58 plotnum=1;
59
60 % solve pde and plot results
61
62 for n =1:Nt+1
63     nonlin=(u.^4 -uold.^4)./(u-uold+0.1^14);
64     nonlinhat=fft2(nonlin);
65     chg=1;
66     unew=u;
67     while (chg>tol)
68         utemp=unew;
69         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);
```

```
103         En(plotnum+1)=EnStr(plotnum+1)+EnKin(plotnum+1)+EnPot(plotnum+1);
104         Enchange(plotnum)=log(abs(1-En(1+plotnum)/En0));
105         plotnum=plotnum+1;
106     end
107     % update old terms
108     vold=v;
109     v=vnew;
110     uold=u;
111     u=unew;
112 end
113 figure(4); clf;
114 % coordinate slice to show plots on
115 sx=[0]; sy=[0]; sz=[0];
116 slice(xx,yy,zz,u,sx,sy,sz); colormap jet;
117 title(num2str(t)); colorbar('location','EastOutside'); drawnow;
118
119 xlabel('x'); ylabel('y'); zlabel('z');
120 axis equal; axis square;  view(3); drawnow;
121
122 figure(5); clf; plot(tdata,En,'r-',tdata,EnKin,'b:',tdata,EnPot,'g-.',
    tdata,EnStr,'y--');
123 xlabel time; ylabel Energy; legend('Total','Kinetic','Potential','Strain')
    ;
124 figure(6); clf; plot(tdata(2:end),Enchange,'r-'); xlabel time; ylabel('
    Energy change');
125
126 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
```

```
51  !  nonlin       = nonlinear term, u^3
52  !  nonlinhat    = Fourier transform of nonlinear term, u^3
53  ! .. Vectors ..
54  !  kx       = fourier frequencies in x direction
55  !  ky       = fourier frequencies in y direction
56  !  x        = x locations
57  !  y        = y locations
58  !  time       = times at which save data
59  !  en       = total energy
60  !  enstr      = strain energy
61  !  enpot      = potential energy
62  !  enkin      = kinetic energy
63  !  name_config    = array to store filename for data to be saved
64  !  fftfxy     = array to setup 2D Fourier transform
65  !  fftbxy     = array to setup 2D Fourier transform
66  !
67  ! REFERENCES
68  !
69  ! ACKNOWLEDGEMENTS
70  !
71  ! ACCURACY
72  !
73  ! ERROR INDICATORS AND WARNINGS
74  !
75  ! FURTHER COMMENTS
76  ! Check that the initial iterate is consistent with the
77  ! boundary conditions for the domain specified
78  !-----------------------------------------------------------------------
79  ! External routines required
80  ! getgrid.f90 -- Get initial grid of points
81  !   initialdata.f90 -- Get initial data
82  ! enercalc.f90 -- Subroutine to calculate the energy
83  ! savedata.f90 -- Save initial data
84  ! storeold.f90 -- Store old data
85  ! External libraries required
86  !   FFTW3  -- Fast Fourier Transform in the West Library
87  !     (http://www.fftw.org/)
88  !   OpenMP library
89
90  PROGRAM Kg
91  USE omp_lib
92  ! Declare variables
93  IMPLICIT NONE
94  INTEGER(kind=4), PARAMETER  :: Nx=128
95  INTEGER(kind=4), PARAMETER  :: Ny=128
96  INTEGER(kind=4), PARAMETER  :: Nt=20
97  INTEGER(kind=4), PARAMETER  :: plotgap=5
98  REAL(kind=8), PARAMETER   :: &
99    pi=3.14159265358979323846264338327950288419716939937510d0
100 REAL(kind=8), PARAMETER   ::  Lx=3.0d0
101 REAL(kind=8), PARAMETER   ::  Ly=3.0d0
```

```
102   REAL(kind=8), PARAMETER    ::  Es=1.0d0
103   REAL(kind=8)          ::  dt=0.10d0/REAL(Nt,kind(0d0))
104   COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  kx,ky
105   REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  x,y
106   COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  u,nonlin
107   COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  v,nonlinhat
108   COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  uold
109   COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  vold
110   COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  unew
111   COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE::  vnew
112   REAL(kind=8), DIMENSION(:,:), ALLOCATABLE :: savearray
113   REAL(kind=8), DIMENSION(:), ALLOCATABLE ::  time,enkin,enstr,enpot,en
114   INTEGER(kind=4) ::  ierr,i,j,n,allocatestatus,numthreads
115   INTEGER(kind=4) ::  start, finish, count_rate, plotnum
116   INTEGER(kind=4), PARAMETER  ::  FFTW_IN_PLACE = 8, FFTW_MEASURE = 0, &
117     FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32, FFTW_ESTIMATE = 64
118   INTEGER(kind=4),PARAMETER ::  FFTW_FORWARD = -1, FFTW_BACKWARD=1
119   INTEGER(kind=8) ::  planfxy,planbxy
120   CHARACTER*100 ::  name_config
121   ! Start short parallel region to count threads
122   numthreads=omp_get_max_threads()
123   PRINT *,'There are ',numthreads,' threads.'
124   ALLOCATE(kx(1:Nx),ky(1:Ny),x(1:Nx),y(1:Ny),u(1:Nx,1:Ny),&
125       v(1:Nx,1:Ny),nonlin(1:Nx,1:Ny),nonlinhat(1:Nx,1:Ny),&
126       uold(1:Nx,1:Ny),vold(1:Nx,1:Ny),&
127       unew(1:Nx,1:Ny),vnew(1:Nx,1:Ny),savearray(1:Nx,1:Ny),&
128       time(1:1+Nt/plotgap),enkin(1:1+Nt/plotgap),&
129       enstr(1:1+Nt/plotgap),enpot(1:1+Nt/plotgap),&
130       en(1:1+Nt/plotgap),stat=allocatestatus)
131   IF (allocatestatus .ne. 0) stop
132   PRINT *,'allocated arrays'
133
134   ! set up multithreaded ffts
135   CALL dfftw_init_threads_(ierr)
136   PRINT *,'Initiated threaded FFTW'
137   CALL dfftw_plan_with_nthreads_(numthreads)
138   PRINT *,'Inidicated number of threads to be used in planning'
139   CALL dfftw_plan_dft_2d_(planfxy,Nx,Ny,u,v,&
140               FFTW_FORWARD,FFTW_ESTIMATE)
141   CALL dfftw_plan_dft_2d_(planbxy,Nx,Ny,v,u,&
142               FFTW_BACKWARD,FFTW_ESTIMATE)
143   PRINT *,'Setup FFTs'
144   ! setup fourier frequencies
145   CALL getgrid(Nx,Ny,Lx,Ly,pi,name_config,x,y,kx,ky)
146   PRINT *,'Setup grid and fourier frequencies'
147   CALL initialdata(Nx,Ny,x,y,u,uold)
148   plotnum=1
149   name_config = 'data/u'
150   savearray=REAL(u)
151   CALL savedata(Nx,Ny,plotnum,name_config,savearray)
152
```

```
153   CALL dfftw_execute_dft_(planfxy,u,v)
154   CALL dfftw_execute_dft_(planfxy,uold,vold)
155
156   CALL enercalc(Nx,Ny,planfxy,planbxy,dt,Es,&
157         enkin(plotnum),enstr(plotnum),&
158         enpot(plotnum),en(plotnum),&
159         kx,ky,nonlin,nonlinhat,&
160         v,vold,u,uold)
161
162   PRINT *,'Got initial data, starting timestepping'
163   time(plotnum)=0.0d0
164     CALL system_clock(start,count_rate)
165   DO n=1,Nt
166     !$OMP PARALLEL DO PRIVATE(i,j) SCHEDULE(static)
167     DO j=1,Ny
168       DO i=1,Nx
169         nonlin(i,j)=(abs(u(i,j))*2)*u(i,j)
170       END DO
171     END DO
172     !$OMP END PARALLEL DO
173     CALL dfftw_execute_dft_(planfxy,nonlin,nonlinhat)
174     !$OMP PARALLEL DO PRIVATE(i,j) SCHEDULE(static)
175     DO j=1,Ny
176       DO i=1,Nx
177         vnew(i,j)=( 0.25*(kx(i)*kx(i) + ky(j)*ky(j)-1.0d0)&
178           *(2.0d0*v(i,j)+vold(i,j))&
179           +(2.0d0*v(i,j)-vold(i,j))/(dt*dt)&
180           +Es*nonlinhat(i,j) )&
181           /(1/(dt*dt)-0.25*(kx(i)*kx(i) + ky(j)*ky(j)-1.0d0))
182       END DO
183     END DO
184     !$OMP END PARALLEL DO
185     CALL dfftw_execute_dft_(planbxy,vnew,unew)
186     ! normalize result
187     !$OMP PARALLEL DO PRIVATE(i,j) SCHEDULE(static)
188     DO j=1,Ny
189       DO i=1,Nx
190         unew(i,j)=unew(i,j)/REAL(Nx*Ny,kind(0d0))
191       END DO
192     END DO
193     !$OMP END PARALLEL DO
194     IF (mod(n,plotgap)==0) then
195       plotnum=plotnum+1
196       time(plotnum)=n*dt
197       PRINT *,'time',n*dt
198       CALL enercalc(Nx,Ny,planfxy,planbxy,dt,Es,&
199         enkin(plotnum),enstr(plotnum),&
200         enpot(plotnum),en(plotnum),&
201         kx,ky,&
202         nonlin,nonlinhat,&
203         vnew,v,unew,u)
```

```
204       savearray=REAL(unew,kind(0d0))
205       CALL savedata(Nx,Ny,plotnum,name_config,savearray)
206     END IF
207       ! .. Update old values ..
208     CALL storeold(Nx,Ny,&
209         unew,u,uold,&
210         vnew,v,vold)
211   END DO
212   PRINT *,'Finished time stepping'
213   CALL system_clock(finish,count_rate)
214   PRINT*,'Program took ',&
215     REAL(finish-start,kind(0d0))/REAL(count_rate,kind(0d0)),&
216     'for Time stepping'
217   CALL saveresults(Nt,plotgap,time(1:1+n/plotgap),en(1:1+n/plotgap),&
218       enstr(1:1+n/plotgap),enkin(1:1+n/plotgap),enpot(1:1+n/plotgap))
219
220   ! Save times at which output was made in text format
221   PRINT *,'Saved data'
222
223   CALL dfftw_destroy_plan_(planbxy)
224   CALL dfftw_destroy_plan_(planfxy)
225   CALL dfftw_cleanup_threads_()
226   DEALLOCATE(kx,ky,x,y,u,v,nonlin,nonlinhat,savearray,&
227     uold,vold,unew,vnew,time,enkin,enstr,enpot,en,&
228     stat=allocatestatus)
229   IF (allocatestatus .ne. 0) STOP
230   PRINT *,'Deallocated arrays'
231   PRINT *,'Program execution complete'
232   END PROGRAM Kg
```

Listing 14.6: A Fortran subroutine to get the grid to solve the 2D Klein-Gordon equation on.

```
1   SUBROUTINE getgrid(Nx,Ny,Lx,Ly,pi,name_config,x,y,kx,ky)
2   !-----------------------------------------------------------------------
3   !
4   !
5   ! PURPOSE
6   !
7   ! This subroutine gets grid points and fourier frequencies for a
8   ! pseudospectral simulation of the 2D nonlinear Klein-Gordon equation
9   !
10  ! u_{tt}-u_{xx}+u_{yy}+u=Es*u^3
11  !
12  ! The boundary conditions are u(x=0,y)=u(2*Lx*\pi,y),
13  ! u(x,y=0)=u(x,y=2*Ly*\pi)
14  !
15  ! INPUT
16  !
17  ! .. Scalars ..
```

```
18  !  Nx        = number of modes in x - power of 2 for FFT
19  !  Ny        = number of modes in y - power of 2 for FFT
20  !  pi        = 3.142....
21  !  Lx        = width of box in x direction
22  !  Ly        = width of box in y direction
23  ! OUTPUT
24  !
25  ! .. Vectors ..
26  !  kx        = fourier frequencies in x direction
27  !  ky        = fourier frequencies in y direction
28  !  x         = x locations
29  !  y         = y locations
30  !
31  ! LOCAL VARIABLES
32  !
33  ! .. Scalars ..
34  !  i         = loop counter in x direction
35  !  j         = loop counter in y direction
36  !
37  ! REFERENCES
38  !
39  ! ACKNOWLEDGEMENTS
40  !
41  ! ACCURACY
42  !
43  ! ERROR INDICATORS AND WARNINGS
44  !
45  ! FURTHER COMMENTS
46  ! Check that the initial iterate is consistent with the
47  ! boundary conditions for the domain specified
48  !-----------------------------------------------------------------------
49  ! External routines required
50  !
51  ! External libraries required
52  ! OpenMP library
53  IMPLICIT NONE
54  USE omp_lib
55  ! Declare variables
56  INTEGER(KIND=4), INTENT(IN)              :: Nx,Ny
57  REAL(kind=8), INTENT(IN)               :: Lx,Ly,pi
58  REAL(KIND=8), DIMENSION(1:NX), INTENT(OUT)       :: x
59  REAL(KIND=8), DIMENSION(1:NY), INTENT(OUT)       :: y
60  COMPLEX(KIND=8), DIMENSION(1:NX), INTENT(OUT)    :: kx
61  COMPLEX(KIND=8), DIMENSION(1:NY), INTENT(OUT)    :: ky
62  CHARACTER*100, INTENT(OUT)               :: name_config
63  INTEGER(kind=4)                    :: i,j
64
65  !$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
66  DO i=1,1+Nx/2
67    kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
68  END DO
```

```
69  !$OMP END PARALLEL DO
70  kx(1+Nx/2)=0.0d0
71  !$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
72  DO i = 1,Nx/2 -1
73    kx(i+1+Nx/2)=-kx(1-i+Nx/2)
74  END DO
75  !$OMP END PARALLEL DO
76
77  !$OMP PARALLEL DO PRIVATE(i) SCHEDULE(static)
78  DO i=1,Nx
79    x(i)=(-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)))*pi*Lx
80  END DO
81  !$OMP END PARALLEL DO
82  !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
83  DO j=1,1+Ny/2
84    ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
85  END DO
86  !$OMP END PARALLEL DO
87  ky(1+Ny/2)=0.0d0
88  !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
89  DO j = 1,Ny/2 -1
90    ky(j+1+Ny/2)=-ky(1-j+Ny/2)
91  END DO
92  !$OMP END PARALLEL DO
93  !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
94  DO j=1,Ny
95    y(j)=(-1.0d0 + 2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)))*pi*Ly
96  END DO
97  !$OMP END PARALLEL DO
98  ! Save x grid points in text format
99  name_config = 'xcoord.dat'
100 OPEN(unit=11,FILE=name_config,status="UNKNOWN")
101 REWIND(11)
102 DO i=1,Nx
103   WRITE(11,*) x(i)
104 END DO
105 CLOSE(11)
106 ! Save y grid points in text format
107 name_config = 'ycoord.dat'
108 OPEN(unit=11,FILE=name_config,status="UNKNOWN")
109 REWIND(11)
110 DO j=1,Ny
111   WRITE(11,*) y(j)
112 END DO
113 CLOSE(11)
114
115
116 END SUBROUTINE getgrid
```

Listing 14.7: A Fortran subroutine to get the initial data to solve the 2D Klein-Gordon equation for.

```
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
```

```
50  !
51  ! External libraries required
52  ! OpenMP library
53  USE omp_lib
54  IMPLICIT NONE
55  ! Declare variables
56  INTEGER(KIND=4), INTENT(IN)              :: Nx,Ny
57  REAL(KIND=8), DIMENSION(1:NX), INTENT(IN)       :: x
58  REAL(KIND=8), DIMENSION(1:NY), INTENT(IN)       :: y
59  COMPLEX(KIND=8), DIMENSION(1:NX,1:NY), INTENT(OUT)  :: u,uold
60  INTEGER(kind=4)                       :: i,j
61  !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
62  DO j=1,Ny
63    u(1:Nx,j)=0.5d0*exp(-1.0d0*(x(1:Nx)**2 +y(j)**2))*&
64          sin(10.0d0*x(1:Nx)+12.0d0*y(j))
65  END DO
66  !$OMP END PARALLEL DO
67  !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
68  DO j=1,Ny
69    uold(1:Nx,j)=0.5d0*exp(-1.0d0*(x(1:Nx)**2 +y(j)**2))*&
70          sin(10.0d0*x(1:Nx)+12.0d0*y(j))
71  END DO
72  !$OMP END PARALLEL DO
73
74  END SUBROUTINE initialdata
```

Listing 14.8: A Fortran program to save a field from the solution of the 2D Klein-Gordon equation.

```
1   SUBROUTINE savedata(Nx,Ny,plotnum,name_config,field)
2   !-----------------------------------------------------------------------
3   !
4   !
5   ! PURPOSE
6   !
7   ! This subroutine saves a two dimensional real array in binary
8   ! format
9   !
10  ! INPUT
11  !
12  ! .. Scalars ..
13  !  Nx       = number of modes in x - power of 2 for FFT
14  !  Ny       = number of modes in y - power of 2 for FFT
15  !  plotnum      = number of plot to be made
16  ! .. Arrays ..
17  !  field      = real data to be saved
18  !  name_config    = root of filename to save to
19  !
20  ! .. Output ..
21  ! plotnum     = number of plot to be saved
```

```
!
! 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)

```

```
73  END SUBROUTINE savedata
```

Listing 14.9: A Fortran subroutine to update arrays when solving the 2D Klein-Gordon equation.

```
1   SUBROUTINE storeold(Nx,Ny,unew,u,uold,vnew,v,vold)
2   !-----------------------------------------------------------------------
3   !
4   !
5   ! PURPOSE
6   !
7   ! This subroutine copies arrays for a
8   ! pseudospectral simulation of the 2D nonlinear Klein-Gordon equation
9   !
10  ! u_{tt}-u_{xx}+u_{yy}+u=Es*u^3
11  !
12  ! INPUT
13  !
14  ! .. Parameters ..
15  !  Nx        = number of modes in x - power of 2 for FFT
16  !  Ny        = number of modes in y - power of 2 for FFT
17  !  .. Arrays ..
18  !  unew        = approximate solution
19  !  vnew        = Fourier transform of approximate solution
20  !  u         = approximate solution
21  !  v         = Fourier transform of approximate solution
22  !  uold        = approximate solution
23  !  vold        = Fourier transform of approximate solution
24  !
25  ! OUTPUT
26  !
27  !  u         = approximate solution
28  !  v         = Fourier transform of approximate solution
29  !  uold        = approximate solution
30  !  vold        = Fourier transform of approximate solution
31  !
32  ! LOCAL VARIABLES
33  !
34  ! .. Scalars ..
35  !  i         = loop counter in x direction
36  !  j         = loop counter in y direction
37  !
38  ! REFERENCES
39  !
40  ! ACKNOWLEDGEMENTS
41  !
42  ! ACCURACY
43  !
44  ! ERROR INDICATORS AND WARNINGS
45  !
```

```
46  ! FURTHER COMMENTS
47  !---------------------------------------------------------------------
48  ! External routines required
49  !
50  ! External libraries required
51  ! OpenMP library
52  USE omp_lib
53  IMPLICIT NONE
54  ! Declare variables
55  INTEGER(KIND=4), INTENT(IN)               :: Nx,Ny
56  COMPLEX(KIND=8), DIMENSION(1:NX,1:NY), INTENT(OUT)  :: vold,uold
57  COMPLEX(KIND=8), DIMENSION(1:NX,1:NY), INTENT(INOUT):: u,v
58  COMPLEX(KIND=8), DIMENSION(1:NX,1:NY), INTENT(IN) :: unew,vnew
59  INTEGER(kind=4)                      :: i,j
60
61  !$OMP PARALLEL PRIVATE(i,j)
62
63  !$OMP DO SCHEDULE(static)
64  DO j=1,Ny
65    DO i=1,Nx
66      vold(i,j)=v(i,j)
67    END DO
68  END DO
69  !$OMP END DO NOWAIT
70
71  !$OMP DO SCHEDULE(static)
72  DO j=1,Ny
73    DO i=1,Nx
74      uold(i,j)=u(i,j)
75    END DO
76  END DO
77  !$OMP END DO NOWAIT
78
79  !$OMP DO SCHEDULE(static)
80  DO j=1,Ny
81    DO i=1,Nx
82      u(i,j)=unew(i,j)
83    END DO
84  END DO
85  !$OMP END DO NOWAIT
86
87  !$OMP DO SCHEDULE(static)
88  DO j=1,Ny
89    DO i=1,Nx
90      v(i,j)=vnew(i,j)
91    END DO
92  END DO
93  !$OMP END DO NOWAIT
94
95  !$OMP END PARALLEL
96
```

```
97   END SUBROUTINE storeold
```

Listing 14.10: A Fortran subroutine to calculate the energy when solving the 2D Klein-Gordon equation.

```
1    SUBROUTINE enercalc(Nx,Ny,planfxy,planbxy,dt,Es,enkin,enstr,&
2          enpot,en,kx,ky,temp1,temp2,v,vold,u,uold)
3    !-----------------------------------------------------------------------
4    !
5    !
6    ! PURPOSE
7    !
8    ! This subroutine program calculates the energy for the nonlinear
9    ! Klein-Gordon equation in 2 dimensions
10   ! u_{tt}-u_{xx}+u_{yy}+u=Es*|u|^2u
11   !
12   ! The energy density is given by
13   ! 0.5u_t^2+0.5u_x^2+0.5u_y^2+0.5u^2+Es*0.25u^4
14   !
15   ! INPUT
16   !
17   ! .. Scalars ..
18   !  Nx        = number of modes in x - power of 2 for FFT
19   !  Ny        = number of modes in y - power of 2 for FFT
20   !  planfxy      = Forward 2d fft plan
21   !  planbxy      = Backward 2d fft plan
22   !  dt        = timestep
23   !  Es        = +1 for focusing, -1 for defocusing
24   ! .. Arrays ..
25   !  u         = approximate solution
26   !  v         = Fourier transform of approximate solution
27   !  uold        = approximate solution
28   !  vold        = Fourier transform of approximate solution
29   !  temp1       = array to hold temporary values
30   !  temp2       = array to hold temporary values
31   ! .. Vectors ..
32   !  kx        = fourier frequencies in x direction
33   !  ky        = fourier frequencies in y direction
34   !
35   ! OUTPUT
36   !
37   ! .. Scalars ..
38   !  enkin       = Kinetic energy
39   !  enstr       = Strain energy
40   !  enpot       = Potential energy
41   !  en        = Total energy
42   !
43   ! LOCAL VARIABLES
44   !
45   ! .. 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
```

```
97      temp1(1:Nx,j)=0.5d0*ky(j)*(vold(1:Nx,j)+v(1:Nx,j))
98    END DO
99    !$OMP END PARALLEL DO
100   CALL dfftw_execute_dft_(planbxy,temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny))
101   !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
102   DO j=1,Ny
103     temp1(1:Nx,j)=abs(temp2(1:Nx,j)/REAL(Nx*Ny,kind(0d0)))**2
104   END DO
105   !$OMP END PARALLEL DO
106   CALL dfftw_execute_dft_(planfxy,temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny))
107   enstr=enstr+0.5d0*REAL(abs(temp2(1,1)),kind(0d0))/REAL(Nx*Ny,kind(0d0))
108
109   ! .. Kinetic Energy ..
110   !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
111   DO j=1,Ny
112     temp1(1:Nx,j)=( abs(u(1:Nx,j)-uold(1:Nx,j))/dt )**2
113   END DO
114   !$OMP END PARALLEL DO
115   CALL dfftw_execute_dft_(planfxy,temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny))
116   enkin=0.5d0*REAL(abs(temp2(1,1)),kind(0d0))/REAL(Nx*Ny,kind(0d0))
117
118   ! .. Potential Energy ..
119   !$OMP PARALLEL DO PRIVATE(j) SCHEDULE(static)
120   DO j=1,Ny
121     temp1(1:Nx,j)=0.5d0*(abs((u(1:Nx,j)+uold(1:Nx,j))*0.50d0))**2&
122           -0.125d0*Es*(abs(u(1:Nx,j))**4+abs(uold(1:Nx,j))**4)
123   END DO
124   !$OMP END PARALLEL DO
125   CALL dfftw_execute_dft_(planfxy,temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny))
126   enpot=REAL(abs(temp2(1,1)),kind(0d0))/REAL(Nx*Ny,kind(0d0))
127
128   en=enpot+enkin+enstr
129
130   END SUBROUTINE enercalc
```

Listing 14.11: A Fortran subroutine to save final results after solving the 2D Klein-Gordon equation.

```
1   SUBROUTINE saveresults(Nt,plotgap,time,en,enstr,enkin,enpot)
2   !---------------------------------------------------------------------
3   !
4   !
5   ! PURPOSE
6   !
7   ! This subroutine saves the energy and times stored during the
8   ! computation for the nonlinear Klein-Gordon equation
9   !
10  ! INPUT
11  !
12  ! .. Parameters ..
```

```
13  !  Nx       = number of modes in x - power of 2 for FFT
14  !  Ny       = number of modes in y - power of 2 for FFT
15  ! .. Vectors ..
16  !  time       = times at which save data
17  !  en       = total energy
18  !  enstr      = strain energy
19  !  enpot      = potential energy
20  !  enkin      = kinetic energy
21  !
22  ! OUTPUT
23  !
24  !
25  ! LOCAL VARIABLES
26  !
27  ! .. Scalars ..
28  !  n        = loop counter
29  ! .. Arrays ..
30  !   name_config   = array to hold the filename
31  !
32  ! REFERENCES
33  !
34  ! ACKNOWLEDGEMENTS
35  !
36  ! ACCURACY
37  !
38  ! ERROR INDICATORS AND WARNINGS
39  !
40  ! FURTHER COMMENTS
41  !-----------------------------------------------------------------------
42  ! External routines required
43  !
44  ! External libraries required
45  IMPLICIT NONE
46  ! Declare variables
47  INTEGER(kind=4), INTENT(IN)              :: plotgap,Nt
48  REAL(KIND=8), DIMENSION(1+Nt/plotgap), INTENT(IN) :: enpot, enkin
49  REAL(KIND=8), DIMENSION(1+Nt/plotgap), INTENT(IN) :: en,enstr,time
50  INTEGER(kind=4)                    :: j
51  CHARACTER*100                    :: name_config
52
53  name_config = 'tdata.dat'
54  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
55  REWIND(11)
56  DO j=1,1+Nt/plotgap
57    WRITE(11,*) time(j)
58  END DO
59  CLOSE(11)
60
61  name_config = 'en.dat'
62  OPEN(unit=11,FILE=name_config,status="UNKNOWN")
63  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
```

```
33  !  finish     = variable to record end time of program
34  !  count_rate   = variable for clock count rate
35  !  dt       = timestep
36  !  modescalereal  = Number to scale after backward FFT
37  !  ierr       = error code
38  !  plotnum      = number of plot
39  !  myid       = Process id
40  !  p_row      = number of rows for domain decomposition
41  !  p_col      = number of columns for domain decomposition
42  !  filesize     = total filesize
43  !  disp       = displacement to start writing data from
44  ! .. Arrays ..
45  !  unew       = approximate solution
46  !  vnew       = Fourier transform of approximate solution
47  !  u        = approximate solution
48  !  v        = Fourier transform of approximate solution
49  !  uold       = approximate solution
50  !  vold       = Fourier transform of approximate solution
51  !  nonlin       = nonlinear term, u^3
52  !  nonlinhat    = Fourier transform of nonlinear term, u^3
53  ! .. Vectors ..
54  !  kx       = fourier frequencies in x direction
55  !  ky       = fourier frequencies in y direction
56  !  kz       = fourier frequencies in z direction
57  !  x        = x locations
58  !  y        = y locations
59  !  z        = z locations
60  !  time       = times at which save data
61  !  en       = total energy
62  !  enstr      = strain energy
63  !  enpot      = potential energy
64  !  enkin      = kinetic energy
65  !  name_config    = array to store filename for data to be saved
66  !  fftfxyz      = array to setup 2D Fourier transform
67  !  fftbxyz      = array to setup 2D Fourier transform
68  ! .. Special Structures ..
69  !  decomp     = contains information on domain decomposition
70  !         see http://www.2decomp.org/ for more information
71  ! REFERENCES
72  !
73  ! ACKNOWLEDGEMENTS
74  !
75  ! ACCURACY
76  !
77  ! ERROR INDICATORS AND WARNINGS
78  !
79  ! FURTHER COMMENTS
80  ! Check that the initial iterate is consistent with the
81  ! boundary conditions for the domain specified
82  !-----------------------------------------------------------------------
83  ! External routines required
```

```
84  ! getgrid.f90 -- Get initial grid of points
85  !   initialdata.f90 -- Get initial data
86  ! enercalc.f90 -- Subroutine to calculate the energy
87  ! savedata.f90 -- Save initial data
88  ! storeold.f90 -- Store old data
89  ! External libraries required
90  !   2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
91  !     (http://www.2decomp.org/index.html)
92  ! MPI library
93  PROGRAM Kg
94  IMPLICIT NONE
95  USE decomp_2d
96  USE decomp_2d_fft
97  USE decomp_2d_io
98  INCLUDE 'mpif.h'
99  ! Declare variables
100 INTEGER(kind=4)        :: Nx, Ny, Nz, Nt, plotgap
101 REAL(kind=8), PARAMETER    :: &
102   pi=3.14159265358979323846264338327950288419716939937510d0
103 REAL(kind=8)         ::  Lx,Ly,Lz,Es,dt,starttime,modescalereal
104 COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE  ::  kx,ky,kz
105 REAL(kind=8),    DIMENSION(:), ALLOCATABLE  ::  x,y,z
106 COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE::  u,nonlin
107 COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE::  v,nonlinhat
108 COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE::  uold
109 COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE::  vold
110 COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE::  unew
111 COMPLEX(kind=8), DIMENSION(:,:,:), ALLOCATABLE::  vnew
112 REAL(kind=8), DIMENSION(:,:,:), ALLOCATABLE :: savearray
113 REAL(kind=8), DIMENSION(:), ALLOCATABLE ::  time,enkin,enstr,enpot,en
114 INTEGER(kind=4)    ::  ierr,i,j,k,n,allocatestatus,myid,numprocs
115 INTEGER(kind=4)    ::  start, finish, count_rate, plotnum
116 TYPE(DECOMP_INFO) ::  decomp
117 INTEGER(kind=MPI_OFFSET_KIND) :: filesize, disp
118 INTEGER(kind=4) ::  p_row=0, p_col=0
119 CHARACTER*100 ::  name_config
120     ! initialisation of 2DECOMP&FFT
121 CALL MPI_INIT(ierr)
122 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
123 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
124
125 CALL readinputfile(Nx,Ny,Nz,Nt,plotgap,Lx,Ly,Lz, &
126             Es,DT,starttime,myid,ierr)
127 ! do automatic domain decomposition
128 CALL decomp_2d_init(Nx,Ny,Nz,p_row,p_col)
129 ! get information about domain decomposition choosen
130 CALL decomp_info_init(Nx,Ny,Nz,decomp)
131 ! initialise FFT library
132 CALL decomp_2d_fft_init
133 ALLOCATE(kx(decomp%zst(1):decomp%zen(1)),&
134     ky(decomp%zst(2):decomp%zen(2)),&
```

```
135       kz(decomp%zst(3):decomp%zen(3)),&
136        x(decomp%xst(1):decomp%xen(1)),&
137        y(decomp%xst(2):decomp%xen(2)),&
138        z(decomp%xst(3):decomp%xen(3)),&
139       u(decomp%xst(1):decomp%xen(1),&
140           decomp%xst(2):decomp%xen(2),&
141           decomp%xst(3):decomp%xen(3)),&
142       v(decomp%zst(1):decomp%zen(1),&
143                decomp%zst(2):decomp%zen(2),&
144                decomp%zst(3):decomp%zen(3)),&
145       nonlin(decomp%xst(1):decomp%xen(1),&
146           decomp%xst(2):decomp%xen(2),&
147           decomp%xst(3):decomp%xen(3)),&
148       nonlinhat(decomp%zst(1):decomp%zen(1),&
149                decomp%zst(2):decomp%zen(2),&
150                decomp%zst(3):decomp%zen(3)),&
151       uold(decomp%xst(1):decomp%xen(1),&
152           decomp%xst(2):decomp%xen(2),&
153           decomp%xst(3):decomp%xen(3)),&
154       vold(decomp%zst(1):decomp%zen(1),&
155                decomp%zst(2):decomp%zen(2),&
156                decomp%zst(3):decomp%zen(3)),&
157       unew(decomp%xst(1):decomp%xen(1),&
158           decomp%xst(2):decomp%xen(2),&
159           decomp%xst(3):decomp%xen(3)),&
160       vnew(decomp%zst(1):decomp%zen(1),&
161                decomp%zst(2):decomp%zen(2),&
162                decomp%zst(3):decomp%zen(3)),&
163       savearray(decomp%xst(1):decomp%xen(1),&
164           decomp%xst(2):decomp%xen(2),&
165           decomp%xst(3):decomp%xen(3)),&
166       time(1:1+Nt/plotgap),enkin(1:1+Nt/plotgap),&
167       enstr(1:1+Nt/plotgap),enpot(1:1+Nt/plotgap),&
168       en(1:1+Nt/plotgap),stat=allocatestatus)
169   IF (allocatestatus .ne. 0) stop
170   IF (myid.eq.0) THEN
171     PRINT *,'allocated arrays'
172   END IF
173   ! setup fourier frequencies
174   CALL getgrid(myid,Nx,Ny,Nz,Lx,Ly,Lz,pi,name_config,x,y,z,kx,ky,kz,decomp
      )
175   IF (myid.eq.0) THEN
176     PRINT *,'Setup grid and fourier frequencies'
177   END IF
178   CALL initialdata(Nx,Ny,Nz,x,y,z,u,uold,decomp)
179   plotnum=1
180   name_config = 'data/u'
181   savearray=REAL(u)
182   CALL savedata(Nx,Ny,Nz,plotnum,name_config,savearray,decomp)
183
184   CALL decomp_2d_fft_3d(u,v,DECOMP_2D_FFT_FORWARD)
```

```
185   CALL decomp_2d_fft_3d(uold,vold,DECOMP_2D_FFT_FORWARD)
186
187   modescalereal=1.0d0/REAL(Nx,KIND(0d0))
188   modescalereal=modescalereal/REAL(Ny,KIND(0d0))
189   modescalereal=modescalereal/REAL(Nz,KIND(0d0))
190
191   CALL enercalc(myid,Nx,Ny,Nz,dt,Es,modescalereal,&
192         enkin(plotnum),enstr(plotnum),&
193         enpot(plotnum),en(plotnum),&
194         kx,ky,kz,nonlin,nonlinhat,&
195         v,vold,u,uold,decomp)
196
197   IF (myid.eq.0) THEN
198     PRINT *,'Got initial data, starting timestepping'
199   END IF
200   time(plotnum)=0.0d0+starttime
201     CALL system_clock(start,count_rate)
202   DO n=1,Nt
203     DO k=decomp%xst(3),decomp%xen(3)
204       DO j=decomp%xst(2),decomp%xen(2)
205         DO i=decomp%xst(1),decomp%xen(1)
206           nonlin(i,j,k)=(abs(u(i,j,k))*2)*u(i,j,k)
207         END DO
208       END DO
209     END DO
210     CALL decomp_2d_fft_3d(nonlin,nonlinhat,DECOMP_2D_FFT_FORWARD)
211     DO k=decomp%zst(3),decomp%zen(3)
212       DO j=decomp%zst(2),decomp%zen(2)
213         DO i=decomp%zst(1),decomp%zen(1)
214           vnew(i,j,k)=&
215           ( 0.25*(kx(i)*kx(i) + ky(j)*ky(j)+ kz(k)*kz(k)-1.0d0)&
216           *(2.0d0*v(i,j,k)+vold(i,j,k))&
217           +(2.0d0*v(i,j,k)-vold(i,j,k))/(dt*dt)&
218           +Es*nonlinhat(i,j,k) )&
219           /(1/(dt*dt)-0.25*(kx(i)*kx(i)+ ky(j)*ky(j)+ kz(k)*kz(k)-1.0d0))
220         END DO
221       END DO
222     END DO
223     CALL decomp_2d_fft_3d(vnew,unew,DECOMP_2D_FFT_BACKWARD)
224     ! normalize result
225     DO k=decomp%xst(3),decomp%xen(3)
226       DO j=decomp%xst(2),decomp%xen(2)
227         DO i=decomp%xst(1),decomp%xen(1)
228           unew(i,j,k)=unew(i,j,k)*modescalereal
229         END DO
230       END DO
231     END DO
232     IF (mod(n,plotgap)==0) THEN
233       plotnum=plotnum+1
234       time(plotnum)=n*dt+starttime
235       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.

```
1   SUBROUTINE getgrid(myid,Nx,Ny,Nz,Lx,Ly,Lz,pi,name_config,x,y,z,kx,ky,kz,
        decomp)
2   !-----------------------------------------------------------------------
3   !
4   !
5   ! PURPOSE
6   !
7   ! This subroutine gets grid points and fourier frequencies for a
8   ! 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
```

```
60  !
61  ! External libraries required
62  ! 2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
63  !     (http://www.2decomp.org/index.html)
64  ! MPI library
65  IMPLICIT NONE
66  USE decomp_2d
67  INCLUDE 'mpif.h'
68  ! Declare variables
69  INTEGER(KIND=4), INTENT(IN)              :: myid,Nx,Ny,Nz
70  REAL(kind=8), INTENT(IN)               :: Lx,Ly,Lz,pi
71  TYPE(DECOMP_INFO), INTENT(IN)            ::  decomp
72  REAL(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1)), INTENT(OUT)    :: x
73  REAL(KIND=8), DIMENSION(decomp%xst(2):decomp%xen(2)), INTENT(OUT)    :: y
74  REAL(KIND=8), DIMENSION(decomp%xst(3):decomp%xen(3)), INTENT(OUT)    :: z
75  COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1)), INTENT(OUT)::
       kx
76  COMPLEX(KIND=8), DIMENSION(decomp%zst(2):decomp%zen(2)), INTENT(OUT)::
       ky
77  COMPLEX(KIND=8), DIMENSION(decomp%zst(3):decomp%zen(3)), INTENT(OUT)::
       kz
78  CHARACTER*100, INTENT(OUT)               :: name_config
79  INTEGER(kind=4)                     :: i,j,k
80
81
82  DO i = 1,1+ Nx/2
83    IF ((i.GE.decomp%zst(1)).AND.(i.LE.decomp%zen(1))) THEN
84      kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
85    END IF
86  END DO
87  IF ((Nx/2 + 1 .GE.decomp%zst(1)).AND.(Nx/2 + 1 .LE.decomp%zen(1))) THEN
88    kx( Nx/2 + 1 ) = 0.0d0
89  ENDIF
90  DO i = Nx/2+2, Nx
91    IF ((i.GE.decomp%zst(1)).AND.(i.LE.decomp%zen(1))) THEN
92      Kx( i) = cmplx(0.0d0,-1.0d0)*REAL(1-i+Nx,KIND(0d0))/Lx
93    ENDIF
94  END DO
95  DO i=decomp%xst(1),decomp%xen(1)
96    x(i)=(-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)))*pi*Lx
97  END DO
98
99  DO j = 1,1+ Ny/2
100   IF ((j.GE.decomp%zst(2)).AND.(j.LE.decomp%zen(2))) THEN
101     ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
102   END IF
103 END DO
104 IF ((Ny/2 + 1 .GE.decomp%zst(2)).AND.(Ny/2 + 1 .LE.decomp%zen(2))) THEN
105   ky( Ny/2 + 1 ) = 0.0d0
106 ENDIF
107 DO j = Ny/2+2, Ny
```

```
108     IF ((j.GE.decomp%zst(2)).AND.(j.LE.decomp%zen(2))) THEN
109       ky(j) = cmplx(0.0d0,-1.0d0)*REAL(1-j+Ny,KIND(0d0))/Ly
110     ENDIF
111   END DO
112   DO j=decomp%xst(2),decomp%xen(2)
113     y(j)=(-1.0d0 + 2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)))*pi*Ly
114   END DO
115
116   DO k = 1,1+ Nz/2
117     IF ((k.GE.decomp%zst(3)).AND.(k.LE.decomp%zen(3))) THEN
118       kz(k)= cmplx(0.0d0,1.0d0)*REAL(k-1,kind(0d0))/Lz
119     END IF
120   END DO
121   IF ((Nz/2 + 1 .GE.decomp%zst(3)).AND.(Nz/2 + 1 .LE.decomp%zen(3))) THEN
122     kz( Nz/2 + 1 ) = 0.0d0
123   ENDIF
124   DO k = Nz/2+2, Nz
125     IF ((k.GE.decomp%zst(3)).AND.(k.LE.decomp%zen(3))) THEN
126       kz(k) = cmplx(0.0d0,-1.0d0)*REAL(1-k+Nz,KIND(0d0))/Lz
127     ENDIF
128   END DO
129   DO k=decomp%xst(3),decomp%xen(3)
130     z(k)=(-1.0d0 + 2.0d0*REAL(k-1,kind(0d0))/REAL(Nz,kind(0d0)))*pi*Lz
131   END DO
132
133   IF (myid.eq.0) THEN
134     ! Save x grid points in text format
135     name_config = 'xcoord.dat'
136     OPEN(unit=11,FILE=name_config,status="UNKNOWN")
137     REWIND(11)
138     DO i=1,Nx
139       WRITE(11,*) (-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)))*
            pi*Lx
140     END DO
141     CLOSE(11)
142     ! Save y grid points in text format
143     name_config = 'ycoord.dat'
144     OPEN(unit=11,FILE=name_config,status="UNKNOWN")
145     REWIND(11)
146     DO j=1,Ny
147       WRITE(11,*) (-1.0d0 + 2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)))*
            pi*Ly
148     END DO
149     CLOSE(11)
150     ! Save z grid points in text format
151     name_config = 'zcoord.dat'
152     OPEN(unit=11,FILE=name_config,status="UNKNOWN")
153     REWIND(11)
154     DO k=1,Nz
155       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
  !
```

```
42  ! REFERENCES
43  !
44  ! ACKNOWLEDGEMENTS
45  !
46  ! ACCURACY
47  !
48  ! ERROR INDICATORS AND WARNINGS
49  !
50  ! FURTHER COMMENTS
51  ! Check that the initial iterate is consistent with the
52  ! boundary conditions for the domain specified
53  !-----------------------------------------------------------------------
54  ! External routines required
55  !
56  ! External libraries required
57  ! 2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
58  !     (http://www.2decomp.org/index.html)
59  ! MPI library
60  IMPLICIT NONE
61  USE decomp_2d
62  INCLUDE 'mpif.h'
63  ! Declare variables
64  INTEGER(KIND=4), INTENT(IN)              :: Nx,Ny,Nz
65  TYPE(DECOMP_INFO), INTENT(IN)            ::  decomp
66  REAL(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1)), INTENT(IN) :: x
67  REAL(KIND=8), DIMENSION(decomp%xst(2):decomp%xen(2)), INTENT(IN) :: y
68  REAL(KIND=8), DIMENSION(decomp%xst(3):decomp%xen(3)), INTENT(IN) :: z
69  COMPLEX(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
70                 decomp%xst(2):decomp%xen(2),&
71                 decomp%xst(3):decomp%xen(3)),&
72                         INTENT(OUT) :: u,uold
73  INTEGER(kind=4)                    :: i,j,k
74
75  DO k=decomp%xst(3),decomp%xen(3)
76    DO j=decomp%xst(2),decomp%xen(2)
77      DO i=decomp%xst(1),decomp%xen(1)
78        u(i,j,k)=0.5d0*exp(-1.0d0*(x(i)**2 +y(j)**2+z(k)**2))!*&
79          !sin(10.0d0*x(i)+12.0d0*y(j))
80      END DO
81    END DO
82  END DO
83  DO k=decomp%xst(3),decomp%xen(3)
84    DO j=decomp%xst(2),decomp%xen(2)
85      DO i=decomp%xst(1),decomp%xen(1)
86        uold(i,j,k)=0.5d0*exp(-1.0d0*(x(i)**2 +y(j)**2+z(k)**2))!*&
87          !sin(10.0d0*x(i)+12.0d0*y(j))
88        END DO
89    END DO
90  END DO
91
92  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
```

```
18  !  .. Arrays ..
19  !  unew       = approximate solution
20  !  vnew       = Fourier transform of approximate solution
21  !  u        = approximate solution
22  !  v        = Fourier transform of approximate solution
23  !  uold       = approximate solution
24  !  vold       = Fourier transform of approximate solution
25  ! .. Special Structures ..
26  !  decomp     = contains information on domain decomposition
27  !         see http://www.2decomp.org/ for more information
28  ! OUTPUT
29  !
30  !  u        = approximate solution
31  !  v        = Fourier transform of approximate solution
32  !  uold       = approximate solution
33  !  vold       = Fourier transform of approximate solution
34  !
35  ! LOCAL VARIABLES
36  !
37  ! .. Scalars ..
38  !  i        = loop counter in x direction
39  !  j        = loop counter in y direction
40  !  k        = loop counter in z direction
41  !
42  ! REFERENCES
43  !
44  ! ACKNOWLEDGEMENTS
45  !
46  ! ACCURACY
47  !
48  ! ERROR INDICATORS AND WARNINGS
49  !
50  ! FURTHER COMMENTS
51  !-----------------------------------------------------------------------
52  ! External routines required
53  !
54  ! External libraries required
55  ! 2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
56  !     (http://www.2decomp.org/index.html)
57  ! MPI library
58  IMPLICIT NONE
59  USE decomp_2d
60  USE decomp_2d_fft
61  USE decomp_2d_io
62  INCLUDE 'mpif.h'
63  ! Declare variables
64  INTEGER(KIND=4), INTENT(IN)                :: Nx,Ny,Nz
65  TYPE(DECOMP_INFO), INTENT(IN)              ::  decomp
66  COMPLEX(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
67                 decomp%xst(2):decomp%xen(2),&
68                 decomp%xst(3):decomp%xen(3)), INTENT(OUT):: uold
```

```
69  COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1),&
70                      decomp%zst(2):decomp%zen(2),&
71                      decomp%zst(3):decomp%zen(3)), INTENT(OUT):: vold
72  COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1),&
73                      decomp%zst(2):decomp%zen(2),&
74                      decomp%zst(3):decomp%zen(3)), INTENT(INOUT):: v
75  COMPLEX(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
76                     decomp%xst(2):decomp%xen(2),&
77                     decomp%xst(3):decomp%xen(3)), INTENT(INOUT):: u
78  COMPLEX(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
79                     decomp%xst(2):decomp%xen(2),&
80                     decomp%xst(3):decomp%xen(3)), INTENT(IN):: unew
81  COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1),&
82                      decomp%zst(2):decomp%zen(2),&
83                      decomp%zst(3):decomp%zen(3)), INTENT(IN):: vnew
84  INTEGER(kind=4)                                 :: i,j,k
85
86  DO k=decomp%zst(3),decomp%zen(3)
87    DO j=decomp%zst(2),decomp%zen(2)
88      DO i=decomp%zst(1),decomp%zen(1)
89        vold(i,j,k)=v(i,j,k)
90      END DO
91    END DO
92  END DO
93  DO k=decomp%xst(3),decomp%xen(3)
94    DO j=decomp%xst(2),decomp%xen(2)
95      DO i=decomp%xst(1),decomp%xen(1)
96        uold(i,j,k)=u(i,j,k)
97      END DO
98    END DO
99  END DO
100 DO k=decomp%xst(3),decomp%xen(3)
101   DO j=decomp%xst(2),decomp%xen(2)
102     DO i=decomp%xst(1),decomp%xen(1)
103       u(i,j,k)=unew(i,j,k)
104     END DO
105   END DO
106 END DO
107 DO k=decomp%zst(3),decomp%zen(3)
108   DO j=decomp%zst(2),decomp%zen(2)
109     DO i=decomp%zst(1),decomp%zen(1)
110       v(i,j,k)=vnew(i,j,k)
111     END DO
112   END DO
113 END DO
114
115 END SUBROUTINE storeold
```

Listing 14.19: A Fortran subroutine to calculate the energy when solving the 3D Klein-Gordon equation.

```
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 ..
```

```
50  !  i        = loop counter in x direction
51  !  j        = loop counter in y direction
52  !  k        = loop counter in z direction
53  !
54  ! REFERENCES
55  !
56  ! ACKNOWLEDGEMENTS
57  !
58  ! ACCURACY
59  !
60  ! ERROR INDICATORS AND WARNINGS
61  !
62  ! FURTHER COMMENTS
63  ! Check that the initial iterate is consistent with the
64  ! boundary conditions for the domain specified
65  !-----------------------------------------------------------------------
66  ! External routines required
67  !
68  ! External libraries required
69  ! 2DECOMP&FFT  -- Domain decomposition and Fast Fourier Library
70  !     (http://www.2decomp.org/index.html)
71  ! MPI library
72  IMPLICIT NONE
73  USE decomp_2d
74  USE decomp_2d_fft
75  USE decomp_2d_io
76  INCLUDE 'mpif.h'
77  ! Declare variables
78  INTEGER(KIND=4), INTENT(IN)                 :: Nx,Ny,Nz,myid
79  REAL(KIND=8), INTENT(IN)                   :: dt,Es,modescalereal
80  TYPE(DECOMP_INFO), INTENT(IN)               :: decomp
81  COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1)),INTENT(IN)  ::
       kx
82  COMPLEX(KIND=8), DIMENSION(decomp%zst(2):decomp%zen(2)),INTENT(IN)  ::
       ky
83  COMPLEX(KIND=8), DIMENSION(decomp%zst(3):decomp%zen(3)),INTENT(IN)  ::
       kz
84  COMPLEX(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
85                  decomp%xst(2):decomp%xen(2),&
86                  decomp%xst(3):decomp%xen(3)),&
87                            INTENT(IN)  :: u,uold
88  COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1),&
89                      decomp%zst(2):decomp%zen(2),&
90                      decomp%zst(3):decomp%zen(3)),&
91                                INTENT(IN)  :: v,vold
92  COMPLEX(KIND=8), DIMENSION(decomp%xst(1):decomp%xen(1),&
93                  decomp%xst(2):decomp%xen(2),&
94                  decomp%xst(3):decomp%xen(3)),&
95                            INTENT(INOUT) :: tempu
96  COMPLEX(KIND=8), DIMENSION(decomp%zst(1):decomp%zen(1),&
97                      decomp%zst(2):decomp%zen(2),&
```

```
                      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.

```
1
2
3
4
5
```

```
6   SUBROUTINE saveresults(Nt,plotgap,time,en,enstr,enkin,enpot)
7   !-----------------------------------------------------------------------
8   !
9   !
10  ! PURPOSE
11  !
12  ! This subroutine saves the energy and times stored during the
13  ! computation for the nonlinear Klein-Gordon equation
14  !
15  ! INPUT
16  !
17  ! .. Parameters ..
18  !  Nt        = number of timesteps
19  !  plotgap       = number of timesteps between plots
20  ! .. Vectors ..
21  !  time        = times at which save data
22  !  en        = total energy
23  !  enstr       = strain energy
24  !  enpot       = potential energy
25  !  enkin       = kinetic energy
26  !
27  ! OUTPUT
28  !
29  !
30  ! LOCAL VARIABLES
31  !
32  ! .. Scalars ..
33  !  n         = loop counter
34  ! .. Arrays ..
35  !   name_config    = array to hold the filename
36  !
37  ! REFERENCES
38  !
39  ! ACKNOWLEDGEMENTS
40  !
41  ! ACCURACY
42  !
43  ! ERROR INDICATORS AND WARNINGS
44  !
45  ! FURTHER COMMENTS
46  !-----------------------------------------------------------------------
47  ! External routines required
48  !
49  ! External libraries required
50
51  ! Declare variables
52  IMPLICIT NONE
53  INTEGER(kind=4), INTENT(IN)              :: plotgap,Nt
54  REAL(KIND=8), DIMENSION(1:1+Nt/plotgap), INTENT(IN) :: enpot, enkin
55  REAL(KIND=8), DIMENSION(1:1+Nt/plotgap), INTENT(IN) :: en,enstr,time
56  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)
```

```
3  !     ------------------------------------------------------------------------------
4  !
5  !
6  !  PURPOSE
7  !
8  ! Read inputfile intialize parameters, which are stocked in the Input
       File
9  !
10 ! .. INPUT ..
11 ! Nx         = number of modes in the x direction
12 ! Ny         = number of modes in the y direction
13 ! Nz         = number of modes in the z direction
14 ! Nt         = the number of timesteps
15 ! plotgap     = the number of timesteps to take before plotting
16 ! myid       = number of MPI process
17 ! ierr       = MPI error output variable
18 ! Lx         = size of the periodic domain of computation in x direction
19 ! Ly         = size of the periodic domain of computation in y direction
20 ! Lz         = size of the periodic domain of computation in z direction
21 ! DT         = the time step
22 ! starttime   = initial time of computation
23 ! InputFileName = name of the Input File
24 ! REFERENCES
25 !
26 ! ACCURACY
27 !
28 ! ERROR INDICATORS AND WARNINGS
29 !
30 ! FURTHER COMMENTS
31 !     ------------------------------------------------------------------------------
32 ! EXTERNAL ROUTINES REQUIRED
33 IMPLICIT NONE
34 INCLUDE 'mpif.h'
35 ! .. Scalar Arguments ..
36 INTEGER(KIND=4), INTENT(IN)    ::  myid
37 INTEGER(KIND=4), INTENT(OUT)   ::  Nx,Ny,Nz,Nt
38 INTEGER(KIND=4), INTENT(OUT)   ::  plotgap, ierr
39 REAL(KIND=8), INTENT(OUT)    ::  Lx, Ly, Lz, DT, starttime, Es
40 ! .. Local scalars ..
41 INTEGER(KIND=4)          ::  stat
42 ! .. Local Arrays ..
43 CHARACTER*40           ::  InputFileName
44 INTEGER(KIND=4), DIMENSION(1:5) ::  intcomm
45 REAL(KIND=8), DIMENSION(1:6)  ::  dpcomm
46
47 IF(myid.eq.0) THEN
```

```
48      CALL GET_ENVIRONMENT_VARIABLE(NAME="inputfile",VALUE=InputFileName,
           STATUS=stat)
49      IF(stat.NE.0) THEN
50        PRINT*,"Set environment variable inputfile to the name of the"
51        PRINT*,"file where the simulation parameters are set"
52        STOP
53      END IF
54      OPEN(unit=11,FILE=trim(InputFileName),status="OLD")
55      REWIND(11)
56      READ(11,*) intcomm(1), intcomm(2), intcomm(3), intcomm(4), intcomm(5),
            &
57          dpcomm(1), dpcomm(2), dpcomm(3), dpcomm(4), dpcomm(5), dpcomm(6)
58      CLOSE(11)
59      PRINT *,"NX ",intcomm(1)
60      PRINT *,"NY ",intcomm(2)
61      PRINT *,"NZ ",intcomm(3)
62      PRINT *,"NT ",intcomm(4)
63      PRINT *,"plotgap ",intcomm(5)
64      PRINT *,"Lx ",dpcomm(1)
65      PRINT *,"Ly ",dpcomm(2)
66      PRINT *,"Lz ",dpcomm(3)
67      PRINT *,"Es ",dpcomm(4)
68      PRINT *,"Dt ",dpcomm(5)
69      PRINT *,"strart time ",dpcomm(6)
70      PRINT *,"Read inputfile"
71    END IF
72    CALL MPI_BCAST(dpcomm,6,MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD,ierr)
73    CALL MPI_BCAST(intcomm,5,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
74
75    Nx=intcomm(1)
76    Ny=intcomm(2)
77    Nz=intcomm(3)
78    Nt=intcomm(4)
79    plotgap=intcomm(5)
80    Lx=dpcomm(1)
81    Ly=dpcomm(2)
82    Lz=dpcomm(3)
83    Es=dpcomm(4)
84    DT=dpcomm(5)
85    starttime=dpcomm(6)
86
87    END SUBROUTINE readinputfile
```

Listing 14.22: An example makefile for compiling the MPI program in listing 14.14.

```
1   # All settings here for use on FLUX, a cluster at the University of
       Michigan
2   # Center for Advanced Computing (CAC), using INTEL nehalem hardware,
3   # Need to load fftw module
4
```

```
  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
```

```
18  ! Es      = focusing or defocusing
19  ! Dt      = the time step
20  !
21  ! REFERENCES
22  !
23  ! ACCURACY
24  !
25  ! ERROR INDICATORS AND WARNINGS
26  !
27  ! FURTHER COMMENTS
28  !
      -------------------------------------------------------------------------------

29  !   EXTERNAL ROUTINES REQUIRED
30  IMPLICIT NONE
31  ! .. Scalar Arguments ..
32  INTEGER(KIND=4)     ::  Nx, Ny, Nz, Nt, plotgap
33  REAL(KIND=8)        ::  Lx, Ly, Lz,  DT, time, Es
34  ! .. Local scalars ..
35  INTEGER(KIND=4)     ::  stat,plotnum,ind,n,numplots
36  ! .. Local Arrays ..
37  CHARACTER*50        ::  InputFileName, OutputFileName, OutputFileName2
38  CHARACTER*10        ::  number_file
39  InputFileName='INPUTFILE'
40  OPEN(unit=11,FILE=trim(InputFileName),status="OLD")
41  REWIND(11)
42  READ(11,*) Nx, Ny, Nz, Nt, plotgap, Lx, Ly, Lz, Es, DT, time
43  CLOSE(11)
44
45  plotnum=1
46  numplots=1+Nt/plotgap
47  DO n=1,numplots
48    OutputFileName = 'data/u'
49    ind = index(OutputFileName,' ') - 1
50    WRITE(number_file,'(i0)') 10000000+plotnum
51    OutputFileName = OutputFileName(1:ind)//number_file
52    ind = index(OutputFileName,' ') - 1
53    OutputFileName = OutputFileName(1:ind)//'.bov'
54    OutputFileName2='u'
55    ind = index(OutputFileName2,' ') - 1
56    OutputFileName2 = OutputFileName2(1:ind)//number_file
57    ind = index(OutputFileName2,' ') - 1
58    OutputFileName2 = OutputFileName2(1:ind)//'.datbin'
59    OPEN(unit=11,FILE=trim(OutputFileName),status="UNKNOWN")
60    REWIND(11)
61    WRITE(11,*) 'TIME: ',time
62    WRITE(11,*) 'DATA_FILE: ',trim(OutputFileName2)
63    WRITE(11,*) 'DATA_SIZE: ', Nx, Ny, Nz
64    WRITE(11,*) 'DATA_FORMAT: DOUBLE'
65    WRITE(11,*) 'VARIABLE: u'
66    WRITE(11,*) 'DATA_ENDIAN: LITTLE'
```

```
67      WRITE(11,*) 'CENTERING: ZONAL'
68      WRITE(11,*) 'BRICK_ORIGIN:', -Nx/2, -Ny/2, -Nz/2
69      WRITE(11,*) 'BRICK_SIZE:', Nx, Ny, Nz
70      WRITE(11,*) 'DIVIDE_BRICK: true'
71      WRITE(11,*) 'DATA_BRICKLETS:', Nx/2, Ny/2, Nz/2
72      CLOSE(11)
73
74      time=time+plotgap*DT
75      plotnum=plotnum+1
76    END DO
77    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

$$\begin{aligned} i\frac{\partial}{\partial t}(\phi + \xi) &= -(\phi - \xi) \\ i\frac{\partial}{\partial t}(\phi - \xi) &= -\Delta(\phi + \xi) - (\phi + \xi) \pm |\phi + \xi|^2 (\phi + \xi). \end{aligned}$$

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] 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] **The (Unfninished) PDE coffee table book.** L. N. Trefethen and K. Embree (Ed.), 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, programs using CUDA Fortran and OpenACC extensions to Fortran are included. 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.

```
1
2
3
4
5   !-----------------------------------------------------------------
6   !
```

```
7   ! PURPOSE
8   !
9   ! This program numerically solves the 2D incompressible Navier-Stokes
10  ! on a Square Domain [0,1]x[0,1] using pseudo-spectral methods and
11  ! Crank-Nicolson timestepping. The numerical solution is compared to
12  ! the exact Taylor-Green Vortex Solution.
13  !
14  ! AUTHORS
15  !
16  ! B. Cloutier, B.K. Muite, P. Rigge
17  ! 4 June 2012
18  !
19  ! Periodic free-slip boundary conditions and Initial conditions:
20  ! u(x,y,0)=sin(2*pi*x)cos(2*pi*y)
21  ! v(x,y,0)=-cos(2*pi*x)sin(2*pi*y)
22  ! Analytical Solution (subscript denote derivatives):
23  ! u(x,y,t)=sin(2*pi*x)cos(2*pi*y)exp(-8*pi^2*t/Re)
24  ! v(x,y,t)=-cos(2*pi*x)sin(2*pi*y)exp(-8*pi^2*t/Re)
25  !   u_y(x,y,t)=-2*pi*sin(2*pi*x)sin(2*pi*y)exp(-8*pi^2*t/Re)
26  ! v_x(x,y,t)=2*pi*sin(2*pi*x)sin(2*pi*y)exp(-8*pi^2*t/Re)
27  ! omega=v_x-u_y
28  !
29  ! .. Parameters ..
30  !  Nx       = number of modes in x - power of 2 for FFT
31  !  Ny       = number of modes in y - power of 2 for FFT
32  !  nplots     = number of plots produced
33  !  plotgap      = number of timesteps inbetween plots
34  !  Re         = dimensionless Renold's number
35  !  ReInv      = 1/Re for optimization
36  !  dt       = timestep size
37  !  dtInv      = 1/dt for optimization
38  !  tol        = determines when convergences is reached
39  !  numthreads   = number of CPUs used in calculation
40  ! .. Scalars ..
41  !  i        = loop counter in x direction
42  !  j        = loop counter in y direction
43  !  n        = loop counter for timesteps direction
44  !  allocatestatus = error indicator during allocation
45  !  time       = times at which data is saved
46  !  chg        = error at each iteration
47  ! .. Arrays (gpu) ..
48  !  omeg_d       = vorticity in real space
49  !  omeghat_d      = 2D Fourier transform of vorticity
50  !           at next iterate
51  !  omegoldhat_d   = 2D Fourier transform of vorticity at previous
52  !           iterate
53  !  nloldhat_d     = nonlinear term in Fourier space
54  !           at previous iterate
55  !  psihat_d     = 2D Fourier transform of streamfunction
56  !           at next iteration
57  !  temp1_d/temp2_d/temp3_d    = reusable complex/real space used for
```

```
58  !               calculations. This reduces number of
59  !               arrays stored.
60  ! .. Vectors (gpu) ..
61  !  kx_d        = fourier frequencies in x direction
62  !  ky_d        = fourier frequencies in y direction
63  !  x_d         = x locations
64  !  y_d         = y locations
65  !  name_config    = array to store filename for data to be saved
66  ! REFERENCES
67  !
68  ! ACKNOWLEDGEMENTS
69  !
70  ! ACCURACY
71  !
72  ! ERROR INDICATORS AND WARNINGS
73  !
74  ! FURTHER COMMENTS
75  ! This program has not been fully optimized.
76  !-----------------------------------------------------------------------
77  module precision
78  ! Precision control
79
80  integer, parameter, public :: Single = kind(0.0) ! Single precision
81  integer, parameter, public :: Double = kind(0.0d0) ! Double precision
82
83  integer, parameter, public :: fp_kind = Double
84  !integer, parameter, public :: fp_kind = Single
85
86  end module precision
87
88  module cufft
89
90  integer, public :: CUFFT_FORWARD = -1
91  integer, public :: CUFFT_INVERSE = 1
92  integer, public :: CUFFT_R2C = Z'2a' ! Real to Complex (interleaved)
93  integer, public :: CUFFT_C2R = Z'2c' ! Complex (interleaved) to Real
94  integer, public :: CUFFT_C2C = Z'29' ! Complex to Complex, interleaved
95  integer, public :: CUFFT_D2Z = Z'6a' ! Double to Double-Complex
96  integer, public :: CUFFT_Z2D = Z'6c' ! Double-Complex to Double
97  integer, public :: CUFFT_Z2Z = Z'69' ! Double-Complex to Double-Complex
98
99  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
100 !
101 ! cufftPlan2d(cufftHandle *plan, int nx,int ny, cufftType type)
102 !
103 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
104
105 interface cufftPlan2d
106 subroutine cufftPlan2d(plan, nx, ny, type) bind(C,name='cufftPlan2d')
107 use iso_c_binding
108 integer(c_int):: plan
```

```
109   integer(c_int),value:: nx, ny, type
110   end subroutine cufftPlan2d
111   end interface cufftPlan2d
112
113   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
114   !
115   ! cufftDestroy(cufftHandle plan)
116   !
117   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
118
119   interface cufftDestroy
120   subroutine cufftDestroy(plan) bind(C,name='cufftDestroy')
121   use iso_c_binding
122   integer(c_int),value:: plan
123   end subroutine cufftDestroy
124   end interface cufftDestroy
125
126   interface cufftExecD2Z
127     subroutine cufftExecD2Z(plan, idata, odata) &
128           & bind(C,name='cufftExecD2Z')
129       use iso_c_binding
130       use precision
131       integer(c_int),  value  :: plan
132       real(fp_kind),   device :: idata(1:nx,1:ny)
133       complex(fp_kind),device :: odata(1:nx,1:ny)
134     end subroutine cufftExecD2Z
135   end interface cufftExecD2Z
136
137   interface cufftExecZ2D
138     subroutine cufftExecZ2D(plan, idata, odata) &
139           & bind(C,name='cufftExecZ2D')
140       use iso_c_binding
141       use precision
142       integer(c_int),value:: plan
143       complex(fp_kind),device:: idata(1:nx,1:ny)
144       real(fp_kind),device :: odata(1:nx,1:ny)
145     end subroutine cufftExecZ2D
146   end interface cufftExecZ2D
147
148   end module cufft
149
150   PROGRAM main
151   use precision
152   use cufft
153   ! declare variables
154     IMPLICIT NONE
155     INTEGER(kind=4), PARAMETER    ::  Nx=4096
156   INTEGER(kind=4), PARAMETER    ::  Ny=4096
157   INTEGER(kind=8)          ::  temp=10000000
158   REAL(fp_kind), PARAMETER    ::  dt=0.000125d0 !dt=0.000002d0
159   REAL(fp_kind), PARAMETER    ::  dtInv=1.0d0/REAL(dt,kind(0d0))
```

```
160   REAL(fp_kind), PARAMETER  &
161     ::  pi=3.14159265358979323846264338327950288419716939937510d0
162   REAL(fp_kind), PARAMETER    ::  Re=1.0d0
163   REAL(fp_kind), PARAMETER    ::  ReInv=1.0d0/REAL(Re,kind(0d0))
164   REAL(fp_kind), PARAMETER    ::  tol=0.1d0**10
165   REAL(fp_kind)          ::  scalemodes,chg
166   INTEGER(kind=4), PARAMETER    ::  nplots=1,plotgap=20
167   REAL(fp_kind),DIMENSION(:), ALLOCATABLE   ::  x,y
168   REAL(fp_kind),DIMENSION(:,:), ALLOCATABLE ::  omeg,omegexact
169   INTEGER(kind=4)               ::  i,j,n,t, AllocateStatus
170   INTEGER(kind=4)               ::  planz2d,pland2z, kersize
171   !variables used for saving data and timing
172   INTEGER(kind=4)               ::  start, finish, count_rate,count, iol
173   CHARACTER*100                 ::  name_config
174   ! Declare variables for GPU
175   REAL(fp_kind), DEVICE, DIMENSION(:,:), ALLOCATABLE    ::  omeg_d,nl_d,
        temp2_d,&
176                                 temp3_d
177   COMPLEX(fp_kind), DEVICE, DIMENSION(:,:), ALLOCATABLE :: omegoldhat_d,
        nloldhat_d,&
178                                 omeghat_d, nlhat_d, psihat_d,&
179                                 temp1_d
180   COMPLEX(fp_kind), DEVICE, DIMENSION(:), ALLOCATABLE   ::  kx_d,ky_d
181   REAL(kind=8),DEVICE, DIMENSION(:), ALLOCATABLE      ::  x_d,y_d
182
183   kersize=min(Nx,256)
184   PRINT *,'Program starting'
185       PRINT *,'Grid:',Nx,'X',Ny
186   PRINT *,'dt:',dt
187   ALLOCATE(x(1:Nx),y(1:Ny),omeg(1:Nx,1:Ny),omegexact(1:Nx,1:Ny),&
188       stat=AllocateStatus)
189   IF (AllocateStatus .ne. 0) STOP
190   PRINT *,'Allocated CPU arrays'
191   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)
        ,&
192       omegoldhat_d(1:Nx/2+1,1:Ny),nloldhat_d(1:Nx/2+1,1:Ny),&
193       omeghat_d(1:Nx/2+1,1:Ny),nl_d(1:Nx,1:Ny),&
194       nlhat_d(1:Nx/2+1,1:Ny),psihat_d(1:Nx/2+1,1:Ny),temp1_d(1:Nx/2+1,1:Ny
          ),&
195       temp2_d(1:Nx,1:Ny),temp3_d(1:Nx,1:Ny),stat=AllocateStatus)
196   IF (AllocateStatus .ne. 0) STOP
197   PRINT *,'Allocated GPU arrays'
198   CALL cufftPlan2D(pland2z,nx,ny,CUFFT_D2Z)
199   CALL cufftPlan2D(planz2d,nx,ny,CUFFT_Z2D)
200   PRINT *,'Setup FFTs'
201
202   ! setup fourier frequencies
203   !$cuf kernel do <<< *,* >>>
204   DO i=1,Nx/2+1
205     kx_d(i)= 2.0d0*pi*cmplx(0.0d0,1.0d0)*REAL(i-1,kind=fp_kind)
206   END DO
```

```
207   kx_d(1+Nx/2)=0.0d0
208   !$cuf kernel do <<< *,* >>>
209   DO i=1,Nx
210     x_d(i)=REAL(i-1,kind(0d0))/REAL(Nx,kind=fp_kind)
211   END DO
212   !$cuf kernel do <<< *,* >>>
213   DO j=1,Ny/2+1
214     ky_d(j)= 2.0d0*pi*cmplx(0.0d0,1.0d0)*REAL(j-1,kind=fp_kind)
215   END DO
216   ky_d(1+Ny/2)=0.0d0
217     !$cuf kernel do <<< *,* >>>
218   DO j = 1,Ny/2 -1
219     ky_d(j+1+Ny/2)=-ky_d(1-j+Ny/2)
220   END DO
221   !$cuf kernel do <<< *, * >>>
222   DO j=1,Ny
223     y_d(j)=REAL(j-1,kind(0d0))/REAL(Ny,kind=fp_kind)
224   END DO
225   scalemodes=1.0d0/REAL(Nx*Ny,kind=fp_kind)
226   PRINT *,'Setup grid and fourier frequencies'
227
228   !$cuf kernel do <<<  *,*  >>>
229   DO j=1,Ny
230     DO i=1,Nx
231       omeg_d(i,j)=4.0d0*pi*sin(2.0d0*pi*x_d(i))*sin(2.0d0*pi*y_d(j))!+0.01
              d0*cos(2.0d0*pi*y_d(j))
232     END DO
233   END DO
234   CALL cufftExecD2Z(pland2z,omeg_d,omeghat_d)
235
236   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
237   !get initial nonlinear term using omeghat to find psihat, u, and v!
238   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
239   !$cuf kernel do <<<  *,*  >>>
240   DO j=1,Ny
241     DO i=1,Nx/2+1
242       psihat_d(i,j)=-omeghat_d(i,j)/(kx_d(i)*kx_d(i)+ky_d(j)*ky_d(j)+0.10
              d0**14)
243     END DO
244   END DO
245
246   !$cuf kernel do <<<  *,*  >>>
247   DO j=1,Ny
248     DO i=1,Nx/2+1
249       temp1_d(i,j)=psihat_d(i,j)*ky_d(j)*scalemodes
250     END DO
251   END DO
252   CALL cufftExecZ2D(planz2d,temp1_d,temp3_d) !u
253
254   !$cuf kernel do <<<  *,*  >>>
255   DO j=1,Ny
```

```
256     DO i=1,Nx/2+1
257       temp1_d(i,j)=omeghat_d(i,j)*kx_d(i)
258     END DO
259   END DO
260   CALL cufftExecZ2D(planz2d,temp1_d,temp2_d) !omega_x
261
262   !$cuf kernel do <<<  *,*  >>>
263   DO j=1,Ny
264     DO i=1,Nx
265       nl_d(i,j)=temp3_d(i,j)*temp2_d(i,j)
266     END DO
267   END DO
268
269   !$cuf kernel do <<<  *,*  >>>
270   DO j=1,Ny
271     DO i=1,Nx/2+1
272       temp1_d(i,j)=-psihat_d(i,j)*kx_d(i)*scalemodes
273     END DO
274   END DO
275   CALL cufftExecZ2D(planz2d,temp1_d,temp3_d) !v
276
277   !$cuf kernel do <<<  *,*  >>>
278   DO j=1,Ny
279     DO i=1,Nx/2+1
280       temp1_d(i,j)=omeghat_d(i,j)*ky_d(j)
281     END DO
282   END DO
283   CALL cufftExecZ2D(planz2d,temp1_d,temp2_d) !omega_y
284
285   !combine to get full nonlinear term in real space
286   !$cuf kernel do <<<  *,*  >>>
287   DO j=1,Ny
288     DO i=1,Nx
289       nl_d(i,j)=(nl_d(i,j)+temp3_d(i,j)*temp2_d(i,j))*scalemodes
290     END DO
291   END DO
292   CALL cufftExecD2Z(pland2z,nl_d,nlhat_d)
293   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
294
295   temp2_d=omeg_d !omegacheck
296   PRINT *,'Got initial data, starting timestepping'
297   CALL system_clock(start,count_rate)
298   DO t=1,nplots
299     DO n=1,plotgap
300       chg=1.0d0
301       nloldhat_d=nlhat_d
302       omegoldhat_d=omeghat_d
303       DO WHILE (chg>tol)
304           !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
305         DO j=1,Ny
306           DO i=1,Nx/2+1
```

```
307             omeghat_d(i,j)=((dtInv+0.5d0*ReInv*(kx_d(i)*kx_d(i)+ky_d(j)*
                    ky_d(j)))&
308              *omegoldhat_d(i,j) - 0.5d0*(nloldhat_d(i,j)+nlhat_d(i,j)))
                     &
309              /(dtInv-0.5d0*ReInv*(kx_d(i)*kx_d(i)+ky_d(j)*ky_d(j)))
310           END DO
311         END DO
312           !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
313         DO j=1,Ny
314           DO i=1,Nx/2+1
315             psihat_d(i,j)=-omeghat_d(i,j)/(kx_d(i)*kx_d(i)+ky_d(j)*ky_d(j)
                    +0.10d0**14)
316           END DO
317         END DO
318         CALL cufftExecZ2D(planz2d,omeghat_d,omeg_d)
319
320         !check for convergence
321         chg=0.0d0
322           !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
323         DO j=1,Ny
324           DO i=1,Nx
325             chg=chg+(omeg_d(i,j)-temp2_d(i,j))*(omeg_d(i,j)-temp2_d(i,j))&
326             *scalemodes*scalemodes
327           END DO
328         END DO
329
330         !!!!!!!!!!!!!!!!
331         !nonlinear term!
332         !!!!!!!!!!!!!!!!
333         !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
334         DO j=1,Ny
335           DO i=1,Nx/2+1
336             temp1_d(i,j)=psihat_d(i,j)*ky_d(j)*scalemodes
337           END DO
338         END DO
339         CALL cufftExecZ2D(planz2d,temp1_d,temp3_d) !u
340
341         !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
342         DO j=1,Ny
343           DO i=1,Nx/2+1
344             temp1_d(i,j)=omeghat_d(i,j)*kx_d(i)
345           END DO
346         END DO
347         CALL cufftExecZ2D(planz2d,temp1_d,temp2_d) !omega_x
348
349         !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
350         DO j=1,Ny
351           DO i=1,Nx
352             nl_d(i,j)=temp3_d(i,j)*temp2_d(i,j)
353           END DO
354         END DO
```

```
355
356         !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
357         DO j=1,Ny
358           DO i=1,Nx/2+1
359             temp1_d(i,j)=-psihat_d(i,j)*kx_d(i)*scalemodes
360           END DO
361         END DO
362         CALL cufftExecZ2D(planz2d,temp1_d,temp3_d) !v
363
364         !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
365         DO j=1,Ny
366           DO i=1,Nx/2+1
367             temp1_d(i,j)=omeghat_d(i,j)*ky_d(j)
368           END DO
369         END DO
370         CALL cufftExecZ2D(planz2d,temp1_d,temp2_d) !omega_y
371
372         !combine to get full nonlinear term in real space
373         !$cuf kernel do(2) <<< (2,*), (kersize,1) >>>
374         DO j=1,Ny
375           DO i=1,Nx
376             nl_d(i,j)=(nl_d(i,j)+temp3_d(i,j)*temp2_d(i,j))*scalemodes
377           END DO
378         END DO
379         CALL cufftExecD2Z(pland2z,nl_d,nlhat_d)
380         !!!!!!!!!!!!!!!!
381
382         temp2_d=omeg_d !save omegacheck
383       END DO
384     END DO
385     !PRINT *, t*plotgap*dt
386   END DO
387   CALL system_clock(finish,count_rate)
388   PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),&
389       'for Time stepping'
390
391   ! Copy grid back to host
392   x=x_d
393   y=y_d
394   omeg=omeg_d
395
396   !get exact omega
397   DO j=1,Ny
398     DO i=1,Nx
399       omegexact(i,j)=4.0d0*pi*sin(2.0d0*pi*x(i))*&
400         sin(2.0d0*pi*y(j))*exp(-8.0d0*ReInv*pi**2*nplots*plotgap*dt)
401     END DO
402   END DO
403   !compute max error
404   PRINT *,'Max Error:',maxval(abs(omeg*scalemodes-omegexact))
405
```

```
406   temp=temp+1
407   write(name_config,'(a,i0,a)') 'omega',temp,'.datbin'
408   INQUIRE(iolength=iol) omeg(1,1)
409   OPEN(unit=11,FILE=name_config,form="unformatted", access="direct",recl=
          iol)
410   count = 1
411   DO j=1,Ny
412     DO i=1,Nx
413       WRITE(11,rec=count) omeg(i,j)*scalemodes
414       count=count+1
415     END DO
416   END DO
417   CLOSE(11)
418
419   CALL cufftDestroy(planz2d)
420   CALL cufftDestroy(pland2z)
421   PRINT *,'Destroyed fft plan'
422
423   DEALLOCATE(x,y,omeg,omegexact,stat=AllocateStatus)
424   IF (AllocateStatus .ne. 0) STOP
425   PRINT *,'Deallocated CPU memory'
426
427   DEALLOCATE(kx_d,ky_d,x_d,y_d,&
428       omeg_d,omegoldhat_d, nloldhat_d,omeghat_d,&
429       nl_d, nlhat_d,temp1_d,temp2_d,temp3_d,&
430       psihat_d,stat=AllocateStatus)
431   IF (AllocateStatus .ne. 0) STOP
432   PRINT *,'Deallocated GPU memory'
433   PRINT *,'Program execution complete'
434   END PROGRAM main
```

Listing A.2: An OpenACC Fortran program to solve the 2D Navier-Stokes equations.

```
1
2   !----------------------------------------------------------------------
3   !
4   ! PURPOSE
5   !
6   ! This program numerically solves the 2D incompressible Navier-Stokes
7   ! on a Square Domain [0,1]x[0,1] using pseudo-spectral methods and
8   ! Crank-Nicolson timestepping. The numerical solution is compared to
9   ! the exact Taylor-Green Vortex Solution.
10  !
11  ! AUTHORS
12  !
13  ! B. Cloutier, B.K. Muite, P. Rigge
14  ! 4 June 2012
15  !
16  ! Periodic free-slip boundary conditions and Initial conditions:
17  ! u(x,y,0)=sin(2*pi*x)cos(2*pi*y)
```

```
18  ! v(x,y,0)=-cos(2*pi*x)sin(2*pi*y)
19  ! Analytical Solution (subscript denote derivatives):
20  ! u(x,y,t)=sin(2*pi*x)cos(2*pi*y)exp(-8*pi^2*t/Re)
21  ! v(x,y,t)=-cos(2*pi*x)sin(2*pi*y)exp(-8*pi^2*t/Re)
22  !   u_y(x,y,t)=-2*pi*sin(2*pi*x)sin(2*pi*y)exp(-8*pi^2*t/Re)
23  ! v_x(x,y,t)=2*pi*sin(2*pi*x)sin(2*pi*y)exp(-8*pi^2*t/Re)
24  ! omega=v_x-u_y
25  !
26  ! .. Parameters ..
27  !  Nx       = number of modes in x - power of 2 for FFT
28  !  Ny       = number of modes in y - power of 2 for FFT
29  !  nplots     = number of plots produced
30  !  plotgap      = number of timesteps inbetween plots
31  !  Re         = dimensionless Renold's number
32  !  ReInv      = 1/Re for optimization
33  !  dt       = timestep size
34  !  dtInv      = 1/dt for optimization
35  !  tol        = determines when convergences is reached
36  !  scalemodes   = 1/(Nx*Ny), scaling after preforming FFTs
37  ! .. Scalars ..
38  !  i        = loop counter in x direction
39  !  j        = loop counter in y direction
40  !  n        = loop counter for timesteps direction
41  !  allocatestatus = error indicator during allocation
42  !  time       = times at which data is saved
43  !  chg        = error at each iteration
44  ! .. Arrays ..
45  !  omeg       = vorticity in real space
46  !  omeghat      = 2D Fourier transform of vorticity
47  !           at next iterate
48  !  omegoldhat   = 2D Fourier transform of vorticity at previous
49  !           iterate
50  !  nl       = nonlinear term
51  !  nlhat      = nonlinear term in Fourier space
52  !  nloldhat     = nonlinear term in Fourier space
53  !           at previous iterate
54  !  omegexact    = taylor-green vorticity at
55  !           at final step
56  !  psihat     = 2D Fourier transform of streamfunction
57  !           at next iteration
58  !  temp1/temp2/temp3= reusable complex/real space used for
59  !           calculations. This reduces number of
60  !           arrays stored.
61  ! .. Vectors ..
62  !  kx       = fourier frequencies in x direction
63  !  ky       = fourier frequencies in y direction
64  !  x        = x locations
65  !  y        = y locations
66  !  name_config    = array to store filename for data to be saved
67  ! REFERENCES
68  !
```

```
69  ! ACKNOWLEDGEMENTS
70  !
71  ! ACCURACY
72  !
73  ! ERROR INDICATORS AND WARNINGS
74  !
75  ! FURTHER COMMENTS
76  ! Check that the initial iterate is consistent with the
77  ! boundary conditions for the domain specified
78  !-----------------------------------------------------------------------
79  ! External libraries required
80  !       Cuda FFT
81  !       OpenACC
82  !       FFTW3           -- Fastest Fourier Transform in the West
83  !                       (http://www.fftw.org/)
84  !       OpenMP
85
86  module precision
87  ! Precision control
88
89  integer, parameter, public :: Single = kind(0.0) ! Single precision
90  integer, parameter, public :: Double = kind(0.0d0) ! Double precision
91
92  integer, parameter, public :: fp_kind = Double
93  !integer, parameter, public :: fp_kind = Single
94
95  end module precision
96
97  module cufft
98
99  integer, public :: CUFFT_FORWARD = -1
100 integer, public :: CUFFT_INVERSE = 1
101 integer, public :: CUFFT_R2C = Z'2a' ! Real to Complex (interleaved)
102 integer, public :: CUFFT_C2R = Z'2c' ! Complex (interleaved) to Real
103 integer, public :: CUFFT_C2C = Z'29' ! Complex to Complex, interleaved
104 integer, public :: CUFFT_D2Z = Z'6a' ! Double to Double-Complex
105 integer, public :: CUFFT_Z2D = Z'6c' ! Double-Complex to Double
106 integer, public :: CUFFT_Z2Z = Z'69' ! Double-Complex to Double-Complex
107
108 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
109 !
110 ! cufftPlan2d(cufftHandle *plan, int nx,int ny, cufftType type)
111 !
112 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
113
114 interface cufftPlan2d
115 subroutine cufftPlan2d(plan, nx, ny, type) bind(C,name='cufftPlan2d')
116 use iso_c_binding
117 integer(c_int):: plan
118 integer(c_int),value:: nx, ny, type
119 end subroutine cufftPlan2d
```

```
120  end interface cufftPlan2d
121
122  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
123  !
124  ! cufftDestroy(cufftHandle plan)
125  !
126  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
127
128  interface cufftDestroy
129  subroutine cufftDestroy(plan) bind(C,name='cufftDestroy')
130  use iso_c_binding
131  integer(c_int),value:: plan
132  end subroutine cufftDestroy
133  end interface cufftDestroy
134
135  interface cufftExecD2Z
136    subroutine cufftExecD2Z(plan, idata, odata) &
137          & bind(C,name='cufftExecD2Z')
138       use iso_c_binding
139       use precision
140       integer(c_int),  value  :: plan
141       real(fp_kind),   device :: idata(1:nx,1:ny)
142       complex(fp_kind),device :: odata(1:nx/2+1,1:ny)
143    end subroutine cufftExecD2Z
144  end interface cufftExecD2Z
145
146  interface cufftExecZ2D
147    subroutine cufftExecZ2D(plan, idata, odata) &
148          & bind(C,name='cufftExecZ2D')
149       use iso_c_binding
150       use precision
151       integer(c_int),value:: plan
152       complex(fp_kind),device:: idata(1:nx/2+1,1:ny)
153       real(fp_kind),device :: odata(1:nx,1:ny)
154    end subroutine cufftExecZ2D
155  end interface cufftExecZ2D
156  end module cufft
157
158
159  PROGRAM main
160  USE precision
161  USE cufft
162  USE openacc
163
164  IMPLICIT NONE
165    INTEGER(kind=4), PARAMETER      ::  Nx=512
166  INTEGER(kind=4), PARAMETER      ::  Ny=512
167  REAL(kind=8), PARAMETER     ::  dt=0.000125d0
168  REAL(kind=8), PARAMETER     ::  dtInv=1.0d0/REAL(dt,kind(0d0))
169  REAL(kind=8), PARAMETER &
170    ::  pi=3.14159265358979323846264338327950288419716939937510d0
```

```
171   REAL(kind=8), PARAMETER      ::  Re=1.0d0
172   REAL(kind=8), PARAMETER      ::  ReInv=1.0d0/REAL(Re,kind(0d0))
173   REAL(kind=8), PARAMETER      ::  tol=0.1d0**10
174   REAL(kind=8)           ::  scalemodes
175   REAL(kind=8)           ::  chg
176   INTEGER(kind=4), PARAMETER     ::  nplots=1, plotgap=20
177   COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE     ::  kx,ky
178   REAL(kind=8), DIMENSION(:), ALLOCATABLE      ::  x,y,time
179   REAL(kind=8), DIMENSION(:,:), ALLOCATABLE    :: omeg,nl, temp2, temp3,
         omegexact
180   COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE  :: omegoldhat, nloldhat,&
181                              omeghat,nlhat, psihat,temp1
182   INTEGER(kind=4)               ::  i,j,n,t, allocatestatus
183   INTEGER(kind=4)               ::  pland2z,planz2d
184   INTEGER(kind=4)               ::  count, iol
185   CHARACTER*100                ::  name_config
186   INTEGER(kind=4)               ::  start, finish, count_rate
187   INTEGER(kind=4)               ::  ierr,vecsize,gangsize
188   INTEGER(kind=8)               ::  planfxy,planbxy
189
190   vecsize=32
191   gangsize=16
192     PRINT *,'Grid:',Nx,'X',Ny
193   PRINT *,'dt:',dt
194   ALLOCATE(time(1:nplots+1),kx(1:Nx),ky(1:Ny),x(1:Nx),y(1:Ny),&
195       omeg(1:Nx,1:Ny),omegoldhat(1:Nx/2+1,1:Ny),&
196       nloldhat(1:Nx/2+1,1:Ny),temp3(1:Nx,1:Ny),omeghat(1:Nx/2+1,1:Ny),&
197       nl(1:Nx,1:Ny),nlhat(1:Nx/2+1,1:Ny), psihat(1:Nx/2+1,1:Ny),&
198       temp1(1:Nx/2+1,1:Ny),omegexact(1:Nx,1:Ny),temp2(1:Nx,1:Ny),&
199       stat=AllocateStatus)
200   IF (AllocateStatus .ne. 0) STOP
201   PRINT *,'allocated space'
202
203   CALL cufftPlan2D(pland2z,nx,ny,CUFFT_D2Z)
204   CALL cufftPlan2D(planz2d,nx,ny,CUFFT_Z2D)
205
206   PRINT *,'Setup 2D FFTs'
207
208   ! setup fourier frequencies in x-direction
209   !$acc data copy(kx,ky,x,y,time,temp3,omeg,nl,temp1,temp2,omegoldhat,
         nloldhat,omeghat,nlhat,psihat)
210   PRINT *, 'Copied arrays over to device'
211   !$acc kernels loop
212   DO i=1,Nx/2+1
213     kx(i)= 2.0d0*pi*cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))
214   END DO
215   !$acc end kernels
216   kx(1+Nx/2)=0.0d0
217   !$acc kernels loop
218   DO i = 1,Nx/2 -1
219     kx(i+1+Nx/2)=-kx(1-i+Nx/2)
```

```
220   END DO
221   !$acc end kernels
222   !$acc kernels loop
223   DO i=1,Nx
224     x(i)=REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0))
225   END DO
226   !$acc end kernels
227   ! setup fourier frequencies in y-direction
228   !$acc kernels loop
229   DO j=1,Ny/2+1
230     ky(j)= 2.0d0*pi*cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))
231   END DO
232   !$acc end kernels
233   ky(1+Ny/2)=0.0d0
234   !$acc kernels loop
235   DO j = 1,Ny/2 -1
236     ky(j+1+Ny/2)=-ky(1-j+Ny/2)
237   END DO
238   !$acc end kernels
239   !$acc kernels loop
240   DO j=1,Ny
241     y(j)=REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0))
242   END DO
243   !$acc end kernels
244   scalemodes=1.0d0/REAL(Nx*Ny,kind(0d0))
245   PRINT *,'Setup grid and fourier frequencies'
246
247   !initial data
248   !$acc kernels loop
249   DO j=1,Ny
250     DO i=1,NX
251       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))
252     END DO
253   END DO
254   !$acc end kernels
255   !\hat{\omega^{n,k}}
256   CALL cufftExecD2Z(pland2z,omeg,omeghat)
257
258   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
259   !get initial nonlinear term using omeghat, psihat, u, and v!
260   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
261   !\hat{\psi^{n+1,k+1}}
262   !$acc kernels loop gang(gangsize), vector(vecsize)
263   DO j=1,Ny
264     DO i=1,Nx/2+1
265       psihat(i,j)=-omeghat(i,j)/(kx(i)*kx(i)+ky(j)*ky(j) + 0.1d0**14)
266     END DO
267   END DO
268   !$acc end kernels
269   !\omega^{n+1,k+1}
```

```
270   CALL cufftExecZ2D(planz2d,omeghat,omeg)
271
272   !get \hat{\psi_y^{n+1,k+1}} used to get u, NOTE: u=\psi_y
273   !$acc kernels loop gang(gangsize), vector(vecsize)
274   DO j=1,Ny
275     DO i=1,Nx/2+1
276       temp1(i,j)=psihat(i,j)*ky(j)*scalemodes
277     END DO
278   END DO
279   !$acc end kernels
280   CALL cufftExecZ2D(planz2d,temp1,temp3) !u
281
282   ! \hat{\omega_x^{n,k}}
283   !$acc kernels loop
284   DO j=1,Ny
285     DO i=1,Nx/2+1
286       temp1(i,j)=omeghat(i,j)*kx(i)
287     END DO
288   END DO
289   !$acc end kernels
290   ! \omega_x^{n,k}
291   CALL cufftExecZ2D(planz2d,temp1,temp2)
292
293   ! first part nonlinear term in real space
294   !$acc kernels loop
295   DO j=1,Ny
296     DO i=1,Nx
297       nl(i,j)=temp3(i,j)*temp2(i,j)
298     END DO
299   END DO
300   !$acc end kernels
301
302   !get \hat{\psi_x^{n+1,k+1}} used to get v, NOTE: v=-\psi_x
303   !$acc kernels loop gang(gangsize), vector(vecsize)
304   DO j=1,Ny
305     DO i=1,Nx/2+1
306       temp1(i,j)=-psihat(i,j)*kx(i)*scalemodes
307     END DO
308   END DO
309   !$acc end kernels
310   CALL cufftExecZ2D(planz2d,temp1,temp3) !v
311
312   ! \hat{\omega_y^{n,k}}
313   !$acc kernels loop
314   DO j=1,Ny
315     DO i=1,Nx/2+1
316       temp1(i,j)=omeghat(i,j)*ky(j)
317     END DO
318   END DO
319   !$acc end kernels
320   ! \omega_y^{n,k}
```

```
321   CALL cufftExecZ2D(planz2d,temp1,temp2)
322
323   ! get the rest of nonlinear term in real space
324   !$acc kernels loop
325   DO j=1,Ny
326     DO i=1,Nx
327       nl(i,j)=(nl(i,j)+temp3(i,j)*temp2(i,j))*scalemodes
328     END DO
329   END DO
330   !$acc end kernels
331   ! transform nonlinear term into fourier space
332   CALL cufftExecD2Z(pland2z,nl,nlhat)
333   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
334
335   !$acc kernels loop
336   DO j=1,Ny
337     DO i=1,Nx
338       temp2(i,j)=omeg(i,j)
339     END DO
340   END DO
341   !$acc end kernels
342
343   PRINT *,'Got initial data, starting timestepping'
344   time(1)=0.0d0
345   CALL system_clock(start,count_rate)
346   DO t=1,nplots
347     DO n=1,plotgap
348       chg=1.0d0
349       ! save old values(__^{n,k} terms in equation)
350       !$acc kernels loop gang(gangsize), vector(vecsize)
351       DO j=1,Ny
352         DO i=1,Nx/2+1
353           nloldhat(i,j)=nlhat(i,j)
354         END DO
355       END DO
356       !$acc end kernels
357       !$acc kernels loop gang(gangsize), vector(vecsize)
358       DO j=1,Ny
359         DO i=1,Nx/2+1
360           omegoldhat(i,j)=omeghat(i,j)
361         END DO
362       END DO
363       !$acc end kernels
364       DO WHILE (chg>tol)
365         !Crank-Nicolson timestepping to get \hat{\omega^{n+1,k+1}}
366         !$acc kernels loop gang(gangsize), vector(vecsize)
367         DO j=1,Ny
368           DO i=1,Nx/2+1
369             omeghat(i,j)=( (dtInv+0.5d0*ReInv*(kx(i)*kx(i)+ky(j)*ky(j)))&
370                 *omegoldhat(i,j) - 0.5d0*(nloldhat(i,j)+nlhat(i,j)))/&
371                 (dtInv-0.5d0*ReInv*(kx(i)*kx(i)+ky(j)*ky(j)))
```

```
372           END DO
373         END DO
374         !$acc end kernels
375         CALL cufftExecZ2D(planz2d,omeghat,omeg)
376
377         ! check for convergence
378         chg=0.0d0
379         !$acc kernels loop gang(gangsize), vector(vecsize)
380         DO j=1,Ny
381           DO i=1,Nx
382             chg=chg+(omeg(i,j)-temp2(i,j))*(omeg(i,j)-temp2(i,j))&
383             *scalemodes*scalemodes
384           END DO
385         END DO
386         !$acc end kernels
387
388         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
389         !get nonlinear term using omeghat, psihat, u, and v!
390         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
391         !\hat{\psi^{n+1,k+1}}
392         !$acc kernels loop gang(gangsize), vector(vecsize)
393         DO j=1,Ny
394           DO i=1,Nx/2+1
395             psihat(i,j)=-omeghat(i,j)/(kx(i)*kx(i)+ky(j)*ky(j) + 0.1d0
                  **14)
396           END DO
397         END DO
398         !$acc end kernels
399         !\omega^{n+1,k+1}
400         CALL cufftExecZ2D(planz2d,omeghat,omeg)
401
402         !get \hat{\psi_y^{n+1,k+1}} used to get u, NOTE: u=\psi_y
403         !$acc kernels loop gang(gangsize), vector(vecsize)
404         DO j=1,Ny
405           DO i=1,Nx/2+1
406             temp1(i,j)=psihat(i,j)*ky(j)*scalemodes
407           END DO
408         END DO
409         !$acc end kernels
410         CALL cufftExecZ2D(planz2d,temp1,temp3) !u
411
412         ! \hat{\omega_x^{n,k}}
413         !$acc kernels loop
414         DO j=1,Ny
415           DO i=1,Nx/2+1
416             temp1(i,j)=omeghat(i,j)*kx(i)
417           END DO
418         END DO
419         !$acc end kernels
420         ! \omega_x^{n,k}
421         CALL cufftExecZ2D(planz2d,temp1,temp2)
```

```
422
423         ! first part nonlinear term in real space
424         !$acc kernels loop
425         DO j=1,Ny
426           DO i=1,Nx
427             nl(i,j)=temp3(i,j)*temp2(i,j)
428           END DO
429         END DO
430         !$acc end kernels
431
432         !get \hat{\psi_x^{n+1,k+1}} used to get v, NOTE: v=-\psi_x
433         !$acc kernels loop gang(gangsize), vector(vecsize)
434         DO j=1,Ny
435           DO i=1,Nx/2+1
436             temp1(i,j)=-psihat(i,j)*kx(i)*scalemodes
437           END DO
438         END DO
439         !$acc end kernels
440         CALL cufftExecZ2D(planz2d,temp1,temp3)
441
442         ! \hat{\omega_y^{n,k}}
443         !$acc kernels loop
444         DO j=1,Ny
445           DO i=1,Nx/2+1
446             temp1(i,j)=omeghat(i,j)*ky(j)
447           END DO
448         END DO
449         !$acc end kernels
450         ! \omega_y^{n,k}
451         CALL cufftExecZ2D(planz2d,temp1,temp2)
452
453         ! get the rest of nonlinear term in real space
454         !$acc kernels loop
455         DO j=1,Ny
456           DO i=1,Nx
457             nl(i,j)=(nl(i,j)+temp3(i,j)*temp2(i,j))*scalemodes
458           END DO
459         END DO
460         !$acc end kernels
461         ! transform nonlinear term into fourier space
462         CALL cufftExecD2Z(pland2z,nl,nlhat)
463         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
464
465         !\omega^{n+1,k+1} is saved for next iteration
466         !$acc kernels loop gang(gangsize), vector(vecsize)
467         DO j=1,Ny
468           DO i=1,Nx
469             temp2(i,j)=omeg(i,j)
470           END DO
471         END DO
472         !$acc end kernels
```

```
473       END DO
474     END DO
475     time(t+1)=time(t)+dt*plotgap
476     !PRINT *, time(t+1)
477   END DO
478   CALL system_clock(finish,count_rate)
479   PRINT*,'Program took ',REAL(finish-start)/REAL(count_rate),&
480       'for Time stepping'
481
482   !get exact omega
483   !$acc kernels loop gang(gangsize), vector(vecsize)
484   DO j=1,Ny
485     DO i=1,Nx
486       omegexact(i,j)=4.0d0*pi*sin(2.0d0*pi*x(i))*&
487         sin(2.0d0*pi*y(j))*exp(-8.0d0*ReInv*pi**2*nplots*plotgap*dt)
488     END DO
489   END DO
490   !$acc end kernels
491   !$acc end data
492
493   !compute max error
494   PRINT *,'Max Error:',maxval(abs(omeg*scalemodes-omegexact))
495
496   !!!!!!!!!!!!!!!!!!!!!!!
497   !copy over data to disk!
498   !!!!!!!!!!!!!!!!!!!!!!!
499   write(name_config,'(a,i0,a)') 'omega',1,'.datbin'
500   INQUIRE(iolength=iol) omeg(1,1)
501   OPEN(unit=11,FILE=name_config,form="unformatted", access="direct",recl=
        iol)
502   count = 1
503   DO j=1,Ny
504     DO i=1,Nx
505       WRITE(11,rec=count) omeg(i,j)*scalemodes
506       count=count+1
507     END DO
508   END DO
509   CLOSE(11)
510
511   name_config = 'time.dat'
512   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
513   REWIND(11)
514   DO i=1,Nplots+1
515     WRITE(11,*) time(i)
516   END DO
517   CLOSE(11)
518
519   name_config = 'xcoord.dat'
520   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
521   REWIND(11)
522   DO i=1,Nx
```

```
523       WRITE(11,*) x(i)
524     END DO
525     CLOSE(11)
526
527     name_config = 'ycoord.dat'
528     OPEN(unit=11,FILE=name_config,status="UNKNOWN")
529     REWIND(11)
530     DO j=1,Ny
531       WRITE(11,*) y(j)
532     END DO
533     CLOSE(11)
534     !!!!!!!!!!!!!!!!!!!!!!!
535
536     CALL cufftDestroy(pland2z)
537     CALL cufftDestroy(planz2d)
538
539     DEALLOCATE(time,temp1,temp2,temp3,kx,ky,x,y,&
540         omeg,omegoldhat,omegexact, nloldhat,&
541         omeghat,nl, nlhat, psihat,&
542         stat=AllocateStatus)
543     IF (AllocateStatus .ne. 0) STOP
544     PRINT *,'Program execution complete'
545
546     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.

```
1   !-----------------------------------------------------------------------
2   !
3   ! PURPOSE
4   !
5   ! This program solves nonlinear Schrodinger equation in 2 dimensions
6   ! i*u_t+Es*|u|^2u+u_{xx}+u_{yy}=0
7   ! using a second order time spectral splitting scheme
8   !
9   ! The boundary conditions are u(x=0,y)=u(x=2*L*\pi,y)
10  ! and u(x,y=0)=u(x,y=2*L*\pi)
11  ! The initial condition is u=exp(-x^2-y^2)
12  !
13  ! AUTHORS
14  !
15  ! B. Cloutier, B.K. Muite, P. Rigge
16  ! 4 June 2012
17  !
18  ! .. Parameters ..
```

```
19  !  Nx       = number of modes in x - power of 2 for FFT
20  !  Ny       = number of modes in y - power of 2 for FFT
21  !  Nt       = number of timesteps to take
22  !  Tmax       = maximum simulation time
23  !  plotgap      = number of timesteps between plots
24  !  pi = 3.14159265358979323846264338327950288419716939937510d0
25  !  L        = width of box
26  !  ES       = +1 for focusing and -1 for defocusing
27  ! .. Scalars ..
28  !  i        = loop counter in x direction
29  !  j        = loop counter in y direction
30  !  n        = loop counter for timesteps direction
31  !  allocatestatus = error indicator during allocation
32  !  start      = variable to record start time of program
33  !  finish     = variable to record end time of program
34  !  count_rate   = variable for clock count rate
35  !  plan       = fft plan
36  !  dt       = timestep
37  !  InMass     = initial mass
38  !  FiMass     = final mass
39  !  InEner     = initial energy
40  !  FiEner     = final energy
41  ! .. Arrays ..
42  !  u        = approximate solution
43  !  v        = Fourier transform of approximate solution
44  !  u_d        = approximate solution on device
45  !  v_d        = Fourier transform of approximate solution on device
46  !  temp1_d      = temporary array used to find mass and energy
47  !  temp2_d      = temporary array used to find mass and energy
48  ! .. Vectors ..
49  !  kx       = fourier frequencies in x direction
50  !  ky       = fourier frequencies in y direction
51  !  x        = x locations
52  !  y        = y locations
53  !  time       = times at which save data
54  !  name_config    = array to store filename for data to be saved
55  !
56  ! REFERENCES
57  !
58  ! ACKNOWLEDGEMENTS
59  !
60  ! This program is based on example code to demonstrate usage of Fortran
       and
61  ! CUDA FFT routines taken from
62  ! http://cudamusing.blogspot.com/2010/05/calling-cufft-from-cuda-fortran
       .html
63  !
64  ! and
65  !
66  ! http://cudamusing.blogspot.com/search?q=cublas
67  !
```

```
68  ! ACCURACY
69  !
70  ! ERROR INDICATORS AND WARNINGS
71  !
72  ! FURTHER COMMENTS
73  ! Check that the initial iterate is consistent with the
74  ! boundary conditions for the domain specified
75  !-----------------------------------------------------------------------
76  ! External routines required
77  !
78  ! External libraries required
79  ! cufft  -- Cuda FFT library
80  !
81
82  !
83  ! Define the INTERFACE to the NVIDIA CUFFT routines
84  !
85
86  module precision
87  ! Precision control
88
89  integer, parameter, public :: Single = kind(0.0) ! Single precision
90  integer, parameter, public :: Double = kind(0.0d0) ! Double precision
91
92  integer, parameter, public :: fp_kind = Double
93  !integer, parameter, public :: fp_kind = Single
94
95  end module precision
96
97  module cufft
98
99  integer, public :: CUFFT_FORWARD = -1
100 integer, public :: CUFFT_INVERSE = 1
101 integer, public :: CUFFT_R2C = Z'2a' ! Real to Complex (interleaved)
102 integer, public :: CUFFT_C2R = Z'2c' ! Complex (interleaved) to Real
103 integer, public :: CUFFT_C2C = Z'29' ! Complex to Complex, interleaved
104 integer, public :: CUFFT_D2Z = Z'6a' ! Double to Double-Complex
105 integer, public :: CUFFT_Z2D = Z'6c' ! Double-Complex to Double
106 integer, public :: CUFFT_Z2Z = Z'69' ! Double-Complex to Double-Complex
107
108 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
109 !
110 ! cufftPlan2d(cufftHandle *plan, int nx,int ny, cufftType type)
111 !
112 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
113
114 interface cufftPlan2d
115 subroutine cufftPlan2d(plan, nx, ny, type) bind(C,name='cufftPlan2d')
116 use iso_c_binding
117 integer(c_int):: plan
118 integer(c_int),value:: nx, ny, type
```

```
119   end subroutine cufftPlan2d
120   end interface cufftPlan2d
121
122   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
123   !
124   ! cufftDestroy(cufftHandle plan)
125   !
126   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
127
128   interface cufftDestroy
129   subroutine cufftDestroy(plan) bind(C,name='cufftDestroy')
130   use iso_c_binding
131   integer(c_int),value:: plan
132   end subroutine cufftDestroy
133   end interface cufftDestroy
134
135   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
136   !
137   ! cufftExecZ2Z(cufftHandle plan,
138   ! cufftDoubleComplex *idata,
139   ! cufftDoubleComplex *odata,
140   ! int direction;
141   !
142   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
143   interface cufftExecZ2Z
144   subroutine cufftExecZ2Z(plan, idata, odata, direction) &
145   & bind(C,name='cufftExecZ2Z')
146   use iso_c_binding
147   use precision
148   integer(c_int),value:: direction
149   integer(c_int),value:: plan
150   complex(fp_kind),device,dimension(1:nx,1:ny):: idata,odata
151   end subroutine cufftExecZ2Z
152   end interface cufftExecZ2Z
153
154   end module cufft
155
156   PROGRAM main
157   use precision
158   use cufft
159   ! Declare host variables and scalars
160   IMPLICIT NONE
161   INTEGER(kind=4), PARAMETER            :: Nx=1024
162   INTEGER(kind=4), PARAMETER            :: Ny=1024
163   INTEGER(kind=4), PARAMETER            :: Nt=20
164   INTEGER(kind=4), PARAMETER            :: plotgap=5
165   REAL(fp_kind), PARAMETER    &
166     ::  pi=3.14159265358979323846264338327950288419716939937510d0
167   REAL(fp_kind), PARAMETER            :: Lx=5.0d0
168   REAL(fp_kind), PARAMETER            :: Ly=5.0d0
169   REAL(fp_kind), PARAMETER            :: Es=1.0d0
```

```
170   REAL(fp_kind)                  :: dt=0.10d0**5
171   REAL(fp_kind)                  :: scalemodes
172   COMPLEX(fp_kind)                 :: InMass,FiMass,InEner,FiEner
173   REAL(fp_kind),  DIMENSION(:), ALLOCATABLE    :: x,y
174   COMPLEX(fp_kind),  DIMENSION(:,:), ALLOCATABLE  :: u
175   REAL(fp_kind), DIMENSION(:), ALLOCATABLE    :: time
176   INTEGER(kind=4) :: i,j,k,n,modes,AllocateStatus,plan, kersize
177   INTEGER(kind=4) :: start, finish, count_rate
178     CHARACTER*100 :: name_config
179   ! Declare variables for GPU
180   COMPLEX(fp_kind), DEVICE, DIMENSION(:), ALLOCATABLE :: kx_d,ky_d
181   REAL(fp_kind), DEVICE, DIMENSION(:), ALLOCATABLE  :: x_d,y_d
182   COMPLEX(fp_kind), DEVICE, DIMENSION(:,:), ALLOCATABLE :: u_d,v_d,temp1_d
        ,temp2_d
183   REAL(fp_kind),DEVICE,DIMENSION(:), ALLOCATABLE    :: time_d
184
185   ! start timing
186   PRINT *,'Program starting'
187   PRINT *,'Grid: ',Nx,'X',Ny
188     ! allocate arrays on the host
189   ALLOCATE(x(1:Nx),y(1:Ny),u(1:Nx,1:Ny),time(1:Nt+1),&
190     stat=AllocateStatus)
191   IF (allocatestatus .ne. 0) STOP
192   PRINT *,'Allocated CPU arrays'
193
194   ! allocate arrays on the device
195   ALLOCATE(kx_d(1:Nx),ky_d(1:Nx),x_d(1:Nx),y_d(1:Nx),&
196       u_d(1:Nx,1:Ny),v_d(1:Nx,1:Ny),time_d(1:Nt+1),&
197       temp1_d(1:Nx,1:Ny),temp2_d(1:Nx,1:Ny),stat=AllocateStatus)
198   IF (allocatestatus .ne. 0) STOP
199   PRINT *,'Allocated GPU arrays'
200
201   kersize=min(Nx,256)
202   ! set up ffts
203   CALL cufftPlan2D(plan,nx,ny,CUFFT_Z2Z)
204   PRINT *,'Setup FFTs'
205     ! setup fourier frequencies
206   !$cuf kernel do <<< *, * >>>
207   DO i=1,1+Nx/2
208     kx_d(i)= cmplx(0.0d0,1.0d0)*(i-1.0d0)/Lx
209   END DO
210   kx_d(1+Nx/2)=0.0d0
211   !$cuf kernel do <<< *, * >>>
212   DO i = 1,Nx/2 -1
213     kx_d(i+1+Nx/2)=-kx_d(1-i+Nx/2)
214   END DO
215   !$cuf kernel do <<< *, * >>>
216   DO i=1,Nx
217     x_d(i)=(-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind=fp_kind))*pi*
          Lx
218   END DO
```

```
219   !$cuf kernel do <<< *, * >>>
220   DO j=1,1+Ny/2
221     ky_d(j)= cmplx(0.0d0,1.0d0)*(j-1.0d0)/Ly
222   END DO
223   ky_d(1+Ny/2)=0.0d0
224   !$cuf kernel do <<< *, * >>>
225   DO j = 1,Ny/2 -1
226     ky_d(j+1+Ny/2)=-ky_d(1-j+Ny/2)
227   END DO
228   !$cuf kernel do <<< *, * >>>
229   DO j=1,Ny
230     y_d(j)=(-1.0d0 + 2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind=fp_kind))*pi*
          Ly
231   END DO
232   scalemodes=1.0d0/REAL(Nx*Ny,kind=fp_kind)
233   PRINT *,'Setup grid and fourier frequencies'
234
235   !$cuf kernel do <<<  *,*  >>>
236   DO j=1,Ny
237     DO i=1,Nx
238       u_d(i,j)=exp(-1.0d0*(x_d(i)**2+y_d(j)**2))
239     END DO
240   END DO
241   ! transform initial data
242   CALL cufftExecZ2Z(plan,u_d,v_d,CUFFT_FORWARD)
243
244   PRINT *,'Got initial data'
245   ! get initial mass
246   !$cuf kernel do <<<  *,* >>>
247   DO j=1,Ny
248     DO i=1,Nx
249       temp1_d(i,j)=abs(u_d(i,j))**2
250     END DO
251   END DO
252   ! Use FFT to get initial mass
253   CALL cufftExecZ2Z(plan,temp1_d,temp2_d,CUFFT_FORWARD)
254   InMass=temp2_d(1,1)
255   ! Get initial energy
256   !$cuf kernel do <<<  *,* >>>
257   DO j=1,Ny
258     DO i=1,Nx
259       temp1_d(i,j)=-ES*0.25d0*abs(u_d(i,j))**4
260     END DO
261   END DO
262   ! Use FFT to find mean
263   CALL cufftExecZ2Z(plan,temp1_d,temp2_d,CUFFT_FORWARD)
264   InEner=temp2_d(1,1)
265   !$cuf kernel do <<< *,* >>>
266   DO j=1,Ny
267     DO i=1,Nx
268       temp2_d(i,j)=kx_d(i)*v_d(i,j)*scalemodes
```

```
269       END DO
270     END DO
271     CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_INVERSE)
272     !$cuf kernel do <<< *,* >>>
273     DO j=1,Ny
274       DO i=1,Nx
275         temp2_d(i,j)=0.5d0*abs(temp1_d(i,j))**2
276       END DO
277     END DO
278     ! Use FFT to find mean
279     CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_FORWARD)
280     InEner=InEner+temp1_d(1,1)
281     !$cuf kernel do <<<  *,*  >>>
282     DO j=1,Ny
283       DO i=1,Nx
284         temp2_d(i,j)=ky_d(j)*v_d(i,j)*scalemodes
285       END DO
286     END DO
287     CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_INVERSE)
288     !$cuf kernel do <<< *,* >>>
289     DO j=1,Ny
290       DO i=1,Nx
291         temp2_d(i,j)=0.5d0*abs(temp1_d(i,j))**2
292       END DO
293     END DO
294     ! Use FFT to find mean
295     CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_FORWARD)
296     InEner=InEner+temp1_d(1,1)
297
298     ! start timing
299     CALL system_clock(start,count_rate)
300     ! Do first half time step
301     CALL cufftExecZ2Z(plan,u_d,v_d,CUFFT_FORWARD)
302     !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
303     DO j=1,Ny
304       DO i=1,Nx
305           v_d(i,j)=exp(dt*( kx_d(i)*kx_d(i) + ky_d(j)*ky_d(j) )&
306             *cmplx(0.0d0,0.50d0) )*v_d(i,j)
307       END DO
308     END DO
309
310     PRINT *,'Starting timestepping'
311     time(1)=0.0d0
312     DO n=1,Nt
313       time_d(n+1)=n*dt
314       CALL cufftExecZ2Z(plan,v_d,u_d,CUFFT_INVERSE)
315       !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
316       DO j=1,Ny
317         DO i=1,Nx
318           v_d(i,j)=Es*u_d(i,j)*conjg(u_d(i,j))*scalemodes**2
319         END DO
```

```
320     END DO
321     !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
322     DO j=1,Ny
323       DO i=1,Nx
324         u_d(i,j)=exp(cmplx(0.0d0,-1.0d0)*dt*v_d(i,j))&
325             *u_d(i,j)*scalemodes
326       END DO
327     END DO
328
329     CALL cufftExecZ2Z(plan,u_d,v_d,CUFFT_FORWARD)
330     !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
331     DO j=1,Ny
332       DO i=1,Nx
333         v_d(i,j)=exp(dt*(kx_d(i)*kx_d(i) + ky_d(j)*ky_d(j))&
334           *cmplx(0.0d0,1.0d0))*v_d(i,j)
335       END DO
336     END DO
337
338   END DO
339
340   ! transform back final data and do another half time step
341
342   CALL cufftExecZ2Z(plan,v_d,u_d,CUFFT_INVERSE)
343   !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
344   DO j=1,Ny
345     DO i=1,Nx
346       v_d(i,j)=Es*u_d(i,j)*conjg(u_d(i,j))*scalemodes**2
347     END DO
348   END DO
349   !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
350   DO j=1,Ny
351     DO i=1,Nx
352       u_d(i,j)=exp(cmplx(0.0d0,-1.0d0)*dt*v_d(i,j))&
353           *u_d(i,j)*scalemodes
354     END DO
355   END DO
356
357   CALL cufftExecZ2Z(plan,u_d,v_d,CUFFT_FORWARD)
358   !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
359   DO j=1,Ny
360     DO i=1,Nx
361       v_d(i,j)=exp(dt*(kx_d(i)*kx_d(i) + ky_d(j)*ky_d(j))&
362           *cmplx(0.0d0,0.50d0))*v_d(i,j)
363     END DO
364   END DO
365   CALL cufftExecZ2Z(plan,v_d,u_d,CUFFT_INVERSE)
366   ! normalize
367   !$cuf kernel do(2) <<< (1,*),(kersize,1) >>>
368   DO j=1,Ny
369     DO i=1,Nx
370       u_d(i,j)=u_d(i,j)*scalemodes
```

```
371     END DO
372   END DO
373
374   CALL system_clock(finish,count_rate)
375   PRINT*,'Program took ',&
376     REAL(finish-start,kind(0d0))/REAL(count_rate,kind(0d0)),'s for
          execution'
377
378   PRINT *,'Finished time stepping'
379
380   ! calculate final mass
381   !$cuf kernel do <<<  *,*  >>>
382   DO j=1,Ny
383     DO i=1,Nx
384       temp1_d(i,j)=abs(u_d(i,j))**2
385     END DO
386   END DO
387   ! Use FFT to get initial mass
388   CALL cufftExecZ2Z(plan,temp1_d,temp2_d,CUFFT_FORWARD)
389   FiMass=temp2_d(1,1)
390
391   PRINT*,'Initial mass',InMass
392   PRINT*,'Final mass',FiMass
393   PRINT*,'Final Mass/Initial Mass', &
394      ABS(REAL(FiMass,kind=fp_kind)/REAL(InMass,kind=fp_kind))
395
396
397   ! Get final energy
398   !$cuf kernel do <<<  *,*  >>>
399   DO j=1,Ny
400     DO i=1,Nx
401       temp1_d(i,j)=-ES*0.25d0*abs(u_d(i,j))**4
402     END DO
403   END DO
404   ! Use FFT to find mean
405   CALL cufftExecZ2Z(plan,temp1_d,temp2_d,CUFFT_FORWARD)
406   FiEner=temp2_d(1,1)
407   !$cuf kernel do <<<  *,*  >>>
408   DO j=1,Ny
409     DO i=1,Nx
410       temp2_d(i,j)=kx_d(i)*v_d(i,j)*scalemodes
411     END DO
412   END DO
413   CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_INVERSE)
414   !$cuf kernel do <<<  *,*  >>>
415   DO j=1,Ny
416     DO i=1,Nx
417       temp2_d(i,j)=0.5d0*abs(temp1_d(i,j))**2
418     END DO
419   END DO
420   ! Use FFT to find mean
```

```
421   CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_FORWARD)
422   FiEner=FiEner+temp1_d(1,1)
423   !$cuf kernel do <<< *,* >>>
424   DO j=1,Ny
425     DO i=1,Nx
426       temp2_d(i,j)=ky_d(j)*v_d(i,j)*scalemodes
427     END DO
428   END DO
429   CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_INVERSE)
430   !$cuf kernel do <<<  *,*  >>>
431   DO j=1,Ny
432     DO i=1,Nx
433       temp2_d(i,j)=0.5d0*abs(temp1_d(i,j))**2
434     END DO
435   END DO
436   ! Use FFT to find mean
437   CALL cufftExecZ2Z(plan,temp2_d,temp1_d,CUFFT_FORWARD)
438   FiEner=FiEner+temp1_d(1,1)
439
440   PRINT*,'Initial energy',InEner
441   PRINT*,'Final energy',FiEner
442   PRINT*,'Final Energy/Initial Energy', &
443      ABS(REAL(FiEner,kind=fp_kind)/REAL(InEner,kind=fp_kind))
444
445   ! Copy results back to host
446   u=u_d
447   time=time_d
448   x=x_d
449   y=y_d
450
451   name_config = 'ufinal.dat'
452   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
453   REWIND(11)
454   DO j=1,Ny
455     DO i=1,Nx
456       WRITE(11,*) abs(u(i,j))**2
457     END DO
458   END DO
459   CLOSE(11)
460
461   name_config = 'tdata.dat'
462   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
463   REWIND(11)
464   DO j=1,1+Nt/plotgap
465     WRITE(11,*) time(j)
466   END DO
467   CLOSE(11)
468
469   name_config = 'xcoord.dat'
470   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
471   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 ..
```

```
20  !  Nx       = number of modes in x - power of 2 for FFT
21  !  Ny       = number of modes in y - power of 2 for FFT
22  !  Nt       = number of timesteps to take
23  !  Tmax       = maximum simulation time
24  !  plotgap      = number of timesteps between plots
25  !  FFTW_IN_PLACE  = value for FFTW input
26  !  FFTW_MEASURE   = value for FFTW input
27  !  FFTW_EXHAUSTIVE  = value for FFTW input
28  !  FFTW_PATIENT   = value for FFTW input
29  !  FFTW_ESTIMATE  = value for FFTW input
30  !  FFTW_FORWARD     = value for FFTW input
31  !  FFTW_BACKWARD  = value for FFTW input
32  !  pi = 3.14159265358979323846264338327950288419716939937510d0
33  !  Lx       = width of box in x direction
34  !  Ly       = width of box in y direction
35  !  ES       = +1 for focusing and -1 for defocusing
36  ! .. Scalars ..
37  !  i        = loop counter in x direction
38  !  j        = loop counter in y direction
39  !  n        = loop counter for timesteps direction
40  !  allocatestatus = error indicator during allocation
41  !  numthreads   = number of openmp threads
42  !  ierr       = error return code
43  !  start      = variable to record start time of program
44  !  finish     = variable to record end time of program
45  !  count_rate   = variable for clock count rate
46  !  planfxy      = Forward 2d fft plan
47  !  planbxy      = Backward 2d fft plan
48  !  dt       = timestep
49  !  InMass     = initial mass
50  !  FiMass     = final mass
51  !  InEner     = initial energy
52  !  FiEner     = final energy
53  ! .. Arrays ..
54  !  u        = approximate solution
55  !  v        = Fourier transform of approximate solution
56  !  temp1      = temporary field
57  !  temp2      = temporary field
58  ! .. Vectors ..
59  !  kx       = fourier frequencies in x direction
60  !  ky       = fourier frequencies in y direction
61  !  x        = x locations
62  !  y        = y locations
63  !  time       = times at which save data
64  !  name_config    = array to store filename for data to be saved
65  !
66  ! REFERENCES
67  !
68  ! This program is based on example code to demonstrate usage of Fortran
       and
69  ! CUDA FFT routines taken from
```

```
70  ! http://cudamusing.blogspot.com/2010/05/CALLing-cufft-from-cuda-fortran
       .html
71  !
72  ! and
73  !
74  ! http://cudamusing.blogspot.com/search?q=cublas
75  !
76  ! ACKNOWLEDGEMENTS
77  !
78  ! ACCURACY
79  !
80  ! ERROR INDICATORS AND WARNINGS
81  !
82  ! FURTHER COMMENTS
83  ! Check that the initial iterate is consistent with the
84  ! boundary conditions for the domain specified
85  !-----------------------------------------------------------------------
86  ! External routines required
87  ! precision
88  ! cufft
89  !
90  ! External libraries required
91  ! CuFFT  -- Cuda FFT Library
92  ! OpenACC
93
94  !
95  ! Define the INTERFACE to the NVIDIA CUFFT routines
96  !
97
98  module precision
99  ! Precision control
100
101 integer, parameter, public :: Single = kind(0.0) ! Single precision
102 integer, parameter, public :: Double = kind(0.0d0) ! Double precision
103
104 integer, parameter, public :: fp_kind = Double
105 !integer, parameter, public :: fp_kind = Single
106
107 end module precision
108
109 module cufft
110
111 integer, public :: CUFFT_FORWARD = -1
112 integer, public :: CUFFT_INVERSE = 1
113 integer, public :: CUFFT_R2C = Z'2a' ! Real to Complex (interleaved)
114 integer, public :: CUFFT_C2R = Z'2c' ! Complex (interleaved) to Real
115 integer, public :: CUFFT_C2C = Z'29' ! Complex to Complex, interleaved
116 integer, public :: CUFFT_D2Z = Z'6a' ! Double to Double-Complex
117 integer, public :: CUFFT_Z2D = Z'6c' ! Double-Complex to Double
118 integer, public :: CUFFT_Z2Z = Z'69' ! Double-Complex to Double-Complex
119
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! 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
```

```
171
172   ! Declare variables
173   IMPLICIT NONE
174   INTEGER(kind=4), PARAMETER  ::  Nx=128
175   INTEGER(kind=4), PARAMETER  ::  Ny=128
176   INTEGER(kind=4), PARAMETER  ::  Nt=20
177   INTEGER(kind=4), PARAMETER  ::  plotgap=20
178   REAL(fp_kind), PARAMETER    ::  &
179   pi=3.14159265358979323846264338327950288419716939937510d0
180   REAL(fp_kind), PARAMETER    ::  Lx=5.0d0
181   REAL(fp_kind), PARAMETER    ::  Ly=5.0d0
182   REAL(fp_kind), PARAMETER    ::  Es=1.0d0
183   REAL(fp_kind)          ::  dt=0.10d0**5
184   REAL(fp_kind)          :: scalemodes
185   COMPLEX(fp_kind)         :: InMass,FiMass,InEner,FiEner
186   COMPLEX(fp_kind), DIMENSION(:), ALLOCATABLE   ::  kx
187   COMPLEX(fp_kind), DIMENSION(:), ALLOCATABLE   ::  ky
188   REAL(fp_kind),     DIMENSION(:), ALLOCATABLE  ::  x
189   REAL(fp_kind),     DIMENSION(:), ALLOCATABLE  ::  y
190   COMPLEX(fp_kind), DIMENSION(:,:), ALLOCATABLE ::  u,v,temp1,temp2
191   REAL(fp_kind),   DIMENSION(:), ALLOCATABLE    ::  time
192   INTEGER(kind=4)        ::  i,j,k,n,allocatestatus,ierr, vecsize,gangsize
193   REAL(fp_kind)        ::  start_time,stop_time
194   INTEGER(kind=4)        ::  plan
195     CHARACTER*100        ::  name_config
196
197   vecsize=32
198   gangsize=16
199   PRINT *,'Program starting'
200   PRINT *,'Grid: ',Nx,'X',Ny
201
202   ALLOCATE(kx(1:Nx),ky(1:Nx),x(1:Nx),y(1:Nx),u(1:Nx,1:Ny),&
203       v(1:Nx,1:Ny),temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny),&
204       time(1:1+Nt/plotgap),stat=allocatestatus)
205   IF (allocatestatus .ne. 0) stop
206   PRINT *,'allocated memory'
207
208   !$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
      time)
209
210   ! set up ffts
211   CALL cufftPlan2D(plan,nx,ny,CUFFT_Z2Z)
212   PRINT *,'Setup FFTs'
213
214   ! setup fourier frequencies
215   !$acc kernels loop
216   DO i=1,1+Nx/2
217     kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
218   END DO
219   !$acc end kernels
220   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)
```

```
272   ! Get initial energy
273   !$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
          time)
274   !$acc kernels loop
275   DO j=1,Ny
276     DO i=1,Nx
277       temp1(i,j)=-ES*0.25d0*abs(u(i,j))**4
278     END DO
279   END DO
280   !$acc end kernels
281   ! Use FFT to find mean
282   CALL cufftExecZ2Z(plan,temp1,temp2,CUFFT_FORWARD)
283   !$acc end data
284   InEner=temp2(1,1)
285   !$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
          time)
286   !$acc kernels loop
287   DO j=1,Ny
288     DO i=1,Nx
289       temp2(i,j)=kx(i)*v(i,j)*scalemodes
290     END DO
291   END DO
292   !$acc end kernels
293   CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_INVERSE)
294   !$acc kernels loop
295   DO j=1,Ny
296     DO i=1,Nx
297       temp2(i,j)=0.5d0*abs(temp1(i,j))**2
298     END DO
299   END DO
300   !$acc end kernels
301   ! Use FFT to find mean
302   CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_FORWARD)
303   !$acc end data
304   InEner=InEner+temp1(1,1)
305   !$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
          time)
306   !$acc kernels loop
307   DO j=1,Ny
308     DO i=1,Nx
309       temp2(i,j)=ky(j)*v(i,j)*scalemodes
310     END DO
311   END DO
312   !$acc end kernels
313   CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_INVERSE)
314   !$acc kernels loop
315   DO j=1,Ny
316     DO i=1,Nx
317       temp2(i,j)=0.5d0*abs(temp1(i,j))**2
318     END DO
319   END DO
```

```
320   !$acc end kernels
321   ! Use FFT to find mean
322   CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_FORWARD)
323   !$acc end data
324   InEner=InEner+temp1(1,1)
325   !$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
        time)
326   CALL cpu_time(start_time)
327
328
329   ! transform initial data and do first half time step
330   !$acc kernels loop gang(gangsize), vector(vecsize)
331   DO j=1,Ny
332     DO i=1,Nx
333       v(i,j)=exp(0.5d0*dt*(kx(i)*kx(i) + ky(j)*ky(j))&
334           *cmplx(0.0d0,1.0d0))*v(i,j)
335     END DO
336   END DO
337   !$acc end kernels
338   PRINT *,'Got initial data, starting timestepping'
339   time(1)=0.0d0
340   DO n=1,Nt
341     CALL cufftExecZ2Z(plan,v,u,CUFFT_INVERSE)
342       !$acc kernels loop gang(gangsize), vector(vecsize)
343     DO j=1,Ny
344       DO i=1,Nx
345         v(i,j)=Es*u(i,j)*conjg(u(i,j))*scalemodes**2
346       END DO
347     END DO
348       !$acc end kernels
349       !$acc kernels loop gang(gangsize), vector(vecsize)
350     DO j=1,Ny
351       DO i=1,Nx
352         u(i,j)=exp(cmplx(0.0d0,-1.0d0)*dt*v(i,j))&
353             *u(i,j)*scalemodes
354       END DO
355     END DO
356       !$acc end kernels
357     CALL cufftExecZ2Z(plan,u,v,CUFFT_FORWARD)
358       !$acc kernels loop gang(gangsize), vector(vecsize)
359     DO j=1,Ny
360       DO i=1,Nx
361         v(i,j)=exp(dt*(kx(i)*kx(i) + ky(j)*ky(j))&
362             *cmplx(0.0d0,1.0d0))*v(i,j)
363       END DO
364     END DO
365       !$acc end kernels
366     IF (mod(n,plotgap)==0) then
367       time(1+n/plotgap)=n*dt
368       PRINT *,'time',n*dt
369     END IF
```

```
370   END DO
371   ! transform back final data and do another half time step
372   CALL cufftExecZ2Z(plan,v,u,CUFFT_INVERSE)
373   !$acc kernels loop gang(gangsize), vector(vecsize)
374   DO j=1,Ny
375     DO i=1,Nx
376       v(i,j)=Es*u(i,j)*conjg(u(i,j))*scalemodes**2
377     END DO
378   END DO
379   !$acc end kernels
380   !$acc kernels loop gang(gangsize), vector(vecsize)
381   DO j=1,Ny
382     DO i=1,Nx
383       u(i,j)=exp(cmplx(0,-1)*dt*v(i,j))*u(i,j)*scalemodes
384     END DO
385   END DO
386   !$acc end kernels
387   CALL cufftExecZ2Z(plan,u,v,CUFFT_FORWARD)
388   !$acc kernels loop gang(gangsize), vector(vecsize)
389   DO j=1,Ny
390     DO i=1,Nx
391       v(i,j)=exp(0.5d0*dt*(kx(i)*kx(i) + ky(j)*ky(j))&
392           *cmplx(0.0d0,1.0d0))*v(i,j)
393     END DO
394   END DO
395   !$acc end kernels
396   CALL cufftExecZ2Z(plan,v,u,CUFFT_INVERSE)
397   !$acc kernels loop gang(gangsize), vector(vecsize)
398   DO j=1,Ny
399     DO i=1,Nx
400       u(i,j)=u(i,j)*scalemodes
401     END DO
402   END DO
403   !$acc end kernels
404   PRINT *,'Finished time stepping'
405   CALL cpu_time(stop_time)
406   !$acc end data
407   PRINT*,'Program took ',stop_time-start_time,&
408     'for Time stepping'
409   !$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
        time)
410
411   ! calculate final mass
412   !$acc kernels loop
413   DO j=1,Ny
414     DO i=1,Nx
415       temp1(i,j)=abs(u(i,j))**2
416     END DO
417   END DO
418   !$acc end kernels
419   ! Use FFT to get initial mass
```

```
420   CALL cufftExecZ2Z(plan,temp1,temp2,CUFFT_FORWARD)
421   !$acc end data
422   FiMass=temp2(1,1)
423
424
425   ! Get final energy
426   !$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
          time)
427   !$acc kernels loop
428   DO j=1,Ny
429     DO i=1,Nx
430       temp1(i,j)=-ES*0.25d0*abs(u(i,j))**4
431     END DO
432   END DO
433   !$acc end kernels
434   ! Use FFT to find mean
435   CALL cufftExecZ2Z(plan,temp1,temp2,CUFFT_FORWARD)
436   !$acc end data
437   FiEner=temp2(1,1)
438   !$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
          time)
439   !$acc kernels loop
440   DO j=1,Ny
441     DO i=1,Nx
442       temp2(i,j)=kx(i)*v(i,j)*scalemodes
443     END DO
444   END DO
445   !$acc end kernels
446   CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_INVERSE)
447   !$acc kernels loop
448   DO j=1,Ny
449     DO i=1,Nx
450       temp2(i,j)=0.5d0*abs(temp1(i,j))**2
451     END DO
452   END DO
453   !$acc end kernels
454   ! Use FFT to find mean
455   CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_FORWARD)
456   !$acc end data
457   FiEner=FiEner+temp1(1,1)
458   !$acc data copy(InMass,FiMass,InEner,FiEner,kx,ky,x,y,u,v,temp1,temp2,
          time)
459   !$acc kernels loop
460   DO j=1,Ny
461     DO i=1,Nx
462       temp2(i,j)=ky(j)*v(i,j)*scalemodes
463     END DO
464   END DO
465   !$acc end kernels
466   CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_INVERSE)
467   !$acc kernels loop
```

```
468   DO j=1,Ny
469     DO i=1,Nx
470       temp2(i,j)=0.5d0*abs(temp1(i,j))**2
471     END DO
472   END DO
473   !$acc end kernels
474   ! Use FFT to find mean
475   CALL cufftExecZ2Z(plan,temp2,temp1,CUFFT_FORWARD)
476   !$acc end data
477   FiEner=FiEner+temp1(1,1)
478
479   PRINT *,'Results copied back to host'
480   PRINT*,'Initial mass',InMass
481   PRINT*,'Final mass',FiMass
482   PRINT*,'Final Mass/Initial Mass', &
483      ABS(REAL(FiMass,kind(0d0))/REAL(InMass,kind(0d0)))
484   PRINT*,'Initial energy',InEner
485   PRINT*,'Final energy',FiEner
486   PRINT*,'Final Energy/Initial Energy', &
487      ABS(REAL(FiEner,kind(0d0))/REAL(InEner,kind(0d0)))
488
489   name_config = 'ufinal.dat'
490   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
491   REWIND(11)
492   DO j=1,Ny
493     DO i=1,Nx
494       WRITE(11,*) abs(u(i,j))**2
495     END DO
496   END DO
497   CLOSE(11)
498
499   name_config = 'tdata.dat'
500   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
501   REWIND(11)
502   DO j=1,1+Nt/plotgap
503     WRITE(11,*) time(j)
504   END DO
505   CLOSE(11)
506
507   name_config = 'xcoord.dat'
508   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
509   REWIND(11)
510   DO i=1,Nx
511     WRITE(11,*) x(i)
512   END DO
513   CLOSE(11)
514
515   name_config = 'ycoord.dat'
516   OPEN(unit=11,FILE=name_config,status="UNKNOWN")
517   REWIND(11)
518   DO j=1,Ny
```

```
519     WRITE(11,*) y(j)
520   END DO
521   CLOSE(11)
522   PRINT *,'Saved data'
523
524   CALL cufftDestroy(plan)
525
526   DEALLOCATE(u,v,temp1,temp2,time,kx,ky,x,y,stat=allocatestatus)
527   IF (allocatestatus .ne. 0) STOP
528   PRINT *,'Deallocated memory'
529
530     PRINT *,'Program execution complete'
531   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.

```
1  !-----------------------------------------------------------------------
2  !
3  !
4  ! PURPOSE
5  !
6  ! This program solves nonlinear sine-Gordon equation in 2 dimensions
7  ! u_{tt}-u_{xx}-u_{yy}=-sin(u)
8  ! using a second order implicit-explicit time stepping scheme.
9  !
10 ! The boundary conditions are u(x=0,y)=u(2*Lx*\pi,y),
11 !       u(x,y=0)=u(x,y=2*Ly*\pi)
12 ! The initial condition is set in initialdata.f90
13 !
14 ! AUTHORS
15 !
16 ! B. Cloutier, B.K. Muite, P. Rigge
17 ! 4 June 2012
18 !
19 ! .. Parameters ..
20 !  Nx                          = number of modes in x - power of 2 for
       FFT
21 !  Ny                          = number of modes in y - power of 2 for
       FFT
22 !  Nt                          = number of timesteps to take
23 !  plotgap                     = number of timesteps between plots
24 !  FFTW_IN_PLACE               = value for FFTW input
```

```
25 !  FFTW_MEASURE                = value for FFTW input
26 !  FFTW_EXHAUSTIVE             = value for FFTW input
27 !  FFTW_PATIENT                = value for FFTW input
28 !  FFTW_ESTIMATE               = value for FFTW input
29 !  FFTW_FORWARD                = value for FFTW input
30 !  FFTW_BACKWARD               = value for FFTW input
31 !  pi                          = 3.1415926535...
32 !  Lx                          = width of box in x direction
33 !  Ly                          = width of box in y direction
34 ! .. Scalars ..
35 !  i                           = loop counter in x direction
36 !  j                           = loop counter in y direction
37 !  n                           = loop counter for timesteps direction
38 !  allocatestatus              = error indicator during allocation
39 !  start                       = variable to record start time of program
40 !  finish                      = variable to record end time of program
41 !  count_rate                  = variable for clock count rate
42 !  planfxy                     = Forward 2d fft plan  (FFTW)
43 !  planbxy                     = Backward 2d fft plan (FFTW)
44 !  planf                       = Forward 2d fft plan  (CUFFT)
45 !  planb                       = Backward 2d fft plan (CUFFT)
46 !  dt                          = timestep
47 !  ierr                        = error code
48 !  plotnum                     = number of plot
49 ! .. Arrays ..
50 !  u                           = approximate solution
51 !  uold                        = approximate solution
52 !  u_d                         = approximate solution (on GPU)
53 !  v_d                         = Fourier transform of approximate
     solution (on GPU)
54 !  uold_d                      = approximate solution (on GPU)
55 !  vold_d                      = Fourier transform of approximate
     solution (on GPU)
56 !  nonlinhat_d                 = Fourier transform of nonlinear term, sin
     (u) (on GPU)
57 !  temp1                       = extra space for energy computation
58 !  temp2                       = extra space for energy computation
59 !  savearray                   = temp array to save out to disk
60 ! .. Vectors ..
61 !  kx                          = Fourier frequencies in x direction
62 !  ky                          = Fourier frequencies in y direction
63 !  kx_d                        = Fourier frequencies in x direction (on
     GPU)
64 !  ky_d                        = Fourier frequencies in y direction (on
     GPU)
65 !  x                           = x locations
66 !  y                           = y locations
67 !  time                        = times at which save data
68 !  en                          = total energy
69 !  enstr                       = strain energy
70 !  enpot                       = potential energy
```

```
71 !  enkin                          = kinetic energy
72 !  name_config                 = array to store filename for data to be saved
73 !
74 ! REFERENCES
75 !
76 ! ACKNOWLEDGEMENTS
77 !
78 ! This program is based on example code to demonstrate usage of Fortran
     and
79 ! CUDA FFT routines taken from
80 ! http://cudamusing.blogspot.com/2010/05/CALLing-cufft-from-cuda-fortran.
     html
81 !
82 ! and
83 !
84 ! http://cudamusing.blogspot.com/search?q=cublas
85 !
86 ! ACCURACY
87 !
88 ! ERROR INDICATORS AND WARNINGS
89 !
90 ! FURTHER COMMENTS
91 ! Check that the initial iterate is consistent with the
92 ! boundary conditions for the domain specified
93 !-----------------------------------------------------------------------
94 ! External routines required
95 !       getgrid.f90     -- Get initial grid of points
96 !       initialdata.f90 -- Get initial data
97 !       enercalc.f90    -- Subroutine to calculate the energy
98 !       savedata.f90    -- Save initial data
99 ! External libraries required
100 !       Cuda FFT        -- http://developer.nvidia.com/cufft
101 !       FFTW3           -- Fastest Fourier Transform in the West
102 !                          (http://www.fftw.org/)
103
104 module precision
105   ! Precision control
106   integer, parameter, public :: Single = kind(0.0) ! Single precision
107   integer, parameter, public :: Double = kind(0.0d0) ! Double precision
108   !
109   integer, parameter, public :: fp_kind = Double
110   !integer, parameter, public :: fp_kind = Single
111 end module precision
112
113 module cufft
114   integer, public :: CUFFT_FORWARD = -1
115   integer, public :: CUFFT_INVERSE = 1
116   integer, public :: CUFFT_R2C = Z'2a' ! Real to Complex (interleaved)
117   integer, public :: CUFFT_C2R = Z'2c' ! Complex (interleaved) to Real
118   integer, public :: CUFFT_C2C = Z'29' ! Complex to Complex, interleaved
119   integer, public :: CUFFT_D2Z = Z'6a' ! Double to Double-Complex
```

```
120   integer, public :: CUFFT_Z2D = Z'6c' ! Double-Complex to Double
121   integer, public :: CUFFT_Z2Z = Z'69' ! Double-Complex to Double-Complex
122 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
123   !
124   ! cufftPlan2d(cufftHandle *plan, int nx,int ny, cufftType type,int batch
          )
125   !
126 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
127   interface cufftPlan2d
128      subroutine cufftPlan2d(plan, nx, ny, type) bind(C,name='cufftPlan2d')
129        use iso_c_binding
130        integer(c_int):: plan
131        integer(c_int),value:: nx, ny, type
132      end subroutine cufftPlan2d
133   end interface cufftPlan2d
134 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
135   !
136   ! cufftDestroy(cufftHandle plan)
137   !
138 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
139   interface cufftDestroy
140      subroutine cufftDestroy(plan) bind(C,name='cufftDestroy')
141        use iso_c_binding
142        integer(c_int),value:: plan
143      end subroutine cufftDestroy
144   end interface cufftDestroy
145 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
146   !
147   ! cufftExecD2Z(cufftHandle plan,
148   ! cufftDoubleReal    *idata,
149   ! cufftDoubleComplex *odata)
150   !
151 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
152   interface cufftExecD2Z
153      subroutine cufftExecD2Z(plan, idata, odata) &
154           & bind(C,name='cufftExecD2Z')
155        use iso_c_binding
156        use precision
157        integer(c_int),  value  :: plan
158        real(fp_kind),   device :: idata(1:nx,1:ny)
159        complex(fp_kind),device :: odata(1:nx,1:ny)
160      end subroutine cufftExecD2Z
161   end interface cufftExecD2Z
162 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
163   !
164   ! cufftExecD2Z(cufftHandle plan,
165   ! cufftDoubleComplex *idata,
166   ! cufftDoubleReal    *odata)
167   !
168 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
169   interface cufftExecZ2D
```

```
170       subroutine cufftExecZ2D(plan, idata, odata) &
171            & bind(C,name='cufftExecZ2D')
172         use iso_c_binding
173         use precision
174         integer(c_int),value    :: plan
175         complex(fp_kind),device:: idata(1:nx,1:ny)
176         real(fp_kind),device    :: odata(1:nx,1:ny)
177       end subroutine cufftExecZ2D
178     end interface cufftExecZ2D
179   end module cufft
180
181   PROGRAM sg2d
182     USE precision
183     USE cudafor
184     USE cufft
185     ! Declare variables
186     IMPLICIT NONE
187     INTEGER(kind=4), PARAMETER                              :: Nx=1024
188     INTEGER(kind=4), PARAMETER                              :: Ny=Nx
189     INTEGER(kind=4), PARAMETER                              :: Nt=500
190     INTEGER(kind=4), PARAMETER                              :: plotgap=Nt+1
191     REAL(kind=8), PARAMETER                                 :: &
192          pi=3.14159265358979323846264338327950288419716939937510d0
193     REAL(kind=8), PARAMETER                                 :: Lx=5.0d0
194     REAL(kind=8), PARAMETER                                 :: Ly=5.0d0
195     REAL(kind=8)                                            :: dt=0.001d0
196     COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE              :: kx,ky
197     REAL(kind=8),           DIMENSION(:), ALLOCATABLE       :: x,y
198     REAL    (kind=8), DIMENSION(:,:), ALLOCATABLE           :: u,uold
199     COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE            :: temp1,temp2
200     REAL(kind=8), DIMENSION(:,:), ALLOCATABLE               :: savearray
201     REAL(kind=8), DIMENSION(:), ALLOCATABLE                 :: time,enkin,enstr
        ,enpot,en
202     REAL(kind=8)                                            :: scalemodes
203     INTEGER(kind=4)                                         :: ierr,i,j,n,
        allocatestatus
204     INTEGER(kind=4)                                         :: start, finish,
        count_rate, plotnum
205     INTEGER(kind=4), PARAMETER                              :: FFTW_IN_PLACE =
        8, FFTW_MEASURE = 0, &
206       FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32, FFTW_ESTIMATE = 64
207     INTEGER(kind=4),PARAMETER                               :: FFTW_FORWARD =
        -1, FFTW_BACKWARD=1
208     INTEGER(kind=8)                                         :: planfxy,planbxy
209     CHARACTER*100                                           :: name_config
210     INTEGER(kind=4)                                         :: planf,planb,
        kersize
211     ! GPU variables
212     COMPLEX(fp_kind),DEVICE,DIMENSION(:), ALLOCATABLE       :: kx_d,ky_d
213     COMPLEX(fp_kind),DEVICE,DIMENSION(:,:), ALLOCATABLE     :: vold_d,v_d,
        nonlinhat_d
```

```
214   REAL   (fp_kind),DEVICE,DIMENSION(:,:), ALLOCATABLE  :: uold_d,u_d
215   ! print run information
216   PRINT *,"Nx=", Nx
217   PRINT *,"Ny=", Ny
218   PRINT *,"Nt=", Nt
219   PRINT *,"Lx=", Lx
220   PRINT *,"Ly=", Ly
221   PRINT *,"dt=", dt
222   kersize=min(Nx,256)
223   ALLOCATE(kx(1:Nx),ky(1:Ny),kx_d(1:Nx),ky_d(1:Ny),x(1:Nx),y(1:Ny),&
224        u(1:Nx,1:Ny),uold(1:Nx,1:Ny),u_d(1:Nx,1:Ny),uold_d(1:Nx,1:Ny),&
225        v_d(1:Nx/2+1,1:Ny),vold_d(1:Nx/2+1,1:Ny),&
226        savearray(1:Nx,1:Ny),time(1:1+Nt/plotgap),enkin(1:1+Nt/plotgap+1),&
227        enstr(1:1+Nt/plotgap+1),enpot(1:1+Nt/plotgap+1),en(1:1+Nt/plotgap)
           ,&
228        nonlinhat_d(1:Nx/2+1,1:Ny),&
229        temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny),&
230        stat=allocatestatus)
231   IF (allocatestatus .ne. 0) stop
232   PRINT *,'allocated arrays'
233   scalemodes=1.0d0/REAL(Nx*Ny,kind(0d0))
234   ! set up cuda ffts
235   call cufftPlan2D(planf,nx,ny,CUFFT_D2Z)
236   call cufftPlan2D(planb,nx,ny,CUFFT_Z2D)
237   ! set up fftw ffts
238   CALL dfftw_plan_dft_2d_(planfxy,Nx,Ny,u,temp2,FFTW_FORWARD,FFTW_ESTIMATE
      )
239   CALL dfftw_plan_dft_2d_(planbxy,Nx,Ny,temp2,u,FFTW_BACKWARD,
      FFTW_ESTIMATE)
240   PRINT *,'Setup FFTs'
241   ! setup grid, wave numbers
242   CALL getgrid(Nx,Ny,Lx,Ly,pi,name_config,x,y,kx,ky)
243   kx_d=kx
244   ky_d=ky
245   PRINT *,'Got grid and fourier frequencies'
246
247   CALL initialdata(Nx,Ny,x,y,u,uold)
248   u_d=u
249   uold_d=uold
250   plotnum=1
251   name_config = 'data/u'
252   savearray=REAL(u)
253   ! CALL savedata(Nx,Ny,plotnum,name_config,savearray) ! disabled for
      benchmarking
254   PRINT *,'data saved'
255
256   CALL enercalc(Nx,Ny,planfxy,planbxy,dt,enkin(plotnum),enstr(plotnum),&
257        enpot(plotnum),en(plotnum),kx,ky,temp1,temp2,u,uold)
258   call cufftExecD2Z(planf,u_d,v_d)
259   call cufftExecD2Z(planf,uold_d,vold_d)
260   PRINT *,'Got initial data, starting timestepping'
```

```
261   time(plotnum)=0.0d0
262   CALL system_clock(start,count_rate)
263   DO n=1,Nt
264      !$cuf kernel do(2) <<< (1,*), (kersize,1) >>>
265      DO j=1,Ny
266         DO i=1,Nx
267            uold_d(i,j)=u_d(i,j)
268         END DO
269      END DO
270      !$cuf kernel do(2) <<< (1,*), (kersize,1) >>>
271      DO j=1,Ny
272         DO i=1,Nx
273            u_d(i,j)=sin(u_d(i,j))
274         END DO
275      END DO
276      call cufftExecD2Z(planf,u_d,nonlinhat_d)
277      !$cuf kernel do(2) <<< (1,*), (kersize,1) >>>
278      DO j=1,Ny
279         DO i=1,Nx/2+1
280            nonlinhat_d(i,j)=scalemodes*( 0.25*(kx_d(i)*kx_d(i) + ky_d(j)*
                  ky_d(j))&
281              *(2.0d0*v_d(i,j)+vold_d(i,j))&
282              +(2.0d0*v_d(i,j)-vold_d(i,j))/(dt*dt)&
283              -nonlinhat_d(i,j) )&
284              /(1/(dt*dt)-0.25*(kx_d(i)*kx_d(i) + ky_d(j)*ky_d(j)))
285         END DO
286      END DO
287      !$cuf kernel do(2) <<< (1,*), (kersize,1) >>>
288      DO j=1,Ny
289         DO i=1,Nx/2+1
290            vold_d(i,j)=v_d(i,j)
291         END DO
292      END DO
293      !$cuf kernel do(2) <<< (1,*), (kersize,1) >>>
294      DO j=1,Ny
295         DO i=1,Nx/2+1
296            v_d(i,j)=nonlinhat_d(i,j)/scalemodes
297         END DO
298      END DO
299      call cufftExecZ2D(planb,nonlinhat_d,u_d)
300      IF (mod(n,plotgap)==0) then
301         plotnum=plotnum+1
302         time(plotnum)=n*dt
303         PRINT *,'time',n*dt
304         u=u_d
305         uold=uold_d
306         ! savearray=REAL(u,kind(0d0)) ! disabled for benchmarking
307         ! CALL savedata(Nx,Ny,plotnum,name_config,savearray)
308         CALL enercalc(Nx,Ny,planfxy,planbxy,dt,enkin(plotnum),enstr(
               plotnum),&
309              enpot(plotnum),en(plotnum),kx,ky,temp1,temp2,u,uold)
```

```
310       END IF
311     END DO
312     CALL system_clock(finish,count_rate)
313     PRINT *,'Finished time stepping'
314     u=u_d
315     uold=uold_d
316     ! compute energy at the end
317     CALL enercalc(Nx,Ny,planfxy,planbxy,dt,enkin(plotnum+1),enstr(plotnum+1)
            ,&
318           enpot(plotnum+1),en(plotnum+1),kx,ky,temp1,temp2,u,uold)
319
320     PRINT*,'Program took ',&
321           REAL(finish-start,kind(0d0))/REAL(count_rate,kind(0d0)),&
322           'for Time stepping'
323     CALL saveresults(Nt,plotgap,time(1:1+n/plotgap),en(1:1+n/plotgap+1),&
324           enstr(1:1+n/plotgap+1),enkin(1:1+n/plotgap+1),enpot(1:1+n/plotgap
                +1))
325
326     ! Save times at which output was made in text format
327     PRINT *,'Saved data'
328
329     call cufftDestroy(planf)
330     call cufftDestroy(planb)
331     PRINT *,'Destroy CUFFT Plans'
332     call dfftw_destroy_plan_(planbxy)
333     call dfftw_destroy_plan_(planfxy)
334     PRINT *,'Destroy FFTW Plans'
335     DEALLOCATE(kx,ky,x,y,u,uold,time,enkin,enstr,enpot,en,savearray,temp1,
            temp2,&
336           stat=allocatestatus)
337     IF (allocatestatus .ne. 0) STOP
338     PRINT *,'Deallocated host arrays'
339     DEALLOCATE(uold_d,vold_d,u_d,v_d,nonlinhat_d,&
340           kx_d,ky_d,&
341           stat=allocatestatus)
342     IF (allocatestatus .ne. 0) STOP
343     PRINT *,'Deallocated gpu arrays'
344     PRINT *,'Program execution complete'
345   END PROGRAM sg2d
```

Listing A.6: An OpenACC Fortran program to solve the 2D sine-Gordon equation.

```
1  !----------------------------------------------------------------------
2  !
3  !
4  ! PURPOSE
5  !
6  ! This program solves nonlinear sine-Gordon equation in 2 dimensions
7  ! u_{tt}-u_{xx}-u_{yy}=-sin(u)
8  ! 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)
```

```
55 !  temp1                          = extra space for energy computation
56 !  temp2                          = extra space for energy computation
57 !  savearray                      = temp array to save out to disk
58 ! .. Vectors ..
59 !  kx                             = fourier frequencies in x direction
60 !  ky                             = fourier frequencies in y direction
61 !  x                              = x locations
62 !  y                              = y locations
63 !  time                           = times at which save data
64 !  en                             = total energy
65 !  enstr                          = strain energy
66 !  enpot                          = potential energy
67 !  enkin                          = kinetic energy
68 !  name_config                    = array to store filename for data to be
       saved
69 !
70 ! REFERENCES
71 !
72 ! ACKNOWLEDGEMENTS
73 !
74 ! This program is based on example code to demonstrate usage of Fortran
     and
75 ! CUDA FFT routines taken from
76 ! http://cudamusing.blogspot.com/2010/05/CALLing-cufft-from-cuda-fortran.
     html
77 !
78 ! and
79 !
80 ! http://cudamusing.blogspot.com/search?q=cublas
81 !
82 ! ACCURACY
83 !
84 ! ERROR INDICATORS AND WARNINGS
85 !
86 ! FURTHER COMMENTS
87 ! Check that the initial iterate is consistent with the
88 ! boundary conditions for the domain specified
89 !-----------------------------------------------------------------------
90 ! External routines required
91 !       getgrid.f90     -- Get initial grid of points
92 !       initialdata.f90 -- Get initial data
93 !       enercalc.f90    -- Subroutine to calculate the energy
94 !       savedata.f90    -- Save initial data
95 ! External libraries required
96 !       Cuda FFT
97 !       OpenACC
98 !       FFTW3           -- Fastest Fourier Transform in the West
99 !                       (http://www.fftw.org/)
100 !       OpenMP
101 module precision
102   ! Precision control
```

```
103   integer, parameter, public :: Single = kind(0.0) ! Single precision
104   integer, parameter, public :: Double = kind(0.0d0) ! Double precision
105   !
106   integer, parameter, public :: fp_kind = Double
107   !integer, parameter, public :: fp_kind = Single
108 end module precision
109
110 module cufft
111   integer, public :: CUFFT_FORWARD = -1
112   integer, public :: CUFFT_INVERSE = 1
113   integer, public :: CUFFT_R2C = Z'2a' ! Real to Complex (interleaved)
114   integer, public :: CUFFT_C2R = Z'2c' ! Complex (interleaved) to Real
115   integer, public :: CUFFT_C2C = Z'29' ! Complex to Complex, interleaved
116   integer, public :: CUFFT_D2Z = Z'6a' ! Double to Double-Complex
117   integer, public :: CUFFT_Z2D = Z'6c' ! Double-Complex to Double
118   integer, public :: CUFFT_Z2Z = Z'69' ! Double-Complex to Double-Complex
119 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
120   !
121   ! cufftPlan2d(cufftHandle *plan, int nx,int ny, cufftType type,int batch
        )
122   !
123 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
124   interface cufftPlan2d
125      subroutine cufftPlan2d(plan, nx, ny, type) bind(C,name='cufftPlan2d')
126        use iso_c_binding
127        integer(c_int):: plan
128        integer(c_int),value:: nx, ny, type
129      end subroutine cufftPlan2d
130   end interface cufftPlan2d
131 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
132   !
133   ! cufftDestroy(cufftHandle plan)
134   !
135 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
136   interface cufftDestroy
137      subroutine cufftDestroy(plan) bind(C,name='cufftDestroy')
138        use iso_c_binding
139        integer(c_int),value:: plan
140      end subroutine cufftDestroy
141   end interface cufftDestroy
142 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
143   !
144   ! cufftExecD2Z(cufftHandle plan,
145   ! cufftDoubleReal    *idata,
146   ! cufftDoubleComplex *odata)
147   !
148 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
149   interface cufftExecD2Z
150      subroutine cufftExecD2Z(plan, idata, odata) &
151           & bind(C,name='cufftExecD2Z')
152        use iso_c_binding
```

```
153         use precision
154         integer(c_int),  value  :: plan
155         real(fp_kind),   device :: idata(1:nx,1:ny)
156         complex(fp_kind),device :: odata(1:nx,1:ny)
157       end subroutine cufftExecD2Z
158     end interface cufftExecD2Z
159 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
160     !
161     ! cufftExecD2Z(cufftHandle plan,
162     ! cufftDoubleComplex *idata,
163     ! cufftDoubleReal    *odata)
164     !
165 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
166     interface cufftExecZ2D
167       subroutine cufftExecZ2D(plan, idata, odata) &
168           & bind(C,name='cufftExecZ2D')
169         use iso_c_binding
170         use precision
171         integer(c_int),value   :: plan
172         complex(fp_kind),device:: idata(1:nx,1:ny)
173         real(fp_kind),device   :: odata(1:nx,1:ny)
174       end subroutine cufftExecZ2D
175     end interface cufftExecZ2D
176 end module cufft
177
178 PROGRAM sg2d
179     USE precision
180     USE cufft
181     USE openacc
182     ! Declare variables
183     IMPLICIT NONE
184     INTEGER(kind=4), PARAMETER                              :: Nx=1024
185     INTEGER(kind=4), PARAMETER                              :: Ny=Nx
186     INTEGER(kind=4), PARAMETER                              :: Nt=500
187     INTEGER(kind=4), PARAMETER                              :: plotgap=Nt+1
188     REAL(kind=8), PARAMETER                                 :: &
189          pi=3.14159265358979323846264338327950288419716939937510d0
190     REAL(kind=8), PARAMETER                                 :: Lx=5.0d0
191     REAL(kind=8), PARAMETER                                 :: Ly=5.0d0
192     REAL(kind=8)                                            :: dt=0.001d0
193     COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE              :: kx,ky
194     REAL(kind=8),           DIMENSION(:), ALLOCATABLE       :: x,y
195     REAL    (kind=8), DIMENSION(:,:), ALLOCATABLE           :: u,uold
196     COMPLEX(kind=8), DIMENSION(:,:), ALLOCATABLE            :: temp1,temp2,v,
        vold,nonlinhat
197     REAL(kind=8), DIMENSION(:,:), ALLOCATABLE               :: savearray
198     REAL(kind=8), DIMENSION(:), ALLOCATABLE                 :: time,enkin,enstr
        ,enpot,en
199     INTEGER(kind=4)                                         :: ierr,i,j,n,
        allocatestatus
```

```
200   INTEGER(kind=4)                                          :: start, finish,
          count_rate, plotnum
201   INTEGER(kind=4), PARAMETER                               :: FFTW_IN_PLACE =
          8, FFTW_MEASURE = 0, &
202         FFTW_EXHAUSTIVE = 8, FFTW_PATIENT = 32, FFTW_ESTIMATE = 64
203   INTEGER(kind=4),PARAMETER                                :: FFTW_FORWARD =
          -1, FFTW_BACKWARD=1
204   INTEGER(kind=8)                                          :: planfxy,planbxy
205   CHARACTER*100                                            :: name_config
206   INTEGER(kind=4)                                          :: planf,planb
207   ! print run information
208   PRINT *,"Nx=", Nx
209   PRINT *,"Ny=", Ny
210   PRINT *,"Nt=", Nt
211   PRINT *,"Lx=", Lx
212   PRINT *,"Ly=", Ly
213   PRINT *,"dt=", dt
214   ALLOCATE(kx(1:Nx),ky(1:Ny),x(1:Nx),y(1:Ny),u(1:Nx,1:Ny),uold(1:Nx,1:Ny)
          ,&
215         v(1:Nx/2+1,1:Ny),vold(1:Nx/2+1,1:Ny),nonlinhat(1:Nx/2+1,1:Ny),&
216         savearray(1:Nx,1:Ny),time(1:1+Nt/plotgap),enkin(1:1+Nt/plotgap+1),&
217         enstr(1:1+Nt/plotgap+1),enpot(1:1+Nt/plotgap+1),en(1:1+Nt/plotgap)
              ,&
218         temp1(1:Nx,1:Ny),temp2(1:Nx,1:Ny),&
219         stat=allocatestatus)
220   IF (allocatestatus .ne. 0) stop
221   PRINT *,'allocated arrays'
222   ! set up cuda ffts
223   call cufftPlan2D(planf,nx,ny,CUFFT_D2Z)
224   call cufftPlan2D(planb,nx,ny,CUFFT_Z2D)
225   ! set up fftw ffts
226   CALL dfftw_plan_dft_2d_(planfxy,Nx,Ny,u,temp2,FFTW_FORWARD,FFTW_ESTIMATE
          )
227   CALL dfftw_plan_dft_2d_(planbxy,Nx,Ny,temp2,u,FFTW_BACKWARD,
          FFTW_ESTIMATE)
228   PRINT *,'Setup FFTs'
229   ! setup grid, wave numbers
230   !$acc data copy(x, y, kx, ky, vold, v, nonlinhat, uold, u)
231   !$acc kernels loop
232   DO i=1,1+Nx/2
233      kx(i)= cmplx(0.0d0,1.0d0)*REAL(i-1,kind(0d0))/Lx
234   END DO
235   !$acc end kernels
236   kx(1+Nx/2)=0.0d0
237   !$acc kernels loop
238   DO i = 1,Nx/2 -1
239      kx(i+1+Nx/2)=-kx(1-i+Nx/2)
240   END DO
241   !$acc end kernels
242   !$acc kernels loop
243   DO i=1,Nx
```

```
244       x(i)=(-1.0d0 + 2.0d0*REAL(i-1,kind(0d0))/REAL(Nx,kind(0d0)))*pi*Lx
245     END DO
246     !$acc end kernels
247     !$acc kernels loop
248     DO j=1,1+Ny/2
249        ky(j)= cmplx(0.0d0,1.0d0)*REAL(j-1,kind(0d0))/Ly
250     END DO
251     !$acc end kernels
252     ky(1+Ny/2)=0.0d0
253     !$acc kernels loop
254     DO j = 1,Ny/2 -1
255        ky(j+1+Ny/2)=-ky(1-j+Ny/2)
256     END DO
257     !$acc end kernels
258     !$acc kernels loop
259     DO j=1,Ny
260        y(j)=(-1.0d0 + 2.0d0*REAL(j-1,kind(0d0))/REAL(Ny,kind(0d0)))*pi*Ly
261     END DO
262     !$acc end kernels
263     PRINT *,'Got grid and fourier frequencies'
264     !$acc kernels loop
265     DO j=1,Ny
266       DO i=1,Nx
267           u(i,j)=0.5d0*exp(-1.0d0*(x(i)**2 +y(j)**2))
268       END DO
269     END DO
270     !$acc end kernels
271     !$acc kernels loop
272     DO j=1,Ny
273       DO i=1,Nx
274         uold(i,j)=0.5d0*exp(-1.0d0*(x(i)**2 +y(j)**2))
275       END DO
276     END DO
277     !$acc end kernels
278     savearray=REAL(u)
279     plotnum=1
280     name_config = 'data/u'
281     ! CALL savedata(Nx,Ny,plotnum,name_config,savearray) ! disabled for
          benchmarking
282     PRINT *,'data saved'
283     !$acc end data
284     CALL enercalc(Nx,Ny,planfxy,planbxy,dt,enkin(plotnum),enstr(plotnum),&
285          enpot(plotnum),en(plotnum),kx(1:Nx),ky(1:Ny),temp1,temp2,&
286          u(1:Nx,1:Ny),uold(1:Nx,1:Ny))
287     !$acc data copy(x, y, kx, ky, vold, v, nonlinhat, uold, u)
288     call cufftExecD2Z(planf,u,v)
289     call cufftExecD2Z(planf,uold,vold)
290     PRINT *,'Got initial data, starting timestepping'
291     time(plotnum)=0.0d0
292     CALL system_clock(start,count_rate)
293     DO n=1,Nt
```

```
294       !$acc kernels loop
295       DO j=1,Ny
296          DO i=1,Nx
297             uold(i,j)=u(i,j)
298             u(i,j)=sin(u(i,j))
299          END DO
300       END DO
301       !$acc end kernels
302       call cufftExecD2Z(planf,u,nonlinhat)
303       !$acc kernels loop
304       DO j=1,Ny
305          DO i=1,Nx/2+1
306             nonlinhat(i,j)=( 0.25*(kx(i)*kx(i) + ky(j)*ky(j))&
307                  *(2.0d0*v(i,j)+vold(i,j))+(2.0d0*v(i,j)-vold(i,j))/(dt*dt)
                        &
308                  -nonlinhat(i,j) )/(1/(dt*dt)-0.25*(kx(i)*kx(i) + ky(j)*ky(
                        j)))
309             vold(i,j)=v(i,j)
310             v(i,j)=nonlinhat(i,j)
311             ! prescale nonlinhat
312             nonlinhat(i,j)=nonlinhat(i,j)/REAL(Nx*Ny,kind(0d0))
313          END DO
314       END DO
315       !$acc end kernels
316       call cufftExecZ2D(planb,nonlinhat,u)
317    END DO
318    CALL system_clock(finish,count_rate)
319    !$acc end data
320    PRINT *,'Finished time stepping'
321    ! compute energy at the end
322    ! savearray=REAL(u(1:Nx,1:Ny),kind(0d0)) ! disabled for benchmarking
323    ! CALL savedata(Nx,Ny,plotnum+1,name_config,savearray)
324    CALL enercalc(Nx,Ny,planfxy,planbxy,dt,enkin(plotnum+1),enstr(plotnum+1)
          ,&
325         enpot(plotnum+1),en(plotnum+1),kx,ky,temp1,temp2,u(1:Nx,1:Ny),uold
             (1:Nx,1:Ny))
326    PRINT*,'Program took ',&
327         REAL(finish-start,kind(0d0))/REAL(count_rate,kind(0d0)),&
328         'for Time stepping'
329    CALL saveresults(Nt,plotgap,time(1:1+n/plotgap),en(1:1+n/plotgap+1),&
330         enstr(1:1+n/plotgap+1),enkin(1:1+n/plotgap+1),enpot(1:1+n/plotgap
             +1))
331
332    ! Save times at which output was made in text format
333    PRINT *,'Saved data'
334
335    call cufftDestroy(planf)
336    call cufftDestroy(planb)
337    PRINT *,'Destroy CUFFT Plans'
338    call dfftw_destroy_plan_(planbxy)
339    call dfftw_destroy_plan_(planfxy)
```

```
340   PRINT *,'Destroy FFTW Plans'
341   DEALLOCATE(kx,ky,x,y,u,uold,time,enkin,enstr,enpot,en,savearray,temp1,
          temp2,&
342         stat=allocatestatus)
343   IF (allocatestatus .ne. 0) STOP
344   PRINT *,'Deallocated host arrays'
345   PRINT *,'Program execution complete'
346 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.

```
#!/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
```

```
15
16 # Grid
17 N = 64                          # Number of steps
18 h = 2*math.pi/N                      # step size
19 x = h*numpy.arange(0,N)      # discretize x-direction
20
21 alpha = 0.5                   # Thermal Diffusivity constant
22 t = 0
23 dt = .001
24
25 # Initial conditions
26 v = numpy.sin(x)
27 I = complex(0,1)
28 k = numpy.array([I*y for y in range(0,N/2) + [0] + range(-N/2+1,0)])
29 k2=k**2;
30
31 # Setting up Plot
32 tmax = 5; tplot = .1;
33 plotgap = int(round(tplot/dt))
34 nplots  = int(round(tmax/tplot))
35
36 data = numpy.zeros((nplots+1,N))
37 data[0,:] = v
38 tdata = [t]
39
40 for i in xrange(nplots):
41     v_hat = numpy.fft.fft(v)
42
43     for n in xrange(plotgap):
44         v_hat = v_hat+dt*alpha*k2*v_hat # FE timestepping
45
46     v = numpy.real(numpy.fft.ifft(v_hat))   # back to real space
47     data[i+1,:] = v
48
49     # real time vector
50     t = t+plotgap*dt
51     tdata.append(t)
52
53 # Plot using mesh
54 xx,tt = (numpy.mat(A) for A in (numpy.meshgrid(x,tdata)))
55
56 fig = plt.figure()
57 ax = fig.gca(projection='3d')
58 surf = ax.plot_surface(xx, tt, data,rstride=1, cstride=1, cmap=cm.jet,
59         linewidth=0, antialiased=False)
60 fig.colorbar(surf, shrink=0.5, aspect=5)
61 plt.xlabel('x')
62 plt.ylabel('t')
63 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.

```
#!/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
```

```
17
18 # Grid
19 Lx=4.0       # Period 2*pi*Lx
20 Ly=4.0       # Period 2*pi*Ly
21 Lz=4.0       # Period 2*pi*Lz
22 Nx=64        # Number of harmonics
23 Ny=64        # Number of harmonics
24 Nz=64        # Number of harmonics
25 Nt=100       # Number of time slices
26 tmax=1.0     # Maximum time
27 dt=tmax/Nt   # time step
28 plotgap=10   # time steps between plots
29 Es= 1.0      # focusing (+1) or defocusing (-1) parameter
30 numplots=Nt/plotgap  # number of plots to make
31
32 x = [i*2.0*math.pi*(Lx/Nx) for i in xrange(-Nx/2,1+Nx/2)]
33 y = [i*2.0*math.pi*(Ly/Ny) for i in xrange(-Ny/2,1+Ny/2)]
34 z = [i*2.0*math.pi*(Lz/Nz) for i in xrange(-Nz/2,1+Nz/2)]
35 k_x = (1.0/Lx)*numpy.array([complex(0,1)*n for n in range(0,Nx/2) \
36 + [0] + range(-Nx/2+1,0)])
37 k_y = (1.0/Ly)*numpy.array([complex(0,1)*n for n in range(0,Ny/2) \
38 + [0] + range(-Ny/2+1,0)])
39 k_z = (1.0/Lz)*numpy.array([complex(0,1)*n for n in range(0,Nz/2) \
40 + [0] + range(-Nz/2+1,0)])
41
42 k2xm=numpy.zeros((Nx,Ny,Nz), dtype=float)
43 k2ym=numpy.zeros((Nx,Ny,Nz), dtype=float)
44 k2zm=numpy.zeros((Nx,Ny,Nz), dtype=float)
45 xx=numpy.zeros((Nx,Ny,Nz), dtype=float)
46 yy=numpy.zeros((Nx,Ny,Nz), dtype=float)
47 zz=numpy.zeros((Nx,Ny,Nz), dtype=float)
48
49
50 for i in xrange(Nx):
51     for j in xrange(Ny):
52         for k in xrange(Nz):
53             k2xm[i,j,k] = numpy.real(k_x[i]**2)
54             k2ym[i,j,k] = numpy.real(k_y[j]**2)
55             k2zm[i,j,k] = numpy.real(k_z[k]**2)
56             xx[i,j,k]=x[i]
57             yy[i,j,k]=y[j]
58             zz[i,j,k]=z[k]
59
60
61 # allocate arrays
62 usquared=numpy.zeros((Nx,Ny,Nz), dtype=float)
63 pot=numpy.zeros((Nx,Ny,Nz), dtype=float)
64 u=numpy.zeros((Nx,Ny,Nz), dtype=complex)
65 una=numpy.zeros((Nx,Ny,Nz), dtype=complex)
66 unb=numpy.zeros((Nx,Ny,Nz), dtype=complex)
67 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
```

```
117     plotnum+=1
118     usquared=abs(u)**2
119     src.mlab_source.scalars = usquared
120     mass=numpy.fft.fftn(usquared)
121     ma=numpy.real(mass[0,0,0])
122     test[plotnum-1]=numpy.log(abs(1-ma/ma0))
123     print(test[plotnum-1])
124     tdata[plotnum-1]=t
125 
126 plt.figure()
127 plt.plot(tdata,test,'r-')
128 plt.title('Time Dependence of Change in Mass')
129 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.

```
1  #!/usr/bin/env python
2  """
3  Numerical solution of the 2D incompressible Navier-Stokes on a
4  Square Domain [0,1]x[0,1] unumpy.sing a Fourier pseudo-spectral method
5  and Crank-Nicolson timestepmath.ping. The numerical solution is compared
       to
6  the exact Taylor-Green Vortex solution of the Navier-Stokes equations
7  
8  Periodic free-slip boundary conditions and Initial conditions:
9      u(x,y,0)=sin(2*pi*x)cos(2*pi*y)
10     v(x,y,0)=-cos(2*pi*x)sin(2*pi*y)
11 Analytical Solution:
12     u(x,y,t)=sin(2*pi*x)cos(2*pi*y)exp(-8*pi^2*nu*t)
13     v(x,y,t)=-cos(2*pi*x)sin(2*pi*y)exp(-8*pi^2*nu*t)
14 """
15 
16 import math
17 import numpy
18 import matplotlib.pyplot as plt
19 from mayavi import mlab
20 import time
21 
22 # Grid
23 N=64; h=1.0/N
24 x = [h*i for i in xrange(1,N+1)]
25 y = [h*i for i in xrange(1,N+1)]
26 numpy.savetxt('x.txt',x)
27 
28 xx=numpy.zeros((N,N), dtype=float)
29 yy=numpy.zeros((N,N), dtype=float)
30 
31 for i in xrange(N):
32     for j in xrange(N):
```

```
33          xx[i,j] = x[i]
34          yy[i,j] = y[j]
35
36
37  dt=0.0025; t=0.0; tmax=0.10
38  #nplots=int(tmax/dt)
39  Rey=1
40
41  u=numpy.zeros((N,N), dtype=float)
42  v=numpy.zeros((N,N), dtype=float)
43  u_y=numpy.zeros((N,N), dtype=float)
44  v_x=numpy.zeros((N,N), dtype=float)
45  omega=numpy.zeros((N,N), dtype=float)
46  # Initial conditions
47  for i in range(len(x)):
48      for j in range(len(y)):
49          u[i][j]=numpy.sin(2*math.pi*x[i])*numpy.cos(2*math.pi*y[j])
50          v[i][j]=-numpy.cos(2*math.pi*x[i])*numpy.sin(2*math.pi*y[j])
51          u_y[i][j]=-2*math.pi*numpy.sin(2*math.pi*x[i])*numpy.sin(2*math.pi
                *y[j])
52          v_x[i][j]=2*math.pi*numpy.sin(2*math.pi*x[i])*numpy.sin(2*math.pi*
                y[j])
53          omega[i][j]=v_x[i][j]-u_y[i][j]
54
55  src = mlab.imshow(xx,yy,omega,colormap='jet')
56  mlab.scalarbar(object=src)
57  mlab.xlabel('x',object=src)
58  mlab.ylabel('y',object=src)
59
60
61  # Wavenumber
62  k_x = 2*math.pi*numpy.array([complex(0,1)*n for n in range(0,N/2) \
63  + [0] + range(-N/2+1,0)])
64  k_y=k_x
65
66  kx=numpy.zeros((N,N), dtype=complex)
67  ky=numpy.zeros((N,N), dtype=complex)
68  kxx=numpy.zeros((N,N), dtype=complex)
69  kyy=numpy.zeros((N,N), dtype=complex)
70
71  for i in xrange(N):
72      for j in xrange(N):
73          kx[i,j] = k_x[i]
74          ky[i,j] = k_y[j]
75          kxx[i,j] = k_x[i]**2
76          kyy[i,j] = k_y[j]**2
77
78  tol=10**(-10)
79  psihat=numpy.zeros((N,N), dtype=complex)
80  omegahat=numpy.zeros((N,N), dtype=complex)
81  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-')
```

```
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

"""
```

```
12
13 import math
14 import numpy
15 from mayavi import mlab
16 import matplotlib.pyplot as plt
17 import time
18
19
20 # Grid
21 Lx=3.0       # Period 2*pi*Lx
22 Ly=3.0       # Period 2*pi*Ly
23 Nx=512       # Number of harmonics
24 Ny=512       # Number of harmonics
25 Nt=200       # Number of time slices
26 tmax=5.0    # Maximum time
27 dt=tmax/Nt   # time step
28 plotgap=10   # time steps between plots
29 Es= 1.0     # focusing (+1) or defocusing (-1) parameter
30 numplots=Nt/plotgap  # number of plots to make
31
32 x = [i*2.0*math.pi*(Lx/Nx) for i in xrange(-Nx/2,1+Nx/2)]
33 y = [i*2.0*math.pi*(Ly/Ny) for i in xrange(-Ny/2,1+Ny/2)]
34 k_x = (1.0/Lx)*numpy.array([complex(0,1)*n for n in range(0,Nx/2) \
35 + [0] + range(-Nx/2+1,0)])
36 k_y = (1.0/Ly)*numpy.array([complex(0,1)*n for n in range(0,Ny/2) \
37 + [0] + range(-Ny/2+1,0)])
38
39 kxm=numpy.zeros((Nx,Ny), dtype=complex)
40 kym=numpy.zeros((Nx,Ny), dtype=complex)
41 xx=numpy.zeros((Nx,Ny), dtype=float)
42 yy=numpy.zeros((Nx,Ny), dtype=float)
43
44
45 for i in xrange(Nx):
46     for j in xrange(Ny):
47         kxm[i,j] = k_x[i]
48         kym[i,j] = k_y[j]
49         xx[i,j] = x[i]
50         yy[i,j] = y[j]
51
52
53 # allocate arrays
54 unew=numpy.zeros((Nx,Ny), dtype=float)
55 u=numpy.zeros((Nx,Ny), dtype=float)
56 uold=numpy.zeros((Nx,Ny), dtype=float)
57 vnew=numpy.zeros((Nx,Ny), dtype=complex)
58 v=numpy.zeros((Nx,Ny), dtype=complex)
59 vold=numpy.zeros((Nx,Ny), dtype=complex)
60 ux=numpy.zeros((Nx,Ny), dtype=float)
61 uy=numpy.zeros((Nx,Ny), dtype=float)
62 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)

```

```
51
52 for i in xrange(Nx):
53     for j in xrange(Ny):
54         for k in xrange(Nz):
55             kxm[i,j,k] = k_x[i]
56             kym[i,j,k] = k_y[j]
57             kzm[i,j,k] = k_z[k]
58             xx[i,j,k]=x[i]
59             yy[i,j,k]=y[j]
60             zz[i,j,k]=z[k]
61
62
63 # allocate arrays
64 unew=numpy.zeros((Nx,Ny,Nz), dtype=float)
65 u=numpy.zeros((Nx,Ny,Nz), dtype=float)
66 uold=numpy.zeros((Nx,Ny,Nz), dtype=float)
67 vnew=numpy.zeros((Nx,Ny,Nz), dtype=complex)
68 v=numpy.zeros((Nx,Ny,Nz), dtype=complex)
69 vold=numpy.zeros((Nx,Ny,Nz), dtype=complex)
70 ux=numpy.zeros((Nx,Ny,Nz), dtype=float)
71 uy=numpy.zeros((Nx,Ny,Nz), dtype=float)
72 uz=numpy.zeros((Nx,Ny,Nz), dtype=float)
73 vx=numpy.zeros((Nx,Ny,Nz), dtype=complex)
74 vy=numpy.zeros((Nx,Ny,Nz), dtype=complex)
75 vz=numpy.zeros((Nx,Ny,Nz), dtype=complex)
76 Kineticenergy=numpy.zeros((Nx,Ny,Nz), dtype=complex)
77 Potentialenergy=numpy.zeros((Nx,Ny,Nz), dtype=complex)
78 Strainenergy=numpy.zeros((Nx,Ny,Nz), dtype=complex)
79 EnKin=numpy.zeros((numplots), dtype=float)
80 EnPot=numpy.zeros((numplots), dtype=float)
81 EnStr=numpy.zeros((numplots), dtype=float)
82 En=numpy.zeros((numplots), dtype=float)
83 Enchange=numpy.zeros((numplots-1),dtype=float)
84 tdata=numpy.zeros((numplots), dtype=float)
85 nonlin=numpy.zeros((Nx,Ny,Nz), dtype=float)
86 nonlinhat=numpy.zeros((Nx,Ny,Nz), dtype=complex)
87
88 u=0.1*numpy.exp(-(xx**2 + yy**2 + zz**2))
89 uold=u
90 v=numpy.fft.fftn(u)
91 vold=numpy.fft.fftn(uold)
92 #src=mlab.contour3d(xx,yy,zz,u,colormap='jet',opacity=0.1,contours=4)
93 src = mlab.pipeline.scalar_field(xx,yy,zz,u,colormap='YlGnBu')
94 mlab.pipeline.iso_surface(src, contours=[u.min()+0.1*u.ptp(), ],
95    colormap='YlGnBu',opacity=0.85)
96 mlab.pipeline.iso_surface(src, contours=[u.max()-0.1*u.ptp(), ],
97    colormap='YlGnBu',opacity=1.0)
98 mlab.pipeline.image_plane_widget(src,plane_orientation='z_axes',
99                                  slice_index=Nz/2,colormap='YlGnBu',
100                                 opacity=0.01)
101 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()
```