The Numerical Solution of Atmospheric Models Describing the Interactions of Inertio-Gravity and Rossby Waves

Mr Graeme W. Wilford

Submitted for the Degree of Doctor of Philosophy

April 7, 1996

Computer Systems Research Group
Dept. of Electrical and Electronic Engineering
University of Surrey
Guildford
Surrey
England
Abstract

This thesis documents three years of work involved in the numerical solution of atmospheric wave models. Derivation of these models is established whilst introducing the basic physical laws governing fluid motion. Numerical techniques are investigated with particular reference to the solution of parabolic and elliptic partial differential equations. Parallel computer systems are discussed and basic concepts introduced with the emphasis placed on distributed virtual parallelism.

The role of inertio-gravity waves under the influence of cyclonic Rossby waves is investigated with respect to the production of atmospheric turbulence. Results from evolving numerical systems bound by various conditions are presented. It is discovered that the wave interaction is not the sole cause of atmospheric blocking as was previously thought. The use of a loosely coupled parallel environment is discussed in relation to potential increases in speed or size of the numerical model. A solution technique is modified to enable such an implementation.

The full nonlinear Barré de Saint-Venant model of fluid motion is solved using a combination of finite difference and spectral methods. Preliminary results are presented and further avenues of investigation are discussed.
Acknowledgements

First and foremost, I'd like to thank Rebecca without whose support this degree would not have been completed. I also acknowledge the support given by my family, friends and colleagues. Special thanks go to Sim, Nusho, Paul, Ian, Tony and Marcel who have kept me sane over the years. I thank the CDCR and especially Roger who has arranged funding for me over the past six months, also the Linux community for giving me the means to work at home. Last but by no means least, thanks go to my mentors, Sasha and Grisha.
Preface

There are three logical parts to this thesis. Part I provides an introduction to the problems that motivated this research and the methods and techniques that were used to resolve them. Specifically, it establishes

1. Physics behind the atmospheric models
2. Technical and computational aspects of the research
3. Numerical solution techniques

Additionally, chapter three contains several implemented examples which go beyond the capacity of an introduction. Their presentation within the introduction is provided as an aid to the reader.

Parts II and III isolate the two independent atmospheric models that were solved during this research. The former describes the numerical methods used, and presents the results obtained for the solution of an aperiodic linear atmospheric wave equation. The later documents the implementation of solution methods to solve a nonlinear system of periodic equations describing incompressible fluid motion. Preliminary results are also included.

All program code written and used during the solution of the two main problems can be found in the Appendix.

©Graeme W. Wilford
# Contents

## I Introduction

### 1 Physical background

1.1 Waves ................................................................. 2
  1.1.1 Physical description ........................................ 2
  1.1.2 Absorption ..................................................... 5
  1.1.3 The diffusion equation .................................... 6

1.2 Equations of fluid motion ....................................... 6
  1.2.1 Continuity equation ........................................ 7
  1.2.2 The Coriolis effect ......................................... 8
  1.2.3 Conservation of momentum ............................ 11
  1.2.4 Energy ......................................................... 13

1.3 The atmosphere .................................................... 14
  1.3.1 Physical description ....................................... 14
  1.3.2 Dispersion relation ........................................ 16
  1.3.3 Atmospheric waves ....................................... 18

1.4 Equations of atmospheric wave motion ...................... 20
  1.4.1 Navier-Stokes ............................................... 20
  1.4.2 Barré de Saint-Venant .................................... 21

1.5 Turbulent inertio-gravity (IG) waves ......................... 22
  1.5.1 2D system ..................................................... 23
  1.5.2 Evolutionary perturbation in 1D ....................... 25

1.6 Rotating nonlinear IG waves .................................... 25
  1.6.1 Nonlinear 2D system ..................................... 25

## 2 Programming and computing ................................. 27

2.1 Language considerations ........................................ 27

2.2 Software ........................................................... 29
  2.2.1 Bundles and libraries ..................................... 29
  2.2.2 Visualisation techniques ............................... 30

2.3 Computer architecture .......................................... 31

2.4 Parallel systems .................................................. 32

2.5 Concurrent programming techniques ......................... 34
  2.5.1 Message passing ........................................... 34
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.2</td>
<td>35</td>
</tr>
<tr>
<td>2.5.3</td>
<td>36</td>
</tr>
<tr>
<td>2.6</td>
<td>36</td>
</tr>
<tr>
<td>2.6.1</td>
<td>37</td>
</tr>
<tr>
<td>2.6.2</td>
<td>38</td>
</tr>
<tr>
<td>2.7</td>
<td>39</td>
</tr>
<tr>
<td>2.7.1</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>43</td>
</tr>
<tr>
<td>3.1</td>
<td>43</td>
</tr>
<tr>
<td>3.1.1</td>
<td>43</td>
</tr>
<tr>
<td>3.1.2</td>
<td>50</td>
</tr>
<tr>
<td>3.2</td>
<td>52</td>
</tr>
<tr>
<td>3.2.1</td>
<td>52</td>
</tr>
<tr>
<td>3.2.2</td>
<td>54</td>
</tr>
<tr>
<td>3.2.3</td>
<td>61</td>
</tr>
<tr>
<td>3.3</td>
<td>63</td>
</tr>
<tr>
<td>3.3.1</td>
<td>64</td>
</tr>
<tr>
<td>3.3.2</td>
<td>66</td>
</tr>
<tr>
<td>3.4</td>
<td>68</td>
</tr>
<tr>
<td>3.4.1</td>
<td>68</td>
</tr>
<tr>
<td>3.4.2</td>
<td>70</td>
</tr>
<tr>
<td>3.5</td>
<td>71</td>
</tr>
<tr>
<td>3.6</td>
<td>74</td>
</tr>
</tbody>
</table>

### II Linear problem

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>77</td>
</tr>
<tr>
<td>4.1</td>
<td>77</td>
</tr>
<tr>
<td>4.2</td>
<td>77</td>
</tr>
<tr>
<td>4.2.1</td>
<td>78</td>
</tr>
<tr>
<td>4.2.2</td>
<td>80</td>
</tr>
<tr>
<td>4.3</td>
<td>81</td>
</tr>
<tr>
<td>4.3.1</td>
<td>81</td>
</tr>
<tr>
<td>4.3.2</td>
<td>85</td>
</tr>
<tr>
<td>4.3.3</td>
<td>86</td>
</tr>
<tr>
<td>4.4</td>
<td>87</td>
</tr>
<tr>
<td>4.4.1</td>
<td>87</td>
</tr>
<tr>
<td>4.4.2</td>
<td>97</td>
</tr>
<tr>
<td>4.5</td>
<td>101</td>
</tr>
<tr>
<td>4.5.1</td>
<td>103</td>
</tr>
<tr>
<td>4.5.2</td>
<td>104</td>
</tr>
<tr>
<td>4.5.3</td>
<td>105</td>
</tr>
</tbody>
</table>
CONTENTS

4.6 Parallel implementation ................................................................. 109
  4.6.1 Example for explicit method .................................................. 109
  4.6.2 ADI ....................................................................................... 110
  4.6.3 Method parallelisation ............................................................ 111
  4.6.4 Speedup realisation .............................................................. 113
  4.6.5 Implementation ................................................................. 114
  4.6.6 Conclusions ................................................................. 115

4.7 Conclusions .................................................................................. 116
  4.7.1 Physical behaviour ............................................................ 116
  4.7.2 Numerical methodology ..................................................... 116
  4.7.3 Parallel implementation ...................................................... 117

III Nonlinear problem ........................................................................ 118

5 IG waves in the presence of 2D turbulence ....................................... 119
  5.1 Physical description .................................................................. 119
  5.2 Numerical description ............................................................ 120
    5.2.1 The velocity vector field .................................................... 120
    5.2.2 Methodology .................................................................. 121
    5.2.3 Managing the model ......................................................... 122
    5.2.4 Initialisation ................................................................. 123
    5.2.5 Evolution ....................................................................... 123
  5.3 Implementation .......................................................................... 127
    5.3.1 Programming style .......................................................... 127
    5.3.2 Restoration of velocity ..................................................... 127
    5.3.3 Time stepping ............................................................... 129
    5.3.4 Nonlinearities ............................................................... 130
  5.4 Results ..................................................................................... 131
    5.4.1 Small scale geopotential perturbation ................................. 131
    5.4.2 Influence of a small scale variability on a large scale pattern 136
  5.5 Validating the results ............................................................... 136
    5.5.1 Conserved properties ....................................................... 141
  5.6 Conclusions ............................................................................. 142
    5.6.1 Physical behaviour .......................................................... 142
    5.6.2 Numerical methodology .................................................. 142
    5.6.3 Parallel implementation .................................................. 143

6 General conclusions ........................................................................ 144
  6.1 Introductory work ..................................................................... 144
  6.2 Numerical methodology ......................................................... 145
  6.3 Linear equation ....................................................................... 145
  6.4 Nonlinear system ...................................................................... 145
  6.5 Achievements .......................................................................... 147
## CONTENTS

### A Program code

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>Code implementing problem posed in Part II</td>
<td>152</td>
</tr>
<tr>
<td>A.1.1</td>
<td>2D ADI solver</td>
<td>152</td>
</tr>
<tr>
<td>A.1.2</td>
<td>1D Crank-Nicolson solver</td>
<td>161</td>
</tr>
<tr>
<td>A.1.3</td>
<td>2D ADI non-homogeneous PDE solver</td>
<td>169</td>
</tr>
<tr>
<td>A.1.4</td>
<td>2D SOR Poisson solver</td>
<td>184</td>
</tr>
<tr>
<td>A.1.5</td>
<td>PVM test programs</td>
<td>190</td>
</tr>
<tr>
<td>A.1.6</td>
<td>2D binary to 1D ASCII data converter</td>
<td>204</td>
</tr>
<tr>
<td>A.1.7</td>
<td>Math library routines</td>
<td>207</td>
</tr>
<tr>
<td>A.1.8</td>
<td>Signal handler library routines</td>
<td>210</td>
</tr>
<tr>
<td>A.1.9</td>
<td>Miscellaneous library routines</td>
<td>213</td>
</tr>
<tr>
<td>A.1.10</td>
<td>UNIRAS visualisation routines</td>
<td>215</td>
</tr>
<tr>
<td>A.2</td>
<td>Code implementing problem posed in Part III</td>
<td>222</td>
</tr>
<tr>
<td>A.2.1</td>
<td>2D explicit solver</td>
<td>222</td>
</tr>
<tr>
<td>A.2.2</td>
<td>2D FFT Poisson solver</td>
<td>235</td>
</tr>
<tr>
<td>A.2.3</td>
<td>Miscellaneous solver routines</td>
<td>246</td>
</tr>
<tr>
<td>A.2.4</td>
<td>2D binary to 2D ASCII data converter</td>
<td>257</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Angular velocity of the Earth in a rotating reference frame .................. 9
1.2 Movement of a wave packet ............................................................... 15

2.1 Memory configuration of parallel computer systems ......................... 33
2.2 Organisation of PVMs virtual machine ............................................... 42

3.1 Domain of dependence with regard to signal propagation speed ........... 46
3.2 Solution domain imposed by data conditions ....................................... 47
3.3 Dependency grid for FTCS method ...................................................... 53
3.4 Stable solution graph of diffusion equation using FTCS method ............ 55
3.5 Unstable solution graph of diffusion equation using FTCS method ......... 55
3.6 BTCS dependence grid ................................................................. 56
3.7 Solution graph of diffusion equation using BTCS method .................... 57
3.8 Crank-Nicolson dependence grid ...................................................... 60
3.9 Solution graph for nonlinear diffusion equation .................................. 63
3.10 Iteration sweep strategy for Poisson solution technique ..................... 69
3.11 Dependency grid for a general 1D leap frog method ........................ 74

4.1 Restricted movement of high frequency wave components .................. 79
4.2 Component by component solution domain sweeping ........................... 86
4.3 Amplitude of IG wave packet at the centre of the Rossby well ............... 89
4.4 Form of IG wave packet inside the cyclone ....................................... 89
4.5 The IG wave packet inside the cyclone at t = 0 .................................. 90
4.6 The IG wave packet inside the cyclone at t = 1 .................................. 90
4.7 The IG wave packet inside the cyclone at t = 2 .................................. 91
4.8 The IG wave packet inside the cyclone at t = 3 .................................. 91
4.9 The IG wave packet inside the cyclone at t = 4 .................................. 92
4.10 The IG wave packet inside the cyclone at t = 5 .................................. 92
4.11 The IG wave packet inside the anticyclone at t = 0 ............................ 93
4.12 The IG wave packet inside the anticyclone at t = 1 ............................ 93
4.13 The IG wave packet inside the anticyclone at t = 2 ............................ 94
4.14 The IG wave packet inside the anticyclone at t = 3 ............................ 94
4.15 The IG wave packet inside the anticyclone at t = 4 ............................ 95
4.16 The IG wave packet inside the anticyclone at t = 5 ............................ 95
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.17</td>
<td>Form of IG wave packet near the cyclone</td>
<td>96</td>
</tr>
<tr>
<td>4.18</td>
<td>Form of IG wave packet near the anticyclone</td>
<td>96</td>
</tr>
<tr>
<td>4.19</td>
<td>Form of IG wave packet over a cyclone-anticyclone dipole</td>
<td>97</td>
</tr>
<tr>
<td>4.20</td>
<td>The IG wave packet over the cyclone-anticyclone dipole at $t = 0$</td>
<td>98</td>
</tr>
<tr>
<td>4.21</td>
<td>The IG wave packet over the cyclone-anticyclone dipole at $t = 1$</td>
<td>98</td>
</tr>
<tr>
<td>4.22</td>
<td>The IG wave packet over the cyclone-anticyclone dipole at $t = 2$</td>
<td>99</td>
</tr>
<tr>
<td>4.23</td>
<td>The IG wave packet over the cyclone-anticyclone dipole at $t = 3$</td>
<td>99</td>
</tr>
<tr>
<td>4.24</td>
<td>The IG wave packet over the cyclone-anticyclone dipole at $t = 4$</td>
<td>100</td>
</tr>
<tr>
<td>4.25</td>
<td>The IG wave packet over the cyclone-anticyclone dipole at $t = 5$</td>
<td>100</td>
</tr>
<tr>
<td>4.26</td>
<td>Interaction between IG and evolutionary Rossby wave packets</td>
<td>102</td>
</tr>
<tr>
<td>4.27</td>
<td>Effect of $\omega$ on iterated $N^M$</td>
<td>108</td>
</tr>
<tr>
<td>4.28</td>
<td>Geometric data partitioning of a 2D explicit method</td>
<td>110</td>
</tr>
<tr>
<td>4.29</td>
<td>Data transfer at sweep direction changeover</td>
<td>111</td>
</tr>
<tr>
<td>4.30</td>
<td>Data transfer for a given number of nodes</td>
<td>112</td>
</tr>
<tr>
<td>5.1</td>
<td>Physical invariants, energy ($\mathcal{E}$) and enstrophy ($\mathcal{N}$)</td>
<td>132</td>
</tr>
<tr>
<td>5.2</td>
<td>The energy ($u^2$) of the wave packet at $t = 0.4$</td>
<td>132</td>
</tr>
<tr>
<td>5.3</td>
<td>The energy ($u^2$) of the wave packet at $t = 0.8$</td>
<td>133</td>
</tr>
<tr>
<td>5.4</td>
<td>The energy ($u^2$) of the wave packet at $t = 1.2$</td>
<td>133</td>
</tr>
<tr>
<td>5.5</td>
<td>Geopotential field ($\phi$) at $t = 0.0$</td>
<td>134</td>
</tr>
<tr>
<td>5.6</td>
<td>Geopotential field ($\phi$) at $t = 0.4$</td>
<td>134</td>
</tr>
<tr>
<td>5.7</td>
<td>Geopotential field ($\phi$) at $t = 0.8$</td>
<td>135</td>
</tr>
<tr>
<td>5.8</td>
<td>Geopotential field ($\phi$) at $t = 1.2$</td>
<td>135</td>
</tr>
<tr>
<td>5.9</td>
<td>Physical invariants, energy ($\mathcal{E}$) and enstrophy ($\mathcal{N}$)</td>
<td>137</td>
</tr>
<tr>
<td>5.10</td>
<td>The energy ($u^2$) of the wave packet at $t = 0.0$</td>
<td>137</td>
</tr>
<tr>
<td>5.11</td>
<td>The energy ($u^2$) of the wave packet at $t = 0.4$</td>
<td>138</td>
</tr>
<tr>
<td>5.12</td>
<td>The energy ($u^2$) of the wave packet at $t = 0.8$</td>
<td>138</td>
</tr>
<tr>
<td>5.13</td>
<td>The energy ($u^2$) of the wave packet at $t = 1.2$</td>
<td>139</td>
</tr>
<tr>
<td>5.14</td>
<td>Geopotential field ($\phi$) at $t = 0.0$</td>
<td>139</td>
</tr>
<tr>
<td>5.15</td>
<td>Geopotential field ($\phi$) at $t = 0.4$</td>
<td>140</td>
</tr>
<tr>
<td>5.16</td>
<td>Geopotential field ($\phi$) at $t = 0.8$</td>
<td>140</td>
</tr>
<tr>
<td>5.17</td>
<td>Geopotential field ($\phi$) at $t = 1.2$</td>
<td>141</td>
</tr>
<tr>
<td>6.1</td>
<td>Methodology chart</td>
<td>146</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Characterization of PDE by discriminant .................................................... 45
3.2 Contrasting explicit and implicit methods .................................................... 48
4.1 Solution and transfer speeds using PVM ..................................................... 113
5.1 Leapfrog time integration using two storage levels ....................................... 124
5.2 Standard time integration using two storage levels ....................................... 124
5.3 Leapfrog time integration using three storage levels ....................................... 124
Part I

Introduction
Chapter 1

Physical background

This chapter provides a general introduction to some of the physics and physical phenomena behind this work. Although no direct research was carried out in this field, an investigation of the underlying processes was appropriate and is included here.

1.1 Waves

1.1.1 Physical description

A wave may take many different forms [38]:

- Simple periodic waves formed by a plucked guitar string.
- Solitary pulse waves such as a tidal wave.
- Electromagnetic waves consisting of a frequency modulated carrier, radio waves.
- Atmospheric waves of planetary scale.

These phenomena are defined as wave-like due to their ability to transmit energy from place to place without a corresponding transmission of matter.

The wave equation

An equation is sought, whose solutions can describe simple wave motion. Let the model be restricted to the following simplified wave description: A wave is a disturbance or parcel of energy propagating with constant velocity and unchanging form. As the wave is unchanging, energy is not dissipated and it can be considered ideal. To further constrain the model, motion is to be considered in a single dimension \((x)\) only.
Consider a wave propagating in the $x$ direction with velocity $u$. Upon this system, a moving frame of reference, $\varphi$, is superimposed, travelling with velocity $u$ in a direction parallel to the $x$ axis. If $\varphi = x$ at time zero,

$$\varphi = x - ut.$$ 

Due to the limitations imposed upon the model, an observer travelling with the moving reference will view a static disturbance. Thus, the wave has a dependence on time that is a function of $\varphi$ only. The wave, $\psi$, can be described by

$$\psi(x, t) = f(\varphi) = f(x - ut). \quad (1.1)$$

Consider another moving frame of reference, $\varphi$, travelling with velocity $-|u|$ in a direction parallel to $x$. In this case, a wave travelling with $\varphi$ could be described by

$$\psi(x, t) = g(\varphi) = g(x + |u|t) \quad (1.2)$$

where $f$ in (1.1) and $g$ in (1.2) are both arbitrary functions of time and space.

A partial differential equation (PDE) that can be satisfied by both waveforms, $f$ and $g$ would, by definition, describe the simple wave motion outlined above. Functions (1.1) and (1.2) combine to form

$$\frac{\partial \varphi}{\partial x} = \frac{\partial \varphi}{\partial x} = 1$$

and

$$\frac{\partial \varphi}{\partial t} = -\frac{\partial \varphi}{\partial t} = -u.$$ 

First order differential motions of $\psi$ through space and time, respectively, are described by

$$\frac{\partial \psi}{\partial x} = \frac{\partial \varphi}{\partial \varphi} + \frac{\partial \psi}{\partial \varphi}, \quad (1.3)$$

$$\frac{\partial \psi}{\partial t} = u \left( \frac{\partial \varphi}{\partial \varphi} - \frac{\partial \varphi}{\partial \varphi} \right). \quad (1.4)$$
CHAPTER 1. PHYSICAL BACKGROUND

Using the former case where $\psi(x, t) = f(\phi)$, (1.3) and (1.4) reduce to

$$\frac{\partial \psi}{\partial t} = -u \frac{\partial \psi}{\partial x}. \tag{1.5}$$

However, in the later case where $\psi(x, t) = g(\phi)$, they reduce to

$$\frac{\partial \psi}{\partial t} = u \frac{\partial \psi}{\partial x}. \tag{1.6}$$

Clearly, (1.5) and (1.6) do not describe the same wave motion. Hence, (1.3) and (1.4) cannot be used to form a wave equation satisfying the predefined conditions. Second order differential motion of $\psi$ gives:

$$\frac{\partial^2 \psi}{\partial x^2} = \frac{\partial^2 \psi}{\partial \phi^2} + 2\frac{\partial^2 \psi}{\partial \phi \partial \theta} + \frac{\partial^2 \psi}{\partial \theta^2}, \tag{1.7}$$

$$\frac{\partial^2 \psi}{\partial t^2} = u^2 \left( \frac{\partial^2 \psi}{\partial \phi^2} - 2\frac{\partial^2 \psi}{\partial \phi \partial \theta} + \frac{\partial^2 \psi}{\partial \theta^2} \right). \tag{1.8}$$

When $\psi(x, t)$ is equal to either $f(\phi)$ or $g(\theta)$, equations (1.7) and (1.8) reduce to

$$\frac{\partial^2 \psi}{\partial t^2} = u^2 \frac{\partial^2 \psi}{\partial x^2}. \tag{1.9}$$

This also holds for any linear combination of $f$ and $g$, e.g. $\psi(x, t) = \alpha f(\phi) + \beta g(\theta)$. Equation (1.9) is known as the one-dimensional wave equation. It is able to describe the motion of any linear wave, $\psi$ through one-dimensional space and time.

To generalise the wave equation, $u$ may be considered as any slow function of $x$ and/or $t$ provided that

$$\frac{1}{u} \frac{\partial u}{\partial x} \ll \frac{1}{\lambda},$$

$$\frac{1}{u} \frac{\partial u}{\partial t} \ll \frac{1}{T}$$

where $\lambda$ is the wavelength and $T$, the period of the wave. This can occur if the medium of wave transport has non-uniform density. The generalisation spoils the simple description of wave motion used to produce (1.9) and also allows more general wave motions to be described. Using vector notation and introducing multi-dimensional wave motion, (1.9) becomes

$$\frac{\partial^2 \psi}{\partial t^2} = \nu^2 \nabla^2 \psi.$$
where $\nabla^2$ represents the Laplacian operator. In two dimensions,

$$\nabla^2 = \left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right).$$

1.1.2 Absorption

Due to its definition, the model for the simple wave equation (1.9) neglects any dissipation of energy from the wave as it travels. One particular form of energy transfer is absorption by the transport medium. This form of dissipation is caused by the inherent physical properties of the medium involved. As the wave motion proceeds, its kinetic energy is diminished. Examples of energy loss are friction caused by viscosity and electromagnetic currents due to conductivity. Other factors may also contribute to dissipation or absorption of energy by a wave depending upon wave type and physical conditions.

Electromagnetic conduction

Simple cases of absorption are modelled by inclusion into the wave equation of a first order partial derivative with respect to time. An example of such an equation is derived in [38] from the physical interpretation of electromagnetic wave motions within a conducting medium:

$$\frac{\partial^2 \psi}{\partial x^2} = \frac{1}{u^2} \left( \frac{\partial^2 \psi}{\partial t^2} + 2\gamma \frac{\partial \psi}{\partial t} \right). \tag{1.10}$$

Wave motion causes currents to be formed in the medium that dissipate the energy of the wave at a rate determined by the medium's conductivity, $\sigma$, and its dielectric constant, $\varepsilon$. Equation (1.10) describes the wave motion produced under such conditions where $\gamma = 2\pi\sigma/\varepsilon$.

In a medium such as free space where the conductivity is effectively zero, the dissipative term in (1.10) vanishes and the electromagnetic wave motion approximates the ideal wave equation. Conversely, if the transmission medium is highly conductive,

$$\frac{\partial^2 \psi}{\partial t^2} \ll \frac{\partial \psi}{\partial t}$$

and (1.10) will approximate the diffusion equation.
1.1.3 The diffusion equation

Consider a rod of uniform cross section, \( A \), with the physical properties: density, \( \rho \), specific heat, \( C \), and thermal conductivity, \( k \). One end of the rod is heated to produce a non-uniform temperature gradient along the length of the rod which runs parallel to the \( x \) axis. The relationship between the flow of heat into the bar, \( Q \), and the temperature along the length of the bar, \( T \), is given by Fourier’s law as

\[
\frac{\partial Q}{\partial t} = -kA \frac{\partial T}{\partial x}.
\]

Heat will flow along the bar in an attempt to produce a uniform temperature gradient throughout its length. The net flow of heat into an element of bar, \( \delta x \), is

\[
\frac{\partial Q(x)}{\partial t} - \frac{\partial Q(x+\delta x)}{\partial t} = kA \frac{\partial^2 T}{\partial x^2} \delta x.
\]

The corresponding rate of temperature change is given as

\[
\frac{\partial Q(x)}{\partial t} - \frac{\partial Q(x+\delta x)}{\partial t} = C \rho A \frac{\partial T}{\partial t} \delta x.
\]

Equation (1.11) with (1.12) forms a one dimensional description of heat conduction:

\[
\frac{\partial^2 \psi}{\partial x^2} = \kappa \frac{\partial \psi}{\partial t}
\]

where \( \kappa = C \rho / k \). Equation (1.13) is known as the 1D conduction or diffusion equation.

1.2 Equations of fluid motion

The equations of motion derived from Newton’s laws underpin fluid motion and wave theory. They are shown for compressible fluids before being reduced to describe incompressible fluid motion.

The atmosphere, or more specifically, the levels of atmosphere concerned with meteorological activity can be described by incompressible models provided that the motion of the fluid is considerably less than the speed of sound. In the areas of interest covered by this research, that is always the case. By definition, this assumption completely removes acoustic waves from meteorological wave behaviour, allowing simpler models to be built. These include the Navier-Stokes and Barré de Saint-Venant (shallow water) equations of atmospheric wave motion. The descriptive phrase “shallow water” can be
applied to any fluid body whose surface pressure distribution is hydrostatic.

The hydrodynamic theories and equations established below can be found in [1, 23, 27] from which further detail may be gained.

### 1.2.1 Continuity equation

Consider a cubic volume of size \( dx \, dy \, dz \). Let \( u, v, w \) represent velocity components in the \( x, y, z \) directions, respectively. The flux of some property, \( \zeta \), passing through a surface of area \( dy \, dz \) is defined by

\[
(u \zeta) \, dy \, dz. \tag{1.14}
\]

Flux leaving through the opposite parallel surface amounts to

\[
(u \zeta + \frac{\partial (u \zeta)}{\partial x} \, dx) \, dy \, dz. \tag{1.15}
\]

The net flux gained through the volume along the \( x \) axis is given by the difference of (1.14) and (1.15):

\[
-\frac{\partial (u \zeta)}{\partial x} \, dx \, dy \, dz.
\]

Let the property \( \zeta \) be conservative. Applying the method above to each coordinate direction in turn produces a general balanced equation with respect to the change in \( \zeta \) over time in the volume \( dx \, dy \, dz \):

\[
\frac{\partial \zeta}{\partial t} \, dx \, dy \, dz = - \left( \frac{\partial (u \zeta)}{\partial x} + \frac{\partial (u \zeta)}{\partial y} + \frac{\partial (u \zeta)}{\partial z} \right) \, dx \, dy \, dz. \tag{1.16}
\]

Introducing the concept of divergence and the vector operator,

\[
\vec{\nabla} = \left( \frac{\partial}{\partial x} + \frac{\partial}{\partial y} + \frac{\partial}{\partial z} \right),
\]

equation (1.16) written in compact vector notation becomes

\[
\frac{\partial \zeta}{\partial t} + \vec{\nabla} \cdot \zeta \vec{v} = 0 \tag{1.17}
\]
where the velocity vector, $\vec{v} = (u, v, w)$. Equation (1.17) holds for any conserved property, however defining a specific property, mass, results in the continuity equation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \rho \vec{v} = 0$$

(1.18)

where $\rho \vec{v} = mn\vec{v}$ represents the flux with $m$ denoting individual particle mass, $n$ the number of particles.

Incompressibility

Stokes operator, $D/Dt$, denotes a differentiation following the motion of a fluid:

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + \vec{v} \cdot \nabla$$

Hence, (1.18) may be rewritten as

$$\frac{1}{\rho} \frac{D\rho}{Dt} + \nabla \cdot \vec{v} = 0.$$  

Incompressibility is a consequence of constant density where

$$\frac{D\rho}{Dt} = 0.$$  

Hence, the continuity equation for an incompressible fluid is

$$\nabla \cdot \vec{v} = 0.$$  

(1.19)

1.2.2 The Coriolis effect

To allow definition of the conservation of momentum equation, (§1.2.3), a digression into the cause of the Coriolis effect [2, 17] is given here, however it is not an equation of fluid motion.

Rotation of the Earth

The Earth rotates about its polar axis in an easterly direction with an angular velocity, $\Omega$:

$$\Omega = \frac{2\pi}{d_s} = 0.7292 \times 10^{-4} \text{rad/sec}$$
CHAPTER 1. PHYSICAL BACKGROUND

In general, the atmosphere\(^1\) rotates with the Earth such that a particle observed as motionless from the surface of the Earth has components of absolute acceleration due to this rotation. The fictitious centrifugal force created by this centripetal acceleration is known as the Coriolis force. Motion created by the Coriolis force varies with latitude. This is due to the sphericity of the Earth causing variations of distance, \(r\), from the axis of rotation and hence, a variation in the rotation velocity, \(v\): \(|v| = \Omega r\). It follows that the magnitude of velocity is greatest where \(r\) is at a maximum: at the Equator.

**Coriolis force**

The Coriolis force per unit mass, \(f\), is defined as

\[
f = -2\vec{\Omega} \times \vec{v}
\]

where \(\vec{\Omega}\) is the angular velocity vector in parallel with the axis of rotation. Fig. 1.1 shows how the components of \(\vec{\Omega}\) are derived in a rotating reference frame, positioned

---

\(^1\)also applies to shallow water
at the Earth's surface. The $x$, $y$ and $z$ axes represent east, north and perpendicular to the surface of the Earth, respectively. Individual components of $\vec{\Omega}$ are

$$
\begin{align*}
\Omega_x &= 0, \\
\Omega_y &= \Omega \cos \theta, \\
\Omega_z &= \Omega \sin \theta.
\end{align*}
$$

The velocity created by the force is almost horizontal with respect to the Earth's surface, hence the vertical component of velocity, $v$, may be ignored, $\vec{v} = (u, v, 0)$. Using this simplification, $f$ is resolved into coordinate directions as

$$
\begin{align*}
f_x &= 2\Omega_x v, \\
f_y &= -2\Omega_y u, \\
f_z &= 2\Omega_z u.
\end{align*}
$$

Letting the $xy$ reference frame rotate such that $x$ assumes the direction of fluid flow further restricts the velocity vector to $\vec{v} = (u, v, 0)$. The $z$ component of the reference frame remains unchanged. In this new coordinate system, $\Omega_x$ and $\Omega_y$ can both be non-zero but the $x$ component of $f$ vanishes completely. Hence, the Coriolis force is a deflecting force only, perpendicular to the fluid flow.

**2D approximation**

In fluid masses such as the atmosphere and shallow water, the vertical component of the Coriolis force is much less than that produced by gravity and may be ignored. Consequently, the force may be approximated by

$$
\begin{align*}
f &= -2\Omega_x \times \vec{v} \\
   &= -2\Omega \sin \theta \times \vec{v}.
\end{align*}
$$

It is common for $f$ to denote the Coriolis parameter rather than the force itself. Using this notation,

$$
2\vec{\Omega} \times \vec{v} = f_n \times \vec{v}
$$

(1.21)
where $f = 2\Omega_z$. Resolving $f_n \times \vec{v}$ produces horizontal components equivalent to those found above:

$$f_x = f v,$$
$$f_y = -f u.$$ 

This notation is used in two dimensional models of fluid flow such as that defined by the shallow water equations, shown later. It is not valid for fluids of arbitrary depth where (1.20) must be used.

### 1.2.3 Conservation of momentum

Newton's second law:

"The rate of change of momentum is proportional to the applied force and takes place in the direction in which the force acts"

can be used as the basis for the conservation of momentum equation:

$$\vec{F} = \frac{\partial \vec{M}}{\partial t} = \delta m \frac{D\vec{v}}{Dt} = \rho \delta V \frac{D\vec{v}}{Dt}$$

(1.22)

where $\vec{M}$ represents momentum and $V$, volume. $\vec{F}$ can be separated into body forces, $\vec{F}_b$, and surface forces, $\vec{F}_s$. The dominating forces acting upon the body of a fluid particle are gravity and the Coriolis force. Approximations neglecting the Coriolis force exist, indeed Euler's equation had no Coriolis component, however to produce an equation of sufficient generality, it is included here. Body forces are represented by

$$\vec{F}_b = \delta m \vec{g} - 2\delta m \vec{\Omega} \times \vec{v}.$$  

(1.23)

In a compressible fluid with a constant coefficient of viscosity, a fluid particle will have surface forces originating from surface pressure, $p$ and viscosity, $\mu$. Other dissipative or additive forces generated by phenomena such as solar heating are ignored, but should be included if their contribution is not negligible. Surface forces are represented by

$$\vec{F}_s = \left( \nabla p + \mu \left[ \nabla^2 \vec{v} + \frac{1}{3} \vec{\nabla} (\vec{\nabla} \cdot \vec{v}) \right] \right) \delta V.$$  

(1.24)
CHAPTER 1. PHYSICAL BACKGROUND

The viscous force in (1.24) is an approximation of the full viscous force found in [2]:

$$\left( \mu \left[ \frac{4}{3} \nabla \left( \nabla \cdot \vec{v} \right) - \nabla \times \left( \nabla \times \vec{v} \right) \right] \right) \delta V.$$ 

However, the approximation uses a vector identity that is valid in the Cartesian coordinate system used here. Combining (1.22) with the forces (1.23) and (1.24) produces the conservation of momentum equation for a compressible fluid:

$$\frac{DV}{DT} = \nabla \cdot \vec{f} - \frac{1}{\rho} \nabla p + \eta \left[ \nabla^2 \vec{v} + \frac{1}{3} \nabla \left( \nabla \cdot \vec{v} \right) \right]$$ (1.25)

where $\eta = \mu / \rho$ is known as kinematic viscosity.

**Incompressibility**

For an incompressible fluid, the viscous term in (1.25) is simplified, reducing the equation to

$$\frac{DV}{DT} = \nabla \cdot \vec{f} - 2 \nabla \times \vec{v} - \frac{1}{\rho} \nabla p + \eta \nabla^2 \vec{v}.$$ (1.26)

In the middle of the 18th century, Euler derived a similar equation regarding the conservation of momentum,

$$\frac{DV}{DT} = \nabla \cdot \vec{f} = \frac{1}{\rho} \nabla p.$$ 

Known as Euler's equation, it was considered for a system with a fixed reference frame and took no account of viscous dissipation. In the early part of the 19th century, Navier added the missing viscous term which later contributed to the Navier-Stokes equations.

**Bernoulli’s theorem**

Use of the momentum conservation equation allowed Bernoulli to show that under specific conditions, certain fluid properties remain constant following the motion of the fluid.

The conservative force produced by gravity may be expressed as the gradient of a potential, $\Phi$, otherwise known as the geopotential:

$$\nabla \Phi = -\vec{g}.$$
Using the vector identity
\[
(\vec{\nabla} \times \vec{v}) \times \vec{v} \equiv (\vec{v} \cdot \vec{\nabla}) \vec{v} - \vec{\nabla} \frac{v^2}{2}
\] (1.27)
and defining vorticity, \( \omega \), relative to the motion of the Earth as
\[
\vec{\omega} = \vec{\nabla} \times \vec{v},
\] (1.28)
equation (1.26) can be rewritten as
\[
\frac{\partial \vec{v}}{\partial t} + (\vec{\omega} + 2\vec{\Omega}) \times \vec{v} = -\vec{\nabla} \left( \frac{p}{\rho} + \Phi + \frac{v^2}{2} \right) + \eta \nabla^2 \vec{v}.
\] (1.29)
The absolute vorticity, \( \vec{\omega} + 2\vec{\Omega} \), is a measure of the vorticity from an absolute reference frame that is still with regard to the Earth’s rotation.

In an irrotational inviscid fluid, the kinematic viscosity and absolute vorticity vanish. Integrating (1.29) under these conditions gives Bernoulli’s equation:
\[
\frac{\partial \phi}{\partial t} + \frac{p}{\rho} + \Phi + \frac{v^2}{2} = C
\]
where the velocity potential \( \phi \) is defined by \( \vec{v} = \vec{\nabla} \phi \) and \( C \) is a constant.

Alternatively, considering a steady inviscid fluid where
\[
\frac{\partial \vec{v}}{\partial t} = 0 \quad \text{and} \quad \frac{D}{Dt} = \vec{v} \cdot \vec{\nabla},
\]
the scalar product of (1.29) with \( \vec{v} \) gives
\[
\frac{D}{Dt} \left( \frac{p}{\rho} + \Phi + \frac{v^2}{2} \right) = 0.
\] (1.30)
Under these conditions, the bracketed quantity in (1.30) is invariant along a streamline, that is, following the motion of the fluid.

1.2.4 Energy

Although models of incompressible fluid motion are fully described by a combination of the continuity and momentum equations derived above, compressible fluid motions usually require an equation of energy conservation to form a complete system. For incompressible systems, an energy equation can provide useful information regarding any
approximations or errors in the system and the extent of their effect. The conservation of potential enstrophy can also be used in a similar manner.

Kinetic energy

Taking the scalar product of (1.29) with \( \rho \vec{v} \) and manipulating the result gives (from [27])

\[
\frac{\partial}{\partial t} \left( \frac{1}{2} \rho v^2 \right) = -\nabla \cdot \left[ \rho \left( \frac{1}{2} \rho v^2 + p \right) \right] + \frac{1}{2} \mu \nabla^2 v^2 + p \nabla \cdot \vec{v} - \rho \vec{v} \cdot \nabla \Phi. \tag{1.31}
\]

Equation (1.31) equates the rate of change of bulk kinetic energy (KE) per unit volume to the divergence of KE flux combined with the dissipation and work of pressure and external forces.

The first two terms on the right hand side (RHS) of (1.31) combine to form the KE flux vector. These are, respectively, the energy flux produced by the total dynamic and static pressure and the molecular flux of bulk KE. The third terms represents the work of pressure during expansion and the last represents the flux of potential energy (PE). A PE flux is caused by the work of gravity during vertical displacements.

1.3 The atmosphere

1.3.1 Physical description

The Earth’s atmosphere is described as being a compressible rotating spherical fluid permeated by density and temperature gradients [2]. However, various assumptions, conditions and approximations can be used to reduce the complexity of such a medium to that described by a barotropic non-divergent\(^2\) model [26]. Such a model is the simplest description of the atmosphere able to produce Rossby waves [39].

Over twenty different named types of wave are known to exist in the atmosphere. Physically, they may all be described by at least one of the three general classes of atmospheric wave: vertical transverse, horizontal transverse and longitudinal.

Transverse waves propagate horizontally. Vertical transverse waves propagate with vertical displacements whereas horizontal transverse waves propagate with horizontal displacements, perpendicular to the direction of motion. Longitudinal waves displace fluid along the direction of motion and are otherwise known as acoustic waves. Longitudinal waves arise out of the compressibility of a fluid and so an assumption of

\(^2\)homogeneous, incompressible, inviscid, purely horizontal motion
incompressibility effectively removes all acoustic waves from the atmosphere. In the context of numerical solutions to atmospheric based models of fluid flow, the removal of acoustic waves can greatly simplify the system, thereby reducing the level of computation required.

Turbulence

Atmospheric waves may exist as small perturbations on an otherwise steady atmospheric state. In this form, they follow linear paths described by linear equations and may superimpose without affecting each other. In contrast, large perturbations can and do exist which may become erratic, nonlinear in behaviour and may interact constructively or destructively with one another or with one’s self. The behaviour of atmospheric waves may be influenced by winds, air masses, fronts, the rotation of the Earth or other meteorological phenomena. Ultimately, waves may break and cause turbulence by releasing their energy to their surroundings. A characteristic of turbulence is its non-deterministic or chaotic behaviour. Interest in the causes of such turbulence [10, 11, 14, 15] has provided the mathematical models which are subsequently investigated by this research.

Group and phase velocities

Electromagnetic waves can originate from either man-made or natural sources. Man-made atmospheric waves may approximate an infinite wave train but all known meteorologically based waves are finite. These finite wave trains are known as packets. Two velocities are associated with wave packets: group velocity and individual phase velocities. The phase velocity, \( c \), is the velocity associated with a wave of well defined
frequency and wavelength. It is shown in fig. 1.2 and is given by:

$$ c = \frac{\omega}{k} $$

(1.32)

where $\omega$ is the angular frequency and $k$, the wavenumber.

Group velocity describes the propagation of energy by the wave packet, showing how the group as a whole is moving. It is defined as

$$ \vec{v} = \left( \frac{\partial \omega}{\partial k_x}, \frac{\partial \omega}{\partial k_y}, \frac{\partial \omega}{\partial k_z} \right) $$

with $\vec{k}$ representing a wave vector, $2\pi/\lambda$, in the respective coordinate direction.

If an observer is placed adjacent to a wave, the group velocity of the wave is the velocity with which the observer must travel to remain abreast with the envelope. In this position, the packet will appear as a standing wave if the phase velocity is identical to the group velocity. If not, the wave is dispersive.

**Dispersion**

If the group and phase velocities of a wave packet are unequal, the wave packet is dispersive. The majority of waves encountered in the atmosphere are dispersive. In such waves, angular frequency is dependent on the wavelength components of the wave: $\omega = \omega(k)$. As such, each wavelength component of the wave packet may have its own distinct phase velocity. Physically, this means that the shape of a dispersive wave packet will change as it propagates. It will contain waves of different wavelengths travelling with different velocities. The effect of this is that the wave will spread out or disperse. Furthermore, a reduction in energy density will occur within the wave packet as a whole.

**1.3.2 Dispersion relation**

All dispersive waves are associated with a dispersion relation which equates the angular frequency of a wave to a function of the coordinate wavenumbers. For instance, gravity waves are described by the dispersion relation:

$$ \omega^2 = gH_0(k_x^2 + k_y^2). $$

(1.33)
Using (1.32) with (1.33) provides a definition of the phase velocity of gravity waves:

\[ c = \sqrt{\frac{g}{H_0}} \]  

Equation (1.34) shows that \( c \) is dependent solely on the mean surface height, \( H_0 \).

Dispersion relations are derived from the equations of wave motion for a particular atmospheric model. These equations, written in terms of velocity differentials, are subjected to a Fourier transform, \( \hat{v} \rightarrow \hat{k} \).

The dispersion relation produced may then describe various wave types, which exist independently under various conditions. For instance, if the wavelengths are assumed to be of sufficient size, the Coriolis effect may be significant. This will describe waves with a particular characteristic that is entirely missing from waves with shorter wavelengths that may exist in a model using the same dispersion relation. An example of this is shown below for the wave model described by the shallow water equations. The dispersion relation describes gravity, inertia-gravity and Rossby waves depending upon the conditions imposed upon the model.

**Barré de Saint-Venant model**

The model of fluid motion is based upon the shallow water equations \[43\] that are derived later in this chapter. Ignoring the effect of viscosity, these are

\[
\begin{align*}
\frac{Du}{Dt} & = -g \frac{\partial h}{\partial x} + f v, \\
\frac{Dv}{Dt} & = -g \frac{\partial h}{\partial y} - f u, \\
\frac{Dh}{Dt} & = -h \nabla \cdot \vec{v}
\end{align*}
\]  

where

\[ f = f_0 + \beta y, \quad \beta = \frac{\partial f_0}{\partial y} \]

with \( f_0 = 2\Omega \) representing the Coriolis parameter. Surface height is represented by \( h \) and \( \nabla^2 \) is two dimensional.

To derive the dispersion relation from this model, the system (1.35) is combined into a single equation. Various substitutions and cross-differentiations finally produce
CHAPTER 1. PHYSICAL BACKGROUND

a third order differential:

\[ \left( \frac{\partial^3}{\partial t^3} + (f^2 - gh \nabla^2) \frac{\partial}{\partial t} - \beta g \frac{\partial}{\partial x} \right) v = 0. \]  \hspace{1cm} (1.36)

Using Fourier transforms, (1.36) becomes

\[ \omega^3 - (c^2 k^2 + f^2)\omega - c^2 \beta k_x = 0 \]  \hspace{1cm} (1.37)

where \( c \) is defined in (1.34) as the phase velocity of long, surface-gravity waves.

Controlling the conditions and the magnitude of terms in (1.37) illustrate the nature of various atmospheric waves with different wavelengths. Where \( \omega^3 \gg c^2 \beta k_x \), (1.37) becomes

\[ \omega^2 = c^2 k^2 + f^2 \]  \hspace{1cm} (1.38)

which describes the motion of inertio-gravity (IG) waves. Surface waves of this type that have a characteristic length less than the external Rossby radius of deformation: \( r_E = \frac{2}{f} \) do not feel the effect of the Coriolis force and can be described by

\[ \omega^2 = c^2 k^2. \]  \hspace{1cm} (1.39)

This is equivalent to (1.33), describing surface gravity waves. Where \( \omega^3 \ll c^2 \beta k_x \), (1.37) becomes

\[ \omega = \frac{-\beta k_x}{\left( k^2 + \frac{r^2}{k_x} \right)} \]  \hspace{1cm} (1.40)

which describes Rossby waves.

1.3.3 Atmospheric waves

Gravity waves

Gravity waves are vertically transverse in nature and exist due to the density variation in the Earth’s atmosphere. In turn, this is caused by the gravitational force of the Earth’s mass. The density variation is a form of stratification and is responsible for the horizontally limited motion displayed by the majority of naturally occurring atmospheric waves. Subsequently, gravity waves provide a stabilizing force within the atmosphere. Indeed, due to the conservation of energy, vertical wave motion has a
severe effect on the amplitude of a wave packet. The kinetic energy of a wave is defined

\[ E_k = \rho \frac{A^2}{2} \]

where \( \rho \) is the density of the fluid and \( A \), the amplitude of the wave. In the atmosphere \( \rho \) decreases at an almost exponential rate with altitude. Since this has the effect of exponential changes in wave magnitude with height, any downward group propagation of a wave packet is evanescent. Conversely, this effect accounts for the very large amplitude waves that are often found at very high altitudes.

The existence of gravity waves is brought about by the restoring force of the density gradient. They occur when this force is the cause of the most dominant motion in the atmosphere. Gravity waves may become excited or unstable in the presence of large wind shears. This instability is called the Kevin-Helmholtz instability and can be held responsible for areas of air turbulence in higher levels of the atmosphere.

The motion of gravity waves is described by the dispersion relation, (1.39). They are fairly short in length, less than \( r_E \), as they are unaffected by the rotation of the Earth. At a height of 6km in medium latitudes\(^3\), \( r_E \approx 3000\text{km} \). Gravity waves have frequencies in the range \( 10^{-1}\text{Hz} \) to \( 10^{-5}\text{Hz} \) (minutes to hours).

**Rossby waves**

Rossby waves are a form of planetary wave named after their discoverer, C. G. Rossby. A detailed description of Rossby waves may be found in [39] or more general descriptions in [2, 22, 17].

Horizontally transverse in nature, they are low frequency long waves. The frequency range includes \( 10^{-5}\text{Hz} \) to \( 10^{-7}\text{Hz} \) (months to years), with typical wavelengths in the thousands of kilometre range.

The rotation of the Earth gives rise to the Coriolis force. See §1.2.2 and fig. 1.1 for reference. This force is dependent on the angular velocity of the Earth and the latitudinal distance from the equator. It is responsible for the directional behaviour of Rossby waves.

Due to the direction of rotation of the Earth, Rossby waves have a characteristic phase velocity that always propagates in a westward direction. It is often found that this velocity is in opposition to that of the background wind. As such, the action of Rossby waves is of most interest in regions dominated by eastward blowing winds.

\(^3\)the lower atmosphere ends at 7-17km depending on latitude
Rossby waves are dispersive with a dispersion relation, \( (1.40) \). Long Rossby waves tend to propagate westward whereas short Rossby waves move their energy eastward.

**Cyclonic activity**

When a fluid mass is in contact with another mass having differing physical properties, shear forces exist. These are most dominant in the boundary layer, an area extending outward from the contacting surface. They exist within the fluid due to the variation in velocity across the fluid. Consider water flowing through a pipe. At the centre of the pipe, or more specifically, furthest from the walls, the water will have the highest velocity in a direction parallel to the walls of the pipe. In contrast, the lowest velocity is seen in water nearest the walls of the pipe. This is due to the motion resisting forces of the wall. The shearing stresses produced are proportional to the viscosity of the fluid. Similarly in the atmosphere, two air masses having differing physical properties may interact and cause shearing waves. The contact surface of two such air masses is known, meteorologically, as a **front**. At these fronts, turbulence in the form of cyclones may occur as sheared Rossby waves convert their potential energy into kinetic energy.

**Inertio-gravity waves**

IG waves are surface gravity waves whose lengths are large enough, \( L > r_E \), to feel the effect of the Earth’s rotation.

They occur with frequencies extending slightly beyond the range \( 10^{-3} \text{Hz} \) to \( 10^{-4} \text{Hz} \) (hours to days) and are described by the dispersion relation, \( (1.38) \). Using Rossby waves as a comparison, IG waves are high frequency.

### 1.4 Equations of atmospheric wave motion

The models of fluid motion derived below [24, 25, 22] are based upon the equations of motion found in §1.2.

#### 1.4.1 Navier-Stokes

The traditional Navier-Stokes model combines the conservation of momentum equation, \( (1.26) \), minus the Coriolis term,\

\[
\frac{D\vec{v}}{Dt} = \vec{g} - \frac{1}{\rho} \vec{\nabla} p + \eta \nabla^2 \vec{v}
\]  
\( (1.41) \)
with the nondivergent continuity equation (1.19). The system defined by (1.41) and (1.19) describes the motion of incompressible fluids in an inertial frame. Although liquids tend to be slightly compressible in reality, for macro models of this type they may be regarded as incompressible. To apply the same notion to highly compressible gaseous fluid such as air requires some justification. The likelihood of significant gaseous compression occurring is dependent on the speed of flow. Indeed, the flow speed must be much lower than the speed of sound, \( c_s \), for a gas to remain approximately incompressible. Lesieur [27] states that flow speeds in the order of \( \frac{1}{3} c_s \) (a few hundred miles per hour) can still be correctly described using the incompressibility assumption. The Navier-Stokes equations written in scalar form are

\[
\begin{align*}
\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} + w \frac{\partial u}{\partial z} &= -\frac{1}{\rho} \frac{\partial p}{\partial x} + \eta \nabla^2 u, \\
\frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} + w \frac{\partial v}{\partial z} &= -\frac{1}{\rho} \frac{\partial p}{\partial y} + \eta \nabla^2 v, \\
\frac{\partial w}{\partial t} + u \frac{\partial w}{\partial x} + v \frac{\partial w}{\partial y} + w \frac{\partial w}{\partial z} &= -\frac{1}{\rho} \frac{\partial p}{\partial z} + \eta \nabla^2 w - g, \\
\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} &= 0.
\end{align*}
\]

### 1.4.2 Barré de Saint-Venant

To form the Barré de Saint-Venant or shallow water equations, the Navier-Stokes equations are considered locally for a fluid with uniform density, \( \rho \). This fluid is located on a sphere and is considered within a rotating reference frame.

The Coriolis parameter, \( f \), is approximated using (1.21) where the planetary vorticity, \( 2f \), is projected onto the local vertical. In this case, (1.26) becomes

\[
\frac{D\vec{\sigma}}{Dt} = \vec{g} - fn_x \times \vec{\sigma} - \frac{1}{\rho} \vec{\nabla}p + \eta \nabla^2 \vec{\sigma}. \tag{1.42}
\]

The definition of shallowness assumes a hydrostatic pressure distribution in the fluid. With \( p_0 \) representing pressure at the free surface,

\[
p = p_0 + \rho gh \tag{1.43}
\]

where \( h(x, y) \) is the free surface height of the fluid. Substituting for \( p \) in the \( z \) compo-
Equation (1.44) shows that the dependence of horizontal velocity components on vertical motion is zero, hence, the model of fluid flow is simplified to two dimensions. The $y$ component of (1.42) therefore becomes

$$\frac{Dw}{Dt} = -g \frac{\partial h}{\partial y} - f u + \eta \nabla^2 v. \quad (1.45)$$

where $\nabla^2$ operates in horizontal directions only.

Integrating the nondivergent continuity equation, (1.19), along the vertical gives

$$\frac{\partial h}{\partial t} + \vec{v} \cdot h \vec{\nabla} = 0. \quad (1.46)$$

Using the vector identity,

$$\vec{v} \cdot (\vec{\nabla} h) = \vec{\nabla} \cdot (h \vec{v}) - h (\vec{\nabla} \cdot \vec{v})$$

and introducing Stokes operator, transforms (1.46) into

$$\frac{Dh}{Dt} = -h \vec{\nabla} \cdot \vec{v}. \quad (1.47)$$

where the operator $\vec{\nabla}$ is two dimensional. The Barré de Saint-Venant model describing shallow fluid motion is the combination of (1.44), (1.45) and (1.47).

### 1.5 Turbulent inertio-gravity (IG) waves

The linear atmospheric model described below is derived in [10, 11]. It is solved in chapter 4, which resulted in the publication of [12].

---

$^4$due to the hydrostatic pressure assumption
1.5.1 2D system

Neglecting viscosity, the shallow water equations:

\[
\frac{Du}{Dt} = -gh \frac{\partial h}{\partial x} + fv, \\
\frac{Dv}{Dt} = -gh \frac{\partial h}{\partial y} - fu, \\
\frac{Dh}{Dt} = -h \nabla \cdot \mathbf{j}
\]

(1.48)

describe IG waves after linearization with the dispersion relation,

\[ \omega^2 = f^2 + c^2 k^2 \]

where \( f \) is the Coriolis parameter and \( c = \sqrt{gh} \), the speed of long, surface-gravity waves.

IG waves are of particular significance when looking at mesoscale circulation of the atmosphere or long tidal wave phenomena. Atmospheric and oceanographic statistical data [50] suggests that the spectra of 2D vortex turbulence has a peak at \( \omega \approx f \) at the lower end of the mesoscales\(^5\). This is at the lowest frequencies of inertia-gravity waves and is due to Kraichnan\'s inverse energy cascade [21]. Similarly, tidal waves have been observed as having a sharp energy density peak at \( \omega \approx f \). It is known that waves contribute to this part of the energy spectrum in geophysical turbulent flows.

**Nonlinear model**

Falkovich [10] uses the system:

\[
\frac{\partial p}{\partial t} - q + \frac{\partial}{\partial x} \left( \frac{p^2}{h} \right) + \frac{\partial}{\partial y} \left( \frac{pq}{h} \right) + h \frac{\partial h}{\partial x} = 0, \\
\frac{\partial q}{\partial t} + p + \frac{\partial}{\partial y} \left( \frac{q^2}{h} \right) + \frac{\partial}{\partial x} \left( \frac{pq}{h} \right) + h \frac{\partial h}{\partial y} = 0,
\]

\[ \frac{\partial h}{\partial t} + \nabla \cdot \mathbf{h} = 0 \]

which is (1.48) re-written in terms of momentum, \( \mathbf{h} = h \mathbf{j} = (p, q) \), using the dimensionless variable set: \( h/H_0, \mathbf{j}/c, ft \) and \( r/p \). This system generates a nonlinear wave

\(^5\)Hundreds of kilometres
CHAPTER I. PHYSICAL BACKGROUND

equation:

\[ 2i + \nabla^2 \Psi + 2iJ(\Psi, |\Psi|^2) = 0 \]  \hspace{1cm} (1.49)

where

\[ J(A, B) = \frac{\partial A}{\partial x} \frac{\partial B}{\partial y} - \frac{\partial A}{\partial y} \frac{\partial B}{\partial x} \]

and \( \Psi = (q_i + ip_i)/2 \).

Equation (1.49) is valid for \( L \gg r_E \) under which conditions the dispersion of IG waves is weak. It describes long nonlinear IG waves in the atmosphere and tidal waves in the oceans. Falkovich then goes on to show that steady solutions with finite \( R_l \) are impossible in the above framework. Consider the mean square radius of distribution:

\[ \frac{\partial^2 R}{\partial t^2} = \frac{\partial}{\partial t} \int r^2 |\Psi|^2 \, dx \, dy = 2 \int |\nabla^2 \Psi|^2 \, dx \, dy > 0. \]

Under these conditions, any localised distribution spreads over the whole space causing the mean radius to increase monotonically. The model produces condensate instabilities and is unable to provide feedback in order to obtain a wave cascade.

Linear approximation

These problems are solved by a linear approximation of (1.49):

\[ 2i \frac{\partial \Psi}{\partial t} + 2iJ(h, \Psi) + h\nabla^2 \Psi - \Psi \nabla^2 h = 0. \]  \hspace{1cm} (1.50)

which lacks the undesirable properties of the nonlinear equation. This is achieved by assuming small flow perturbations of height, \( 1 + \eta \), where \( \eta \ll 1 \). These perturbations are in geostrophic balance with the currents:

\[ 2p_0 = -\frac{\partial h_0^2}{\partial y}, \quad 2 \varphi_0 = \frac{\partial h_0^2}{\partial x}. \]
1.5.2 Evolutionary perturbation in 1D

For scales of motion greater than the intermediate geostrophic radius, the Coriolis force should not be neglected. In this case, \( h \) is evolutionary [11]:

\[
\frac{\partial h}{\partial t} = \beta \frac{\partial}{2 \partial x} \left( h^2 + \frac{\psi^2}{h} \right) .
\]  

(1.51)

Using (1.51), (1.50) reduced to a single dimension becomes

\[
2i \frac{\partial \psi}{\partial t} + h \frac{\partial^2 \psi}{\partial x^2} - \psi \frac{\partial^2 h}{\partial x^2} = 2i \beta \psi \frac{\partial h}{\partial x}.
\]  

(1.52)

1.6 Rotating nonlinear IG waves

Farge [13, 15] has also investigated IG wave interactions. Her efforts were directed towards the effects of IG waves and rotation on 2D turbulent flows.

1.6.1 Nonlinear 2D system

The shallow water equations may be represented in vector form as

\[
\frac{\partial \vec{v}}{\partial t} + (\vec{v} \cdot \vec{\nabla}) \vec{v} = -g \vec{\nabla} h - f n_z \times \vec{v} + \eta \vec{\nabla}^2 \vec{v},
\]  

(1.53)

\[
\frac{\partial h}{\partial t} + \vec{v} \cdot \vec{\nabla} h = -h \vec{\nabla} \cdot \vec{v}.
\]  

(1.54)

Considering the reduction in dimensionality caused by stratification and using the vector identity (1.27) with some manipulations, (1.53) can be re-written as

\[
\frac{\partial \vec{v}}{\partial t} + (\vec{v} + f) \times \vec{v} + \vec{\nabla} \left( \phi + \frac{v^2}{2} \right) = \eta \vec{\nabla}^2 \vec{v}
\]  

(1.55)

where \( \phi = gh \) is introduced as the geopotential and \( \omega \) describes vorticity as usual. Similarly, (1.54) can be re-written as

\[
\frac{\partial \phi}{\partial t} = -\vec{\nabla} \cdot (\phi \vec{v}).
\]  

(1.56)

Ignoring viscous effects, (1.55) can be further separated into PDEs describing the evolution of vorticity, \( \omega = \vec{\nabla} \times \vec{v} \), and divergence, \( \delta = \vec{\nabla} \cdot \vec{v} \). When combined with (1.56) and written in scalar form, a nonlinear model of fluid motion able to describe
the interaction of IG waves in the atmosphere is established:

\[ \frac{\partial \omega}{\partial t} + \nabla \cdot (\omega + f) \delta = 0, \]

\[ \frac{\partial \delta}{\partial t} - \nabla \times (\omega + f) \delta + \nabla^2 \left( \phi + \frac{u^2}{2} \right) = 0, \]

\[ \frac{\partial \phi}{\partial t} + \phi \delta + \vec{v} \cdot \vec{\nabla} \phi = 0. \]
Chapter 2

Programming and computing

An introduction to the technical aspect of this research is included here. The choice of computer language, the specialised software that was used and parallel processing techniques are covered.

2.1 Language considerations

The type of numerical computation involved in solving systems of PDEs basically breaks down to the manipulation of products and sums of array elements. The multi-dimensional arrays are used to store solution domains. They may contain either real or complex floating point numbers (types). The computation is non-interactive. Low-level interaction with either the operating system (OS) or the hardware is not normally necessary.

Initially, no specific machine architecture or operating system was exclusively targeted for the simulations. However, UNIX based single processor Sun workstations were immediately available with FORTRAN 77, C and C++ compilers [46, 45]. A suitable computer language and compiler system with the following characteristics and features was sought:

- Robust compiler and language structure
- Portable
- Complex arithmetic operators and type
- Speed efficient, good optimization
- Simple, high level operations
A robust compiler implies mature compiler technology. This does not dismiss compilers that are still in their infancy, it simply recognises that they are more likely to contain bugs or produce code containing bugs: either by failing to detect programming errors or introducing bugs themselves. They are also more prone to incorrect or underdeveloped optimization.

For numerical computation, the robustness of a language is dependent on its simplicity. The flexibility and power of a language such as C was not required. In fact, the complexities of memory management and the consequential loss of robustness of such a language is not only unnecessary but undesirable in this context. The simpler the language is to use, the lower the likelihood of systematic programming errors, i.e., inability to use or incorrect use of the language. A robust, high level language will shield the user from the intricacies of the system and allow attention to be focused upon the numerical algorithms and their efficient implementation rather than how and where an array is stored in memory.

Some languages are intrinsically more portable than others. Those that allow low level coding tend to be less portable. Hand optimizations based upon machine and OS type do not port well. The way in which a language emerges can also hamper portability. K&R\(^1\) C suffered from having a poorly defined standard, which led to various incompatible implementations. In an attempt to combat this problem, an ANSI committee was established which produced an ANSI standard for C [20].

A native complex typing system was required due to the nature of the numerical models involved. The first model is described by a complex PDE and the second model, although described by a set of real PDEs, used complex type through Fourier transformations. Use of the word “native” implies the availability of generically typed intrinsic functions. For instance, the ability to derive the product of two complex numbers by using the usual multiplication operator:

\[
C_1\{R_1, \Im_1\} \cdot C_2\{R_2, \Im_2\} = C_3\{R_1 \cdot R_2, \Im_1 \cdot \Im_2\}.
\]

Certain languages enable the construction of a complex type. In C it is possible to define a two element real array as a complex type, however the use of special functions is required whenever any arithmetic is performed on the derived type. This produces an added level of difficulty for the programmer and forces the compiler to work harder when attempting to optimize such code.

Another issue determining language choice is the run-speed efficiency attainable by

---

\(^1\)Brian Kernighan and Dennis Ritchie
a given algorithm. This is highly characteristic of the language used. Other things being equal, compiled code will always run faster than interpreted code. The speed of a compiled language is affected by the optimisations produced by the compiler. Inherently, the higher the level of the language, the easier it is for the compiler to optimise the code.

The language chosen was the somewhat archaic FORTRAN 77 [9], which was used throughout. It is not a fashionable language like C++ but nevertheless it is a robust, mature, portable, high level compiled language with a native complex type. The emergence of Fortran 90 [33] and its successor, High Performance Fortran (HPF), are aimed at bringing the Fortran language up to date and providing a language suitable for machines with parallel architectures. Both of these incarnations of Fortran can compile\(^2\) standard FORTRAN 77, thus easing the transition into the world of parallel computing.

2.2 Software

2.2.1 Bundles and libraries

The code produced was intended to be written in a portable manner allowing execution on any platform having a FORTRAN 77 compiler with standard FORTRAN 77 libraries.

Programming tasks involved numerical computations: solution of the model, and the visualisation of these solutions. Each problem could thus be cleanly split into two tasks.

Although the numerical task was speed critical, the visualisation process was not. Hence it was possible for each task to be tackled individually by different programs. Indeed, there was no need for the tasks to be performed on the same platform or even using the same language. However, it is reasonable to assume that the transfer of binary data from one program to another is more straightforward when programs share a common language and platforms share a common byte ordering.

In view of this, portability was only necessary for the speed critical numerical task. The creation of realistically portable code requires that any non-standard or third party mathematical routines be available in source form. This explains the lack of dependence on commercial software libraries such as those produced by NAG.

The nature of the mathematical models and the methods used for their solution called for very few general mathematical procedures not available in the standard math

\(^{2}\)although parallelizing optimisations may be poor
library. At the celerity of this research, the only third party routines used were some of those available in a fast Fourier transform (FFT) library. This library was mature, freely available and was obtained as FORTRAN77 source code.

### 2.2.2 Visualisation techniques

Although the majority of research work was involved with numerical tasks, the solutions obtained by this work required visualisation. Both on-screen and hard-copy versions were needed.

The models themselves simulated 2D movement and the solutions, also 2D, required 3D space for visualisation. Three software packages were used for this purpose:

- **UNIRAS**: 3D, program driven, graphical display library
- **xmgr**: 2D, interactive graph tool
- **gnuplot**: 3D, line driven graph tool

The UNIRAS package was deemed suitable to display such images and was chosen as the main visualisation tool. It was available as a FORTRAN77 library of callable functions and gave a wide variety of 3D and virtual 4D graphical features using colour shading. The graph type chosen, illustrated by results displayed in chapter 4, was a height shaded 2D plot with the inclusion of isolines to ease interpretation. This format was chosen in favour of a more aesthetic output as it was easier to compare and isolate small changes with.

xmgr is an interactive graphical presentation tool allowing the display of 1D data. This was used to track 1D computation variables such as energy and enstrophy. It was also used to show 1D slices of a 2D solution to aid the interpretation of the result. Examples of xmgr output can be found in chapter 3.

gnuplot is a basic, line driven 3D graphical tool. It was used in a similar way to the UNIRAS routines, although its options were limited and the data had to be passed in the form of an ASCII file. This required the use of a translation program to turn the raw binary data produced by simulations into a suitable ASCII form. The need to use a second 3D graphical environment was driven by software incompatibilities between the original UNIRAS libraries used and a recent version of their host OS. gnuplot was used to display the results found in chapter 5.
2.3 Computer architecture

Computer systems come in many different shapes, speeds and sizes. However, there are two distinct classes: serial and parallel [4].

Serial machines have a single main processor and process a single instruction at a time. Multi-tasking operating systems such as UNIX may hide the serial nature of the underlying processor, but nevertheless instructions are queued up in a determinable fashion and executed in a determinable order.

Parallel machines have several main processors and are able to undertake multiple operations simultaneously: concurrent processing. The goal of concurrent processing is to enable a task to complete in \(1/n\)th of the time using \(n\) processors than it would take using a single processor. However, this is not always possible to achieve, due to unequal divisions of the task and the costs associated with process communication or shared memory arbitration.

Examples of the main categories of computer system are:

- **Serial**
  - Intel 8086
  - Super-scalar, branch predictor, multi ALU: Cyrix M1

- **Single instruction multiple data (SIMD)**
  - Vector: Cray Y-MP
  - Parallel array: AMT DAP (Distributed Array Processor)

- **Multiple Instruction Multiple Data (MIMD)**
  - Distributed memory multicomputers
    - Loosely coupled: Network of machines using an implementation of Parallel Virtual Machine (PVM)
    - Tightly coupled: T800 Transputer array
  - Shared memory multiprocessors
    - Sun SPARCstation 1000 with multiple SPARC CPUs
    - Intel Paragon

Traditionally, programming languages such as FORTRAN77 were designed for use upon single threaded, single processor systems - the von Neumann model. The semantics available forced the problems to be implemented as sequential algorithms. Each
algorithm being processed in a fixed, determinable way. Only a single instruction is being processed on a single piece of data at any moment in time on a von Neumann computer. This programming paradigm cannot be used with any form of concurrent processing resource if an increase in program execution speed is warranted [42].

To use a parallel system at its full potential, a program must be organised in a way in which all available processors are always fully occupied. Any time spent by a processor waiting for I/O or synchronisation is time wasted. This requirement led to the development of parallel based languages such as OCCAM, where sets of instructions are grouped and labelled as either parallel or sequential. As implied by the sequential label, parallelism is not necessarily inherent in an algorithm and in some cases may be impossible for a particular task. The target platform of an OCCAM program is usually a Transputer array. Other approaches have been made in a more general direction where the exact details of the target architecture are less important. HPF is a language developed as an extension to Fortran 90. It provides control statements which give the programmer more flexibility in parallel program design, but as a superset of Fortran 90, will compile both Fortran 90 and FORTRAN77 programs. Virtual parallelism has also become popular, using software such as PVM and LINDA, allowing so called “free compute cycles” to be accessed on under-utilised workstation clusters.

2.4 Parallel systems

Parallel systems are defined by the following characteristics

- Memory configuration
  - Shared
  - Distributed
  - Virtual shared

- Process communication
  - Message passing
  - Shared variables

- Interconnection or networking

\[^3\]It is possible for super-scalar processors to compute several branches ahead of the current instruction, but this is prediction only. Results obtained from computing all but the correct branch are discarded.
Memory configurations of parallel machines are illustrated in fig. 2.1. A shared memory computer has a single pool of memory that is shared by all of its processors. This is achieved by arbitration hardware and software memory management schemes.

Distributed memory systems have a collection of processors each with their own individual memory pool. The processors are interconnected to form a distributed network.
In a virtually shared memory system, each processor has its own memory pool, as with distributed memory systems, but rather than interconnection between processors, there are interconnections between the individual memory pools. The memory can be physically addressable by a combination of processor and byte ID or it can be viewed as a virtual global address space.

Communication between processes can be achieved using message passing or shared variables paradigms. The former scheme is used by distributed memory systems. Bundles of data are packed into a message with a header containing relevant information such as sender and receiver identifiers, data type and length. The message is sent to the receiving process or processes where it is unpacked and made available for use.

Shared variables are used in shared memory systems. Memory access is controlled by locking and arbitration mechanisms to protect data areas from concurrent writes and to give processes solitary access to regions of memory.

From a programming point of view, nodal interconnection may be seen as physical, logical or transparent. The routing of messages or memory accesses may be fixed or reconfigurable. This information is required to unlock the full potential of a system and may have a drastic effect on program structure.

### 2.5 Concurrent programming techniques

Due to the many different types of parallel architecture available, there are also many different ways in which the underlying parallelism is made available to the programmer. The main area of concern on any parallel resource is that of sharing data between processors. As there are two main classes of memory model for parallel machines: distributed and shared, there are two conceptually different methods. These are known as message passing and shared variables respectively.

#### 2.5.1 Message passing

In a distributed memory environment, data must be physically transferred from one processor's memory to another. Data is packed up to form a message which is then passed to the appropriate processor. Once received, the message is unpacked and the data becomes available to the new processor.

---

4 A node describes an individual processor and/or memory unit
5 Virtual shared memory is treated in the same way as shared memory but has the scalability usually associated with distributed memory
There are several factors involved in passing messages between processors. Synchronization is one of them. When a message is passed from one node to another, the sender may either wait for an acknowledgement from the receiver (block) to determine that it has been received or simply send it blindly and continue its work (non-blocking). Similarly, a receiver expecting a message may wait for its arrival (block) or continue its work and poll its input queue at regular intervals to check if it has arrived (non-blocking). The addressing of the message is also important as messages may be sent from any number of nodes to any number of nodes.

### 2.5.2 Shared variables

A shared memory environment does not have to deal with the transfer of data. Each processor has a direct connection to the shared memory and consequently, direct access to the data used by other processes. The main problem that shared memory architectures have to address is the actual method used to share the memory. If more than one process tries to access the same memory cell at the same time, undefined results may occur. A similar problem exists in large shared database systems when multiple users try to read, modify and replace a record at the same time. This may be solved by a record locking technique using check-in and check-out mechanisms to control access. In parallel systems, cell locking is used. The shared variables method enables a process to lock and unlock regions of memory, thus preventing access to or from other processes whilst critical actions are performed. In this context, processes are known as *threads* of execution.

Shared variable programming has to deal with protection and synchronization. Protection has already been mentioned. Synchronization allows a thread to block until a particular situation has occurred. This situation must be arranged by another thread, which is itself not waiting for the former thread, either directly or indirectly, else the system will suffer deadlock.

### Semaphores

A semaphore is an abstract value associated with a piece of memory. Before accessing the relevant memory, a thread must call *wait* on the semaphore and is allowed to proceed if it is not already in a wait state. Once finished, the thread must call *signal* on the semaphore to unblock access by other threads. Essentially, the semaphore is a throttling device allowing single thread access to its associated memory. As there is nothing to stop a thread from bypassing the semaphore and accessing the memory...
directly, calling signal and wait in the wrong order or never calling signal at all, various higher level structures have been built around the use of these primitives.

Monitors

Monitors are high level programming constructs that allow process arbitration of shared variables. They provide the programmer with a way to encapsulate several procedures in a single function. Multiple entry points give access to the individual procedures contained within. It is assumed that each of the procedures in the function operates on the same set of shared variables. The object of a monitor is to allow only a single thread to be in a particular monitored function at any one time.

2.5.3 Deadlock

Deadlock defines a process that is waiting for an event which will never happen. The event may be anything that is arranged by another process. It may be caused by a variety of problems, an example of which is bad scheduling, i.e. the event occurred earlier than expected. Other processes that depend upon the deadlocked process may also become deadlocked themselves, in which case the entire system may deadlock. A badly programmed task containing a potential deadlock may not fail consistently due to the somewhat random order of processing which may occur in a parallel environment.

2.6 Task partitioning methods

To achieve concurrency in a task, the task must be split up into pieces that can proceed separately, but in parallel with each other. There are two basic task partitioning methods: data partitioning and program partitioning. In data partitioning methods, the main program is duplicated on the available processing nodes but each node is only responsible for working upon a particular portion of the program data. The portion or portions may be allocated statically or dynamically depending upon the implementation. In program partitioning, the program is split into concurrent algorithms which are shared among the available nodes. Each node is assigned an individual task. In this case, the entire data set is processed by each of the nodes.

The easiest method to implement tends to be data partitioning as there is usually only a single worker program required. Program partitioning, by definition, requires a different program on each node. Based on the same reasoning, data partitioning schemes are inherently more scalable, as further subdivision of data is usually easier.
than subdivision of algorithms. Data partitioning also tends to use the processing resource more efficiently as is shown in a comparison below. This is especially true when dealing with regular data sets, as the load balancing of processors becomes less of a problem.

The partitioning methods described below are relevant to distributed memory systems as this was the type of system eventually used. Their relevance to shared memory systems are not discussed.

2.6.1 Data partitioning

The main data partitioning schemes are known as farm, geometric and Long range. Each scheme is suited to a particular type of problem.

Farm

A farm partitioning scheme can be used on problems whose data can be split into completely independent portions. There should be no data interdependencies. An example of such a problem is ray tracing. A single master process controls a number of identical worker processes. Communication is between the master and the workers only. The scheme tends to be dynamically based with the master passing portions of data to each idle worker. Once a worker has finished processing its data, it returns it to the master while at the same time asking to be assigned some more data. In this way, each worker is kept fully occupied until the master has no more fresh data to pass out. Once the last worker has finished and returned the last portion of data, the entire task is complete. Due to the built-in dynamic behaviour and lack of worker synchronisation, this method is as efficient when dealing with irregular areas of data as with regular: there is no penalty if certain portions of data require higher degrees of processing than others.

Geometric

Geometric data partitioning is used when the data has immediate neighbour dependencies. An example of such a problem is in the solving of PDEs. The mesh of data is statically split up between the number of available nodes. There are various mesh partitioning schemes in existence that seek to minimise the amount of neighbours and/or the size of the interfaces for a given number of partitions. Each node runs

coordinate bisection, Greedy, bandwidth minimisation, MINCUT
an identical program on its pre-allocated portion of data and is also responsible for sharing boundary data with the nodes working on the surrounding portions.

Long range

Long range data partitioning, as the name suggests, is used when the data has long range dependencies. Such a scheme can be envisaged as a systolic ring. The data is partitioned statically as in the geometric case, with each node running an identical program. However, a data packet known as a travelling salesman, visits each node in turn. Any data that is required by the visited process is delivered (removed from the data packet) and any data required by other nodes is given to the salesman (appended to the data packet). The travelling salesman continues circling the ring until the task is complete.

2.6.2 Program partitioning

Various program partitioning schemes exist: pipeline, dataflow, systolic array. Using the common pipeline scheme as an example, the task is split into concurrent work units or algorithms of equal load. Each of the algorithms is assigned to a node. A portion of the data, possibly a single element, is given to node 1. Once the program at node 1 is done, the data is passed on to node 2 and node 1 accepts new data. The data is passed along the nodes in the same way that water flows through a pipe, hence the name. Once the final portion of data emerges from the final node, the task is complete.

Partitioning comparison between farm and pipe

Consider a task with the following criteria:

- 1000 element data array
- 5 nodes
- 10 units of work required per data element
- negligible communication costs

A static farm method is employed with all 5 nodes running the same worker program of work value 10. No master program is required. Each of the nodes processes a separate 200 element portion of the data. This method would produce a solution

\(^7\)not necessarily circling, but in any event visiting each node in turn
after $1000/5 \times 10 = 2000$ units of work time. As a worse case example, the data is irregular with some elements requiring much more processing than others, although on average each element requires 10 units of work. The method is modified to use a master program which farms out the work dynamically. This leaves only 4 nodes available to process the data. The solution would be complete in $1000/4 \times 10 = 2500$ units of work time.

An ideal pipe method would partition the work load evenly amongst the nodes so that each node provided 2 units of work. The first element of data would emerge after 10 units of work time, the second after 12 units. Due to the startup and shutdown costs of the pipe method, the solution would be complete after $10 + (999 \times 2) = 2008$ units of work time. As a worse case, the work can only be split into one node providing 6 units of work, the remaining nodes providing a single unit each. In this case, the solution would require $10 + (999 \times 6) = 6004$ units of work time.

### 2.7 Distributed virtual parallelism

The parallel architectures mentioned so far have been dedicated parallel machines running specialised OSs. Access to such hardware is fairly restrictive due to its considerable purchase costs. An attractive alternative to dedicated parallel hardware is the use of workstation clusters. These clusters can be configured to act as loosely coupled distributed memory systems. Communication speeds are severely restricted by use of TCP/IP over Ethernet (10Mbits/s) or FDDI (100M bits/s) and subnet style networking allows only a single message to pass between any two nodes (workstations) at any one time.\(^8\) However, a suitably coarse grained problem can still attain near ideal speedup using this method.

There are various programming environments available for distributed virtual parallel computing. These include

1. PVM
2. P4
3. TCGMSG
4. Linda

\(^8\)unless subnets of size two are used in which case scalability is severely restricted
The first three involve message passing libraries, each known as a coordination library. The fourth is a coordination language which comes in two flavours: C-Linda and F-Linda. These provide language based extensions to C and FORTRAN 77 respectively. With a coordination language, the compiler is given the opportunity to detect possible inconsistencies and problems with the implementation of the parallelism. A coordination library simply provides pre-built parallel constructs that are linked into the program after compilation.

There have been several comparisons [32, 31] between the environments mentioned above, however the conclusions show that each has its relative advantages and disadvantages.

In deciding which environment to use, the following factors were considered: cost, generality, features, performance and ease of coding and debugging.

Linda was ruled out by virtue of it being a commercial product. The other environments were and are freely available. PVM fared well both in terms of generality and features. Of those left, it was the only environment that supported heterogeneous workstation clusters, virtual nodes on a single machine, dynamic process organisation and the concept of process groups. Virtual nodes allow the testing of a parallel implementation on a single workstation, thus making debugging far simpler. Process groups are special subsets of processes. Processes may join and leave a group at any time. Communication and certain operations are simpler if applied to a group rather than a direct subset of all processes. For instance, consider that a message needs to be sent to all processes that are currently in a particular state. There are several ways to accomplish this:

1. The message is sent to every process. Any process not in the particular state must ignore the message.
2. All processes are queried about their current state. Any process that is in the correct state is sent a message.
3. Whenever a process enters the particular state, it joins a named group. Once finished it leaves the group. The message is sent to the members of the named group.

The first two of the above methods waste communication bandwidth. Decision logic is also required in the receivers (1) or the sender (2). Using process groups (3), wastes no bandwidth and requires no decision logic on the part of the programmer.

Although the added complexity of PVM reduces its ease of programming, the greater degree of flexibility available was seen to give it an advantage over the others
and consequently it was the chosen environment. Considering that it is by far the most common parallel processing environment \[32\], others seeking the rewards of parallelism seem to have come to the same conclusion.

### 2.7.1 Parallel Virtual Machine (PVM)

As mentioned above, PVM \[16\] is a software system that enables a network of heterogeneous serial and parallel machines to resemble and behave like a distributed memory model of parallel computing. The drawback, in common with every distributed software solution, is the significant inter-nodal communication costs. This alone leads to a fundamental difference in programming style to that possible using a dedicated distributed memory parallel computer. Non-negligible communication costs bring the concept of granularity into question. A fine grained problem is defined as having a low ratio of computation to communication. This is usually caused by high degrees of data interdependence. Conversely, a coarse grained problem has a high ratio of computation to communication. There is a granularity threshold, dependent upon available bandwidth and communication overhead, below which any distributed system will fail to provide a speedup. Indeed, it is also possible to slow a program down in attempting to parallelize it. To avoid this problem in a PVM environment, special attention must be placed on increasing the granularity of the problem. As this can only be of benefit, the transfer of a PVM application to a tightly coupled distributed parallel system is not compromised. However, the positive effects of reduced communication will be less apparent.

PVM provides coordination libraries: standard and group, an initialisation program, pvm, and a daemon process, pvm3d. The daemon process resides on each node of the virtual machine. Its job is to organise and oversee the passing of messages between processes. It also provides buffer space for messages that have arrived from other processes destined for a local process. These are held by pvm3d indefinitely, until the recipient process on the local node checks its input queue and accepts the message. The pvm program controls the initialisation of the virtual machine by both user input and a configuration file. It is also responsible for spawning the pvm3ds and user processes. See fig. 2.2 for an illustration of PVMs virtual machine. The configuration file provides information regarding specific machines, their capabilities and the organisation of the virtual machine as a whole. User input is used to override various aspects of the default configuration.

No special privileges are necessary to install and use a PVM system. A standard shell account on each machine is all that is required. The system is compiled on each
of the different platforms on which it will be used and the `pvm` program ensures that the correct architectural version is used.

Both FORTRAN 77 and C were supported by PVM version 3.3.3 (May 1994). The PVM library functions are coded in C and it was noted that the FORTRAN 77 library consisted of FORTRAN 77 to C conversion functions. These functions then called the equivalent C library function. Therefore, use of the FORTRAN 77 library incurred a slight overhead in relation to using the C library directly.
Chapter 3

Numerical methodologies

A considerable amount of numerical methods were both investigated and implemented during this research. An introduction to their derivation, use and implementation is provided here. Further information may be gained from [19, 40, 18, 36] which were used as a learning base for this work.

Many experiments were conducted during familiarisation with these methods and some of the more significant results are included here.

3.1 Introduction

3.1.1 Partial differential equations (PDEs)

The solution of a PDE will yield a function or functions which are said to satisfy it. As an example, the 1D wave equation,

\[
\frac{\partial^2 \psi}{\partial t^2} = u^2 \frac{\partial^2 \psi}{\partial x^2},
\]

is satisfied by all of the functions:

\[
\psi = \begin{cases} 
  f(x - ut), \\
  g(x + ut), \\
  \alpha f(x - ut) + \beta g(x + ut)
\end{cases}
\]

provided that \(f\) and \(g\) contain a second order derivative and \(\alpha\) and \(\beta\) are constants.

If the form of a solution is known, it is sometimes possible to solve a PDE by hand using an analytical method such as the separation of variables. However this
does not apply to certain linear, and all nonlinear PDEs. In these cases, brute force computation provides the answer.

A continuous PDE cannot be represented using a discrete number system. To enable a discrete number machine such as a computer to solve a PDE, it must first be approximated. Such an approximation is defined at discrete points in space and time only. The technique used here to represent continuous partial derivatives in the discrete world of computing is the method of finite differences. Other techniques such as finite elements and spectral methods are also known to exist.

**Discretisation**

Each continuous partial derivative can be fully represented by combinations of Taylor series expansions as illustrated later in §3.1.2. These expansions are then truncated to remove higher order derivatives, the result being a finite difference approximation (FDA) of the continuous derivative. This discrete FDA allows the derivative to be described in terms of neighbouring points and for simple derivatives, the transformation is fairly intuitive.

Consider a slope. The gradient of the slope between two points, \(i+1\) and \(i-1\) is, by definition, the change in vertical height, \(f\), over the change in horizontal distance, \(x\), or

\[ g_{i+1,i-1} = \frac{f_{i+1} - f_{i-1}}{x_{i+1} - x_{i-1}}. \]

The limiting case, where the distance between the two points tends to zero produces the conventional differential gradient:

\[ g = \frac{df}{dx}. \]

Alternatively, let the point \(i\) be introduced midway horizontally between \(i-1\) and \(i+1\), the gradient at \(i\) can then be interpreted as

\[ g_i = \frac{f_{i+1} - f_{i-1}}{2(x_{i+1} - x_i)}. \]

This is analogous to the truncated first order FDA:

\[ \frac{\partial f}{\partial x} = \frac{f_{i+1} - f_{i-1}}{2h}. \]

Once an FDA has been found for each of the continuous derivatives, a full discrete
CHAPTER 3. NUMERICAL METHODOLOGIES

Table 3.1: Characterization of PDE by discriminant

<table>
<thead>
<tr>
<th>$B^2 - 4AC$</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>elliptic</td>
</tr>
<tr>
<td>= 0</td>
<td>parabolic</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>hyperbolic</td>
</tr>
</tbody>
</table>

approximation can be constructed which is known as a finite difference equation (FDE).

Characterization

PDEs may be characterised by their mathematical form and their linearity.

Linear equations follow the principle of superposition in that if $f_1$ and $f_2$ are two solutions, $\alpha f_1 + \beta f_2$ is also a solution where $\alpha$ and $\beta$ are constants. If coefficients of the dependent variable or any of its derivatives depend on any of the independent variables, such as $x$ or $y$, the PDE is described as linear, variable-coefficient:

- linear: $\frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} = g(x, y)$,
- linear, variable-coefficient: $\frac{\partial f}{\partial x} + \beta \frac{\partial f}{\partial y} = g(x, y)$,
- nonlinear: $\alpha \frac{\partial f}{\partial x} + \beta \frac{\partial f}{\partial y} = g(x, y)$.

Nonlinear PDEs pose additional solution problems when compared with linear PDEs. Details of these problems and methods to deal with them are discussed in later sections.

The form of a simple linear PDE with no more than two independent variables, may be mathematically characterised by the discriminant $B^2 - 4AC$ (see table 3.1) of the general, second-order PDE:

$$A \frac{\partial^2 f}{\partial x^2} + B \frac{\partial^2 f}{\partial x \partial y} + C \frac{\partial^2 f}{\partial y^2} + D \frac{\partial f}{\partial x} + E \frac{\partial f}{\partial y} + F f = G$$

The function $G$ determines whether the PDE is described as homogeneous: $G = 0$, or non-homogeneous: $G \neq 0$.

Parabolic and hyperbolic PDEs both describe systems that evolve through time. Their difference, which happens to be numerically subtle, is the range of signal propagation speed throughout the solution. For parabolic PDEs, the speed is infinite, allowing a localised perturbation to spread throughout the solution instantly. For hyperbolic PDEs, the speed is finite. Fig. 3.1 defines the dependency grid, shown by
CHAPTER 3. NUMERICAL METHODOLOGIES

Figure 3.1: Domain of dependence with regard to signal propagation speed

the shaded area, for solution point \( P \). Fig. 3.1(a) represents hyperbolic PDEs and fig. 3.1(b), parabolic PDEs.

All of the equations solved herein are either elliptic or parabolic hence hyperbolic equations and methods pertaining to their solution are not discussed further.

Initial data

There are many ways to provide initial data for a problem. They differ depending on the character of the PDE and in some cases may limit the choice of solution method used. Fig. 3.2 illustrates the solution domains imposed by various sets of initial conditions for parabolic PDEs.

Fig. 3.2(a) shows a pure initial value problem, also known as a Cauchy problem. Initial data is defined for the line of infinite length at \( t = 0 \):

\[
  u = f(x, 0), \quad -\infty < x < \infty.
\]

Fig. 3.2(b) shows a periodic initial value problem. Initial data is defined along the line of length \( 2\pi \) at \( t = 0 \):

\[
  u = f(x, 0), \quad -\pi \leq x < \pi.
\]

Fig. 3.2(c) shows a mixed problem whereby the initial data is defined for a pair of
Figure 3.2: Solution domain imposed by data conditions
perpendicular intersecting lines of infinite length at $t = 0, x = 0$:

$$u = \begin{cases} f(x, 0), & 0 \leq x < \infty, \\ f(0, t), & t \geq 0. \end{cases}$$

Fig. 3.2(d) shows a pure boundary value problem or Dirichlet problem. Initial data is defined along a line of finite length at zero time and along two perpendicular lines at the boundaries of the solution domain:

$$u = \begin{cases} f(x, 0), & 0 \leq x \leq 1, \\ f(0, t), & t \geq 0, \\ f(1, t), & t \geq 0. \end{cases}$$

Elliptic PDEs are non-evolutionary and are solved in a closed solution domain. An example of an elliptic PDE is the Poisson equation, for which two solution methods are described in §3.4.

Method types

There are two basic groups of finite difference solution methods available for parabolic PDEs:

- Explicit
  - Forward Time Centred Space (FTCS)

- Implicit
  - Backward Time Centred Space (BTCS)
  - Crank-Nicolson (CN)
  - Alternating Direction Implicit (ADI)
CHAPTER 3. NUMERICAL METHODOLOGIES

Locally One Dimensional (LOD)

Table 3.2 contrasts some of their properties. ADI and LOD are specially adapted methods suitable for multi-dimensional problems only. All methods require that the solution be stepped forward in time. Therefore a solution domain at 1000 units of time cannot be calculated directly. Instead, pre-determined time steps are taken to advance the solution. For example, using a time step of size 10, 99 intermediate solutions are calculated before the 1000th unit of time is reached.

Using an unconditionally stable implicit method generally allows much larger steps in time to be taken, possibly several power-orders of magnitude larger. However the simplicity of an explicit method and the corresponding increase in solution speed per time level may outweigh the benefits of large leaps in time. An additional factor, the signal propagation speed should also be mentioned. By referring to table 3.2, it would seem that parabolic PDEs are incorrectly modelled by explicit methods. However, in reality the amount of information that propagates at infinite speed is numerically negligible. The signal propagation speed does not appear to limit the choice of method used.

The type of initial conditions supplied can be a limiting factor on the choice of solution method. For example, an aperiodic initial value problem rules out the use of implicit finite difference methods. Reasoning for this is given in §3.2.2.

Discrete characteristics

Several factors govern all finite difference solution methods:

- accuracy
- stability
- convergence
- consistence

Accuracy is found through a combination of the finite difference truncation error term, known as the order of accuracy and the size of the relative grid spacing, that is, the distance between the discrete mesh points. In general terms, the order of error is the rate at which the approximation error tends toward zero as the grid spacing is decreased.

presuming that the initial conditions are defined at time level zero
The size, shape and coordinate system of the spatial grid and the number of points allocated affect the high frequency behaviour of the solution. The grids used here are all symmetric (if two dimensional), regular and are based on the Cartesian coordinate system. The grid spacing imposes a limit on the maximum frequency component that may be represented by the solution and this is inversely proportional to the size of the spacing. A problem which may arise is caused by the lack of freedom of the highest frequency waves. These waves are severely restricted by the grid and may only travel either horizontally or vertically in a regular 2D Cartesian grid. This affects accuracy and may imprint itself on the solution as a spurious additional symmetry on the fine structure of the field. These problems may be tackled by various means, including removal of the highest frequencies, introduction of an irregular grid concentrating the mesh at known high frequency areas, or simply decreasing the overall grid point spacing to compensate.

Stability is an embedded part of the method type. For a method to be conditionally or unconditionally stable, the local truncation errors must not increase unbounded with increasing time levels.

A numerical method is defined as being convergent if the solution obtained approaches that of the original continuous PDE as the grid spacings approach zero.

Consistency is guaranteed if the truncation error produced by the approximation of the PDE tends to zero as the grid spacings approach zero independently. In other words, the approximation must tend towards the exact continuous PDE as the grid spacings are reduced. This is not always the case and is independent of the stability of the solution method. Stability and consistence combined are sufficient but not necessary conditions for convergence.

3.1.2 Finite differences

Consider the Taylor series expansion of $f(x)$ at $x = x_0$:

$$f(x) = f(x_0) + \frac{1}{1!} f'(x_0)(x - x_0) + \frac{1}{2!} f''(x_0)(x - x_0)^2 + \ldots$$

(3.1)

Changing notation to allow discrete representation, (3.1) becomes

$$f_{i+1} = f_i + f'_i(h) + \frac{1}{2} f''_i(h)^2 + \ldots + \frac{1}{n!} f^{(n)}_i(h)^n + \ldots$$

where the subscript $i$ denotes discrete points in space separated by $h$. The heat equation, (1.13), derived in §1.1 is a continuous parabolic 1D diffusion equation. Re-stated
here in compact notation,

\[ U_t = \alpha U_{xx}. \]  \hspace{2cm} (3.2)

Subscripts \( t \) and \( x \) are used in the text to denote a partial derivative and its order.

To represent (3.2) as a discrete FDE, FDAs of the partial derivatives are sought. Using the superscript \( n \) to denote discrete points in time separated by \( \tau \), the Taylor series of \( f^{n+1} \) at \( f^n \):

\[
f^{n+1}_t = f^n_t + \frac{1}{2} f_{tt}^n \tau^2 + \frac{1}{6} f_{ttt}^n \tau^3 + \ldots
\]

can be arranged with respect to \( f_t^n \):

\[
f_t^n = \frac{f^{n+1}_t - f^n_t}{\tau} - \frac{1}{2} f_{tt}^n \tau + \ldots
\]

\[
= f_t^{n+1} - f_t^n - O(\tau).
\]  \hspace{2cm} (3.3)

Higher order derivatives are truncated and an error term is substituted. Equation (3.3) is a forward time FDA of \( f_t \) approximated to the first order. The error term is the lowest order of the truncated part of the Taylor series.

To find a second order space derivative, forward and backward space series' are combined to eliminate the unwanted first order partial derivative. The Taylor series \( f_{i+1} \) at \( f_i \) expands to

\[
f_i^{n+1} = f_i^n + f_x^n h + \frac{1}{2} f_{xx}^n h^2 + \frac{1}{6} f_{xxx}^n h^3 + \frac{1}{24} f_{xxxx}^n h^4 + \ldots
\]  \hspace{2cm} (3.4)

and \( f_{i-1} \) at \( f_i \):

\[
f_i^n = f_i^n - f_x^n h + \frac{1}{2} f_{xx}^n h^2 - \frac{1}{6} f_{xxx}^n h^3 + \frac{1}{24} f_{xxxx}^n h^4 - \ldots
\]  \hspace{2cm} (3.5)
CHAPTER 3. NUMERICAL METHODOLOGIES

Summing (3.4) and (3.5) with respect to $f_{xx|x}^n$ gives

$$\frac{f_{xx|x}^n}{h^2} = \frac{f_{i+1}^{n} - 2f_{i}^{n} + f_{i-1}^{n}}{h^2} - \frac{1}{12}f_{xxxx|x}^n h^2 + \ldots$$

$$= \frac{f_{i+1}^{n} - 2f_{i}^{n} + f_{i-1}^{n}}{h^2} - O(h^2). \quad (3.6)$$

The first order FDA of the time derivative (3.3) may be improved upon in several ways. For explicit methods a special time stepping scheme such as the leap frog method [19] can be used. See §3.6 for a description. Essentially, this method involves three levels of time as opposed to the standard two level FDA. For one dimensional PDEs, there is an extension to the standard implicit method formulated by Crank and Nicolson [6] which has a principle truncation error of $O(\tau^3 + \tau h^2)$. An example of this method is shown in §3.2.2. Indeed, the Crank-Nicolson method has been further improved upon by Douglas [34, 44] to give a local truncation error of $O(\tau^3 + \tau h^4)$.

3.2 Methods for 1D parabolic PDEs

3.2.1 Explicit

Explicit methods are conditionally stable, require no special solution techniques and always produce a linear FDE.

The forward-time centred-space (FTCS) method

Using the FDAs produced above, the diffusion equation, (3.2), can be represented as an FDE. Arranging to give new solution points in terms of old solution values gives

$$U_i^{n+1} = \frac{\tau}{h^2} (U_{i+1}^{n} - 2U_i^{n} + U_{i-1}^{n}) + U_i^{n}. \quad (3.7)$$

It is now possible to interpret the grid point dependencies. In this case it is very straightforward. The point $U_i^1$ is dependent upon the values at $U_{i+1}^{0}, U_i^{0}$ and $U_{i-1}^{0}$. This is represented graphically in fig. 3.3.

With Dirichlet boundary conditions:

$$U = \begin{cases} f(x,0) & -X \leq x \leq X, \\ f(\pm X, t) & t \geq 0 \end{cases}$$

$^2$backward-time
all of the values of the line $U(x, \tau)$ are found directly and independently of each other.

Convergence analysis involves substituting the truncation errors discarded in finding the FDAs, back in to the FDE. The equation produced is known as the modified differential equation (MDE). This can then be analysed to observe the behaviour as $\tau$ and $h$ both tend to zero. In this case the FDE is found to be convergent [19].

An explanation of the von Neumann method for stability analysis is given in §3.2.2 for the backward-time centred-space FDE of the diffusion equation. Stability analysis of (3.7) can be found in [19, 34]. The outcome of this analysis states that for the propagation to be stable,

$$0 < \alpha \frac{\tau}{h^2} \leq \frac{1}{2}.$$

Thus, $\tau$ must be positive and less than or at most equal to $h^2/2\alpha$.

This condition is somewhat analogous to the Courant-Friedrichs-Lewy stability condition [5] imposed upon hyperbolic FDEs:

$$\tau \leq \frac{h}{|v|}$$

where $v$ is the fastest propagation velocity supported by the solution domain.

The values of $\tau$ and $h$ should be chosen carefully to produce an efficient propagation that is also stable and accurate. Using a smooth Gaussian initial condition with boundaries held at zero, suitable data for such a problem is suggested as: boundaries at $x = \pm 5$, $h = \alpha = 0.1$ and

$$U(x, t) = \begin{cases} 
  e^{-x^2} & \text{for } -5 \leq x \leq 5, \quad t = 0, \\
  0 & \text{for } x = \pm 5, \quad t \geq 0.
\end{cases}$$

With $\tau = 0.03$, the solution will be stable. A sample output of an implementation of this method to solve the above problem is shown in fig. 3.4. As an illustration of instability caused by too large a value of $\tau$, fig. 3.5 is also included where $\tau = 0.055$.  

Figure 3.3: Dependency grid for FTCS method
CHAPTER 3. NUMERICAL METHODOLOGIES

The stability limit for this method, under these conditions, is \( \tau \leq 0.05 \).

### 3.2.2 Implicit

Formation of a generic implicit method based on the diffusion equation (3.2) is similar in terms of FDE production to the FTCS method. The difference appears when producing an appropriate FDA for the second order space derivative. Explicit schemes always produce FDEs that depend only on points found in a previous solution domain. In contrast, implicit schemes always produce FDEs that contain at least some references to solution points in the same solution domain as the FDE itself. Indeed, each point in the new solution domain is directly dependent on the solution of its neighbours and thus indirectly dependent on the solution of every other point in the new solution domain. This mutual dependence produces a set of simultaneous equations at each time level.

For Dirichlet problems, the simultaneous equations form a tridiagonal matrix problem which may be resolved using the Thomas algorithm [19, 51], see §3.5. For aperiodic Cauchy problems, the region of dependence is unbounded and the set of equations cannot be solved. Periodic Cauchy problems are bounded but have circular dependence and cannot be described by a tridiagonal matrix. In such cases, the simultaneous equations can be resolved using cyclic reduction methods [41].

#### The backward-time centred-space (BTCS) method

Consider the diffusion equation, (3.2) and the FDA of \( f_{xx_{i}}^{n} \), (3.6). Using (3.6) at time level \( n + 1 \) instead of \( n \) produces an implicit spatial FDA. This is combined with the standard first order time FDA to give

\[
\frac{U_{i}^{n+1} - U_{i}^{n}}{\tau} = \frac{U_{i+1}^{n+1} - 2U_{i}^{n+1} + U_{i-1}^{n+1}}{h^{2}}.
\]

Equation (3.8) is an implicit FDE of the diffusion equation. Rearranged for unknown points in terms of known points yields

\[
(1 + 2d)U_{i}^{n+1} - d(U_{i+1}^{n+1} + U_{i-1}^{n+1}) = U_{i}^{n},
\]

where \( d = \alpha \tau / h^{2} \). The convergence and consistence of (3.9) is established in [19]. The solution point dependency grid is shown in fig. 3.6.

A set of simultaneous equations are found in the production of a new solution.
Figure 3.4: Stable solution graph of diffusion equation using FTCS method
($h = 0.1, \alpha = 0.1, \tau = 0.03$, 100 time steps taken)

Figure 3.5: Unstable solution graph of diffusion equation using FTCS method
($h = 0.1, \alpha = 0.1, \tau = 0.055$, 100 time steps taken)
domain. With $h = 2.0$ and $U(\pm 3, t) = 0$, the solution yields

\begin{align}
    bU_{-2}^1 + aU_{-1}^1 &= cU_{-2}^0, \\
    aU_{-2}^1 + bU_{-1}^1 + aU_0^1 &= cU_{-1}^0, \\
    aU_{-1}^1 + bU_0^1 + aU_1^1 &= cU_0^0, \\
    aU_0^1 + bU_1^1 + aU_2^1 &= cU_1^0, \\
    aU_1^1 + bU_2^1 &= cU_2^0
\end{align}

(3.10)

where $a = \alpha \tau$, $b = -2\alpha \tau - h^2$ and $c = -h^2$. This system can be represented in matrix form as

\begin{equation}
    CU^{n+1} = P
\end{equation}

(3.11)

where $U^{n+1}$ and $P$ are vectors and $C$ is a sparse square matrix containing zeros except for the central, first upper and first lower diagonals. $C$ contains the coefficients on the LHS of (3.10), $P$ contains the previous solution values on the RHS.

The Thomas algorithm is employed to resolve this particular matrix problem, thereby finding the vector $U^{n+1}$. The propagation continues in this fashion, with a tridiagonal matrix to manipulate at each time level. Care must be exercised when delivering the matrix values to the Thomas algorithm as the matrix $C$ can be stored, for reasons of efficiency, as three diagonal vectors, the central vector containing one more value than the other two. This was found to be an easy area for trivial mistakes to occur and render an otherwise working algorithm useless. Similarly, problems may arise when dealing with time and space dependent coefficients which (3.2) does not have. The correct value of $n$ and especially $i$ to use in calculating such coefficients is not always entirely obvious and an example of such a problem is shown in chapter 4.

An output graph of a scheme implementing the BTCS method is shown in fig. 3.7. To enable a stability comparison to be made, the spatial coefficients and initial condition are mirrored from the explicit example. A time step of more than a factor of
ten larger than the stability limit imposed on the explicit FTCS method was used, \( \tau = 0.55 \). There are no instabilities present when using this method and proof of unconditional stability is provided below.

**Von Neumann stability analysis**

There are various methods to perform stability analysis on the finite difference representations of the original continuous equation. The von Neumann method is presented here to establish the stability of the BTCS FDE, (3.9). Modification of the following procedures allow stability analysis to be performed on other linear FDEs.

Briefly explained, each point function component of the FDE can be replaced by a complex Fourier series. The general Fourier component produced, can then be investigated for stability. If the solution of this new equation is bounded throughout time, either conditionally or unconditionally, then the solution of the FDE is either conditionally or unconditionally stable respectively.

Generally, the exact solution of a 1D method, for a single time step, can be repre-
sented as

\[ f_{i}^{n+1} = \hat{G} f_{i}^{n} \]

where \( \hat{G} \) is generally a complex operator, known as the amplification factor. At any time \( t \), where \( t = n\tau \),

\[ f_{i}^{n} = \hat{G}^{n} f_{i}^{0} \]

For the solution to remain bounded and therefore stable,

\[ ||\hat{G}|| \leq 1. \quad (3.12) \]

The complex Fourier series of an arbitrary continuous distribution, is given by

\[ f(x, t) = \sum_{m=-\infty}^{\infty} C_{m} e^{ik_{m}x}. \]

Replacing \( x \) with \( ih \) and considering a single harmonic from this series gives

\[ f_{i}^{n} = Ce^{ik_{h}i}, \quad (3.13) \]

where the spatial frequency index is dropped for clarity. Equation (3.13) can be used to generate replacements for the point functions in the FDE. For the BTCS FDE, these are

\[ f_{i}^{n+1} = Ce^{ik_{h}i}, \]
\[ f_{i+1}^{n+1} = Ce^{ik_{h+1}i} = Ce^{ik_{h}} f_{i}^{n+1}, \]
\[ f_{i-1}^{n+1} = Ce^{ik_{h-1}i} = Ce^{-ik_{h}} f_{i}^{n+1}. \quad (3.14) \]

The BTCS representation (3.9) of the 1D diffusion equation can now be transformed using these generalized Fourier series components. Substituting equations (3.14) into (3.9) gives

\[ U_{i}^{n+1} \left[ 1 - d \left( e^{ikh} + e^{-ikh} - 2 \right) \right] = U_{i}^{n}. \]

Using the trigonometric identity:

\[ \cos(kh) = \frac{e^{ikh} + e^{-ikh}}{2} \]
and reintroducing the concept of $\hat{G}$ gives

\[
U_j^{n+1} = \frac{1}{1 - 2d(\cos(kh) - 1)} U_j^n
\]

\[
= \frac{1}{1 - \gamma} U_j^n
\]

\[
= \hat{G} U_j^n.
\]

Restoring the subscript to $k$ and carrying it through to $\gamma$ gives

\[
\gamma_m = 2d(\cos(k_m h) - 1).
\]

Since $k_m \propto m$, the full range of $\gamma$ over the entire Fourier harmonic range becomes

\[
2d(\cos(\pi) - 1) \leq \gamma \leq 2d(\cos(2\pi) - 1),
\]

\[
-4d \leq \gamma \leq 0.
\]

For any positive value of $d$,

\[
\frac{1}{1 + 4d} \leq \|\hat{G}\| \leq 1.
\]

Consequently the stability condition imposed in (3.12) is met. The BTCS FDE is unconditionally stable for all positive values of $\alpha r / h^2$. The size of time step used with this method is governed entirely by accuracy.

**The Crank-Nicolson method**

Approximations used in the BTCS method are of first order accuracy in time and second order accuracy in space. Crank and Nicolson suggested that a method where both time and space derivatives could be approximated to the second order, would be both more consistent with the continuous equation, and by definition, more accurate. Consider the second order approximation of $f_{xx}$. It is derived using a centred space method involving three points in space. Applying this concept to time yields an $O(\tau^2)$ accurate approximation of $f_t$, however three levels of time are involved. Crank and Nicolson realised that the extra complication of a third real time level was unnecessary. Their solution was to produce an FDA centred at an imaginary in-between time level, $n + 1/2$. 

The Taylor series of $f_t^n$ at $f_t^{n+1/2}$:

$$f_t^n = f_t^{n+1/2} - f_t^{n+1/2} \frac{\tau}{2} + \frac{1}{2} f_{tt}^{n+1/2} \left( \frac{\tau}{2} \right)^2$$

$$- \frac{1}{6} f_{ttt}^{n+1/2} \left( \frac{\tau}{2} \right)^3 + \ldots$$

and $f_t^{n+1}$ at $f_t^{n+1/2}$:

$$f_t^{n+1} = f_t^{n+1/2} + f_t^{n+1/2} \frac{\tau}{2} + \frac{1}{2} f_{tt}^{n+1/2} \left( \frac{\tau}{2} \right)^2$$

$$+ \frac{1}{6} f_{ttt}^{n+1/2} \left( \frac{\tau}{2} \right)^3 + \ldots$$

are combined to yield the $f_t^{n+1/2}$ FDA with second order accuracy:

$$f_t^{n+1/2} = \frac{f_t^{n+1} - f_t^n}{\tau} + \frac{1}{2a} f_{tt}^{n+1/2} \tau^2 + \ldots$$

$$= \frac{f_t^{n+1} - f_t^n}{\tau} + O(\tau^2). \quad (3.15)$$

For purposes of symmetry, space derivatives must also be expressed at the imaginary time level. The production of an FDA for $f_{xx}^{n+1/2}$ involves averaging over the two real time levels, $n$ and $n + 1$:

$$f_{xx}^{n+1/2} = \frac{f_{xx}^{n+1} + f_{xx}^{n}}{2h^2} - \frac{2f_i^{n+1} + f_i^{n+1} + f_i^{n}}{2h^2}.$$  \quad (3.16)

Using (3.15) and (3.16) to implement the Crank-Nicolson method produces the FDE:

$$a(U_{i+1}^{n+1} + U_i^{n+1}) - (2a + b)U_i^{n+1} = (2a - b)U_i^n - a(U_{i+1}^n + U_{i-1}^n) \quad (3.17)$$

where $a = \alpha \tau$ and $b = 2h^2$. The solution domain dependency grid is shown in fig. 3.8. Equation (3.17) can be resolved using the Thomas method, noting that in this case, the vector $P$ is a sum of values at each vector point as opposed to the single value.
used in the BTCS scheme. Stability analysis establishes the FDE to be stable for all positive values of $\alpha$, $h$ and $\tau$.

### 3.2.3 Evolution of nonlinear diffusion

As a learning exercise, the Crank-Nicolson method was implemented for a nonlinear diffusion equation:

\[ U_t + UU_x = \alpha U_{xx}. \tag{3.18} \]

FDAs for $f_{t_i}^{n+1/2}$ and $f_x^{n+1/2}$ are shown above. A second order FDA of $f_x^n$ is found using centred difference Taylor expansions as detailed earlier:

\[ f_x^n = \frac{U_{i+1}^n - U_{i-1}^n}{2h} + O(h^2). \tag{3.19} \]

On first examination of (3.18), the production of the term $UU_x$ seems relatively simple. By using the Crank-Nicolson method, $UU_x = (U_{i+1}^{n+1/2} - U_{i-1}^{n+1/2})$ would appear correct. In addition, $U_i^{n+1/2}$ can be represented by $\frac{1}{2}(U_i^{n+1} + U_i^n)$. Hence, the product at the intermediate time level yields

\[ [UU_x]_i^{n+1/2} = \frac{1}{8h}(U_i^{n+1} + U_i^n)(U_{i+1}^{n+1} - U_{i-1}^{n+1} + U_{i+1}^n - U_{i-1}^n). \]

Expansion of this term results in a nonlinear FDE. Products of unknowns are produced which require iteration techniques at every time level. There are many methods available to implement this iteration, ranging from substituting the old value to Newton's and the Secant method. An alternative and somewhat easier method would be to produce a linear FDE from the nonlinear PDE. A completely explicit method would do this whereby the FDE would only contain products dependent on the previous solution domain. However, it is possible to express the product in a different manner.

As the continuous partial derivatives are expressed in terms of truncated Taylor series expansions, this is also applied to the $U$ term. Also, the Crank-Nicolson notion of $n + 1/2$ is applied to the product as a whole rather than its constituent parts. Consider the function $U(x,t)$ expanded in a Taylor series about $U_i^{n+1}$:

\[ U(x,t) = U_i^{n+1} - U_i^{n+1}(\tau) + \frac{1}{2}U_{tt_i}^{n+1}(\tau^2) - \frac{1}{6}U_{ttt_i}^{n+1}(\tau^3) + \ldots \]
Similarly, \( U(x, t + \tau) \) expanded about \( U_t^n \) yields
\[
U(x, t + \tau) = U_t^n + U_{tt}^n(\tau) + \frac{1}{2} U_{tt}^n(\tau^2) + \frac{1}{6} U_{ttt}^n(\tau^3) + \ldots
\]

Using (3.19) at time level \( n + 1 \) rather than \( n \) gives \( U_{x}^{n+1} \). In this case, the term \( [U_{x}]^n \) is a product of two infinite series'. Cross multiplication must be applied prior to any approximation. Products for both old and new levels of time are
\[
[U_{x}]^n = U_{i}^{n+1}U_{x}^{i}U_{i}^{n+1}U_{x}^{n}(\tau) + U_{i}^{n+1}U_{x}^{n}(\tau^2),
\]
\[
[U_{x}]^{n+1} = U_{i}^{n+1}U_{x}^{n+1} + U_{i}^{n+1}U_{x}^{n+1}(\tau) + U_{i}^{n+1}U_{x}^{n+1}(\tau^2).
\]

Averaging over the two time levels yields
\[
[U_{x}]^{n+\frac{1}{2}} = \frac{1}{2} U_{i}^{n+1} + \frac{1}{2} U_{i}^{n+1}(\tau^2) + O(\tau, h^2).
\]

Unfortunately this term is accurate only to the first order in time, thus losing the potential accuracy gain of the Crank-Nicolson method. Results shown below were produced using this term.

Since this experiment, it has become apparent that a method accurate to the second order in time exists. Consider \( U_{x}^{n} \) expressed as
\[
U_{x}^{n} = \frac{1}{2} U_{x}^{2}.
\]

To satisfy the Crank-Nicolson method, \( U(x, t + \frac{\tau}{2})^2 \) is required. This can be found by the product of the expansion of \( U(x, t) \) and \( U(x, t + \tau) \) at \( t + \tau/2 \):
\[
U(x, t)U(x, t + \tau) = U(x, t + \frac{\tau}{2})^2 + 2 \left( \frac{\tau^2}{2} \right) U(x, t + \frac{\tau}{2})U(x, t + \tau) + \ldots
\]
\[
U(x, t + \frac{\tau}{2})^2 = U(x, t)U(x, t + \tau) + O(\tau^2).
\]

Substituting (3.21) back into (3.20) gives
\[
\frac{1}{2} U_{x}^{2}U_{x}^{n+\frac{1}{2}} = \frac{1}{2} \left( \frac{U_{i+1}^{n+1}U_{i+1}^{n} - U_{i-1}^{n+1}U_{i-1}^{n}}{2h} \right) + O(\tau^2, h^2).
\]

If this FDA had been used in the FDE, a more accurate progression would have resulted.
In retrospect, the exercise was of great value as it uncovered a potential problem which may have affected later work. The numerical theory proved to be sound but its initial implementation in code gave results showing a lack of nonlinear behaviour. The error was traced to use of the initial condition, $e^{-\beta x^2}$ where $\beta$ was of the order of 10. This was originally done to avoid an increase in absolute grid size. As $\beta$ renormalises $x$, the diffusion effects were amplified to such an extent, that the nonlinear behaviour was completely covered up. On increasing the grid size to allow a normalised initial condition, the nonlinearity was finally seen. Fig. 3.9 shows the solution as it evolves through time.

### 3.3 Methods for 2D parabolic PDEs

By definition, adding an extra physical dimension to a problem of complexity $O(N)$ causes an increase in computational effort to at least $O(N^2)$. In fact the increase is even higher as the solution of each point has a greater number of dependencies.

Explicit methods when applied to 2D PDEs do not generally increase the complexity of the problem. The solution, per point, is simply a function of the previous solution domain, as with 1D PDEs.
In contrast, implicit methods of solution applied to a 2D PDE greatly increase the complexity. At each time level, every solution point is dependent on the value of every other solution point within the solution domain. The resultant set of simultaneous equations form a banded penta-diagonal matrix. Resolving such a matrix requires a high degree of compute power. To address this problem, a third group of methods have been derived combining both implicit and explicit concepts. They are known as alternating direction implicit (ADI) or splitting up methods.

3.3.1 Alternating direction implicit (ADI)

The concept of ADI extols the idea that a 2D solution domain of size $I \times J$ may be treated as $J$ 1D solutions of length $I$ and $I$ 1D solutions of length $J$. Once this is established the method is easy to follow.

The solution grid is effectively cut up into longitudinal and latitudinal strips of single point width. A direction is chosen and each strip in that direction is solved using an implicit method, through the resolution of a tridiagonal matrix. Subsequently each strip in the other direction is solved in the same way, effectively producing an overall solution that embodies information from the entire grid. The signal propagation speed is infinite within each strip but is finite across strip boundaries. The advantage of an ADI method over an explicit method is that stability of the solution is unconditional, as provided by a purely implicit method. The only conceivable drawback of such a method, with respect to an implicit method, is the previously mentioned finite signal speed. However, as first mentioned in §3.1 and since illustrated, a parabolic PDE is adequately modelled using a method with finite signal propagation speed.

Literature concerning ADI methods [29, 34, 19] suggest several differing schemes each based upon the concept above. Examples include the Stabilization, Predictor-Corrector and Component-by-Component methods. The stabilization scheme tends to be the generic ADI method and has several realizations. ADI was initiated by Douglas [7], Peaceman [37] and Rachford [8] and has been developed further by, amongst others, Bagrinovskii, Godunov, Yanenko, Samarski, Dyakonov, Saulyev and Marchuk [29, 28, 30]. Of the methods mentioned above, all form consistent FDEs, are of second order accuracy, and are unconditionally stable. An exception to this claim arises when the $x$ and $y$ derivative operators upon the dependent variable are either dependent on time or non-commutative. Under such general conditions, the component-by-component method retains second order accuracy. It is therefore the preferred method.
Homogeneous PDEs

Consider a homogeneous 2D parabolic PDE,

$$\psi_t + \hat{A}\psi = 0,$$

where the operator $\hat{A}$ represents a combination of partial space derivatives. In two dimensions, $\hat{A} = \hat{A}_x + \hat{A}_y$ where both $\hat{A}_x$ and $\hat{A}_y$ are positive.

A technique similar to that used by Crank-Nicolson in 1D may be used for axial separation in 2D. The axial space differentials are solved independently at separate half levels of time:

$$\psi^{n+\frac{1}{2}} - \psi^n + \frac{\hat{A}_x^{n+\frac{1}{2}}}{\tau} \left( \psi^{n+\frac{1}{2}} + \psi^n \right) = 0,$$

$$\psi^{n+1} - \psi^{n+\frac{1}{2}} + \frac{\hat{A}_y^{n+\frac{1}{2}}}{\tau} \left( \psi^{n+1} + \psi^{n+\frac{1}{2}} \right) = 0.$$  (3.22)

Solving for $\psi$ and removing intermediate solution points gives

$$\psi^{n+1} = T^{n+\frac{1}{2}}\psi^n,$$

$$T^{n+\frac{1}{2}} = \left( E + \frac{\tau}{2} \hat{A}_x^{n+\frac{1}{2}} \right)^{-1} \left( E - \frac{\tau}{2} \hat{A}_x^{n+\frac{1}{2}} \right) \cdot \left( E + \frac{\tau}{2} \hat{A}_y^{n+\frac{1}{2}} \right)^{-1} \left( E - \frac{\tau}{2} \hat{A}_y^{n+\frac{1}{2}} \right)$$

where $E$ represents the identity matrix. The operator $T^{n+1/2}$ can be expanded as a power series in $\tau$:

$$T^{n+\frac{1}{2}} = E - \tau \hat{A}^{n+\frac{1}{2}} + \frac{\tau^2}{2} \left[ \left( \hat{A}_x^{n+\frac{1}{2}} \right)^2 + 2 \hat{A}_y^{n+\frac{1}{2}} \hat{A}_x^{n+\frac{1}{2}} + \left( \hat{A}_y^{n+\frac{1}{2}} \right)^2 \right] - \ldots$$  (3.23)

provided that

$$\frac{\tau}{2} \left| \hat{A}_x^{n+\frac{1}{2}} \right| \leq 1, \quad \frac{\tau}{2} \left| \hat{A}_y^{n+\frac{1}{2}} \right| \leq 1.$$  (3.24)

If the axial space operators $(\hat{A}_x, \hat{A}_y)$ commute, (3.23) can be reduced to:

$$T^{n+\frac{1}{2}} = E - \tau \hat{A}^{n+\frac{1}{2}} + \frac{\tau^2}{2} \left( \hat{A}_y^{n+\frac{1}{2}} \right)^2 - \ldots$$  (3.25)
As \( |T^{n+1/2}| \leq 1 \), the difference scheme is unconditionally stable provided that \( \psi \) is sufficiently smooth. With commutative operators (3.25), second order accuracy in \( \tau \) is achieved. Non-commutative operators (3.23) only provide accuracy of an order \( O(\tau) \). To alleviate this problem, Marchuk developed the scheme further to provide unconditional second order accuracy in \( \tau \).

### 3.3.2 Component-by-component splitting up

Consider the single cycle time interval \( n \Rightarrow n + 1 \) extended to cover the range \( n - 1 \Rightarrow n + 1 \). Two cycles of (3.22) can be used to move the solution forward this distance. However, to achieve second order accuracy regardless of commutativity, the second cycle should proceed with the axial operators reversed:

\[
\begin{cases}
\frac{\psi^{n+1} - \psi^n}{\tau} + \hat{\Lambda}^n_x \left( \psi^{n+1} + \psi^n \right) = 0, \\
\frac{\psi^{n} - \psi^{n+1}}{\tau} + \hat{\Lambda}^n_y \left( \psi^{n+1} + \psi^n \right) = 0, \\
\frac{\psi^{n+1} - \psi^{n+\frac{1}{2}}}{\tau} + \hat{\Lambda}^n_x \left( \psi^{n+\frac{1}{2}} + \psi^n \right) = 0, \\
\frac{\psi^n - \psi^{n+\frac{1}{2}}}{\tau} + \hat{\Lambda}^n_y \left( \psi^{n+\frac{1}{2}} + \psi^n \right) = 0.
\end{cases}
\]

Removing intermediate solution levels and considering the scheme in terms of \( T^n \) gives

\[
\psi^{n+1} = T^n \psi^{n-1},
\]

\[
T^n = \left( E + \frac{\tau}{2} \hat{\Lambda}^n_x \right)^{-1} \left( E - \frac{\tau}{2} \hat{\Lambda}^n_x \right) \cdot \left( E + \frac{\tau}{2} \hat{\Lambda}^n_y \right)^{-1} \left( E - \frac{\tau}{2} \hat{\Lambda}^n_y \right) \cdot \left( E + \frac{\tau}{2} \hat{\Lambda}^n_y \right)^{-1} \left( E - \frac{\tau}{2} \hat{\Lambda}^n_y \right) \cdot \left( E + \frac{\tau}{2} \hat{\Lambda}^n_x \right)^{-1} \left( E - \frac{\tau}{2} \hat{\Lambda}^n_x \right)
\]

Providing conditions (3.24) hold, \( T^n \) can be expressed as a power series in \( \tau \):

\[
T^n = E - 2\tau \hat{\Lambda}^n + \frac{(2\tau)^2}{2} (\hat{\Lambda}^n)^2 - \ldots
\]

This full component-by-component scheme provides second order accuracy, regardless of operator commutativity.
CHAPTER 3. NUMERICAL METHODOLOGIES

In terms of implementation, it should be noted that two time steps are required to achieve second order accuracy with non-commutative operators. The intermediate, physical time level \( \psi^n \) should be ignored as a solution domain, its accuracy only achieving \( O(\tau) \) rather than \( O(\tau^2) \).

Non-homogeneous PDEs

To test a finite difference method and its implementation, the particular method can be used to solve an equation with a known analytical solution. For example, a simple PDE,

\[
U(x, y, t)_{t-1} + U(x, y, t)_{xx} + U(x, y, t)_{yy} = g(x, y, t),
\]

(3.26)
can be assigned an arbitrary analytical solution, \( f(x, y, t) \) provided there is freedom to choose \( g(x, y, t) \). The RHS can be found by solving the continuous derivatives \( f_t, f_{xx} \) and \( f_{yy} \) analytically. This value is then substituted back into (3.26) as \( g(x, y, t) \). The initial condition, \( f(x, y, 0) \), is also calculated.

Once this has been done, (3.26) can be solved numerically and the numerical solution, \( U(x, y, t) \), compared with the original analytical solution, \( f(x, y, t) \). Analysis of the behaviour of these two solutions can lead to insights into the validity and accuracy of the particular method and its implementation. As the RHS is non-zero, the equation is non-homogeneous and requires a slightly different technique. The following modification can be used with the component-by-component method. Equation (3.26) can be written as

\[
\psi_t + \hat{\Lambda} \psi = g
\]

where in our case, \( \hat{\Lambda} = \nabla^2 \). The two cycle scheme is extended to incorporate \( g \):

\[
\left( E + \frac{T}{2} \hat{\Lambda}_x^n \right) \psi^{n-\frac{1}{2}} = \left( E - \frac{T}{2} \hat{\Lambda}_x^n \right) \psi^{n-1},
\]

\[
\left( E + \frac{T}{2} \hat{\Lambda}_y^n \right) (\psi^n - \tau g^n) = \left( E - \frac{T}{2} \hat{\Lambda}_y^n \right) \psi^{n-\frac{1}{2}},
\]

\[
\left( E + \frac{T}{2} \hat{\Lambda}_y^n \right) \psi^{n+\frac{1}{2}} = \left( E - \frac{T}{2} \hat{\Lambda}_y^n \right) (\psi^n + \tau g^n),
\]

\[
\left( E + \frac{T}{2} \hat{\Lambda}_x^n \right) \psi^{n+1} = \left( E - \frac{T}{2} \hat{\Lambda}_x^n \right) \psi^{n+\frac{1}{2}}.
\]

Using similar means to the homogeneous scheme, this modified method is shown in [29] to be consistent, unconditionally stable and of second order accuracy in \( \tau \).
CHAPTER 3. NUMERICAL METHODOLOGIES

3.4 Methods for elliptic PDEs

A Poisson type equation solver was required in two different contexts during this research. While solving the model in part II, one of the two time invariant formulae was broken down into a Poisson problem. For the model in part III, a Poisson solver formed an integral part of the time progression.

3.4.1 Iteration

A PDE that is described using a FDE operating in an aperiodic solution domain cannot be represented fully in the frequency domain. For a problem of this type, there appear to be two distinct methods available. Iteration, described here and ADI.

During experimentation, a Poisson solver was used to determine a physical invariant, used to test solution accuracy. As this procedure was completely independent of the evolution of the model, iteration appeared to be more appropriate as it had a simpler implementation than ADI, albeit a potentially slower solution rate. Using Taylor series' the non-homogeneous Laplace equation or Poisson equation,

$$\nabla^2 U = f,$$  \hspace{1cm} (3.27)

can be transformed into an FDE

$$\frac{U_{i-1,j} - 2U_{i,j} + U_{i+1,j}}{h^2} + \frac{U_{i,j-1} - 2U_{i,j} + U_{i,j+1}}{k^2} = f$$

where the subscript \(j\) denotes the point location in \(y\) space, \(k\) representing the relative grid spacing. Rearranging for \(U_{i,j}\) gives

$$U_{i,j} = \frac{k^2(U_{i-1,j} + U_{i+1,j}) + h^2(U_{i,j-1} + U_{i,j+1}) - h^2k^2f}{2(h^2 + k^2)}.$$  \hspace{1cm} (3.28)

Gauss-Seidel

To transform the FDE (3.28) into a Gauss-Seidel model of iteration, \(\pm U_{i,j}\) is added to the RHS and a 2D sweep strategy is adopted to avoid the unnecessary storage of a second solution domain. The level of iteration is represented by \(m\).

Starting at the bottom left of the 2D mesh, see fig. 3.10, a sweep in \(x\) followed by an increment in \(y\) ensures that \(U_{i+1,j}\) and \(U_{i,j+1}\), relative to \(U_{i,j}\), are always available at iteration level \(m\). Conversely, \(U_{i-1,j}\) and \(U_{i,j-1}\) have already been solved at level \(m+1\). Solution point \(U_{i,j}\) is replaced by the freshly iterated value, transforming (3.28)
CHAPTER 3. NUMERICAL METHODOLOGIES

into the Gauss-Seidel model of iteration:

\[ U_{i,j}^{m+1} = U_{i,j}^m + \alpha - U_{i,j}^m \]  

(3.29)

where \( \alpha \) is the sweep transformed value of \( U_{i,j} \). The later two components of (3.29) are combined to give \( \Delta U_{i,j}^m \),

\[ \Delta U_{i,j}^m = \frac{k^2(U_{i-1,j}^{m+1} + U_{i+1,j}^m) + h^2(U_{i,j-1}^{m+1} + U_{i,j+1}^m) - h^2k^2f - 2(h^2 + k^2)U_{i,j}^m}{2(h^2 + k^2)} \]

Finally the iterated value of \( U_{i,j}^{m+1} \) is given by

\[ U_{i,j}^{m+1} = U_{i,j}^m + \Delta U_{i,j}^m \]

Successive over-relaxation (SOR)

To increase the convergence of the solution, over relaxation can be introduced via the factor \( \omega \),

\[ U_{i,j}^{m+1} = U_{i,j}^m + \omega \Delta U_{i,j}^m \]

The Gauss-Seidel iteration method then becomes the SOR method. For an arbitrary elliptic PDE, the optimum value of \( \omega \) can be found empirically and lies between \( 1.0 \leq \omega \leq 2.0 \).
CHAPTER 3. NUMERICAL METHODOLOGIES

Approximation

Common to all iteration techniques is approximation cut-off. The exact solution is never reached. A practical limit is found numerically when the precision of the storage type is exhausted, but this degree of precision is rarely required.

There are several methods available to determine when an appropriate level of accuracy has been achieved. The simplest involves using a fixed number of iteration steps, after which the current solution is unconditionally accepted. Dynamic methods generally provide a theoretically more pleasing approach but impose additional computational cost. A simple comparison between successive iterations can provide a crude error term, $\epsilon$,

$$\sum_{i} \left[ \sum_{j} |\Delta U_{i,j}| \right] < \epsilon.$$

An improvement on this may be a weighted comparison:

$$\sum_{i} \left[ \sum_{j} \left| \frac{\Delta U_{i,j}}{U_{i,j}} \right| \right] < \epsilon.$$

The error is monitored and once it drops below a pre-determined level, the iteration ceases.

3.4.2 Fast Fourier transform (FFT)

If the partial derivatives of the Poisson equation and its solution domain are periodic, then the simplest and fastest numerical solution method is to use a spatial to frequency domain transform. The PDE is transformed into a simple function, of which the expansion is transformed back into the spatial domain.

Each component of the Poisson equation (3.27) undergoes a Fourier transform:

$$\tilde{\mathcal{F}}(f) = g,$$

$$\tilde{\mathcal{F}} \left( \frac{\partial^2 U}{\partial x^2} \right) = - (k_x)^2 \chi,$$

$$\tilde{\mathcal{F}} \left( \frac{\partial^2 U}{\partial y^2} \right) = - (k_y)^2 \chi$$

where $\tilde{k}$ represents frequency components in Fourier space. Substituting the trans-
CHAPTER 3. NUMERICAL METHODOLOGIES

formed values back into (3.27) gives

$$\chi_{i,j} = \frac{g}{-(k_x^2 + k_y^2)}.$$  

The inverse Fourier transform of the solution, $\chi$, obtains the spatial solution, $U$,

$$U = \mathcal{F}^{-1}(\chi).$$

The method requires some consideration regarding the Fourier transforms, especially efficient use of any FFT routines available. The main problems are the general lack of two dimensional FFTs and the use of real numbers. The former problem is solved simply by performing one dimensional sweeps in each direction [3]. This is shown by the identities:

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y)e^{-i(k_x x + k_y y)}dx
dy = \int_{-\infty}^{\infty} e^{-ik_y y} \left[ \int_{-\infty}^{\infty} f(x,y)e^{-ik_x x}dx \right]dy,$$

$$\mathcal{F}_{x,y}(f(x,y)) = \mathcal{F}_y(\mathcal{F}_x(f(x,y))).$$

The latter can be attacked in several ways. If no real FFT is available, it is possible to encode the data to use a half sized complex FFT, rather than wasting all of the imaginary number space. However, the FFT library actually used contained a real FFT solver, thus removing this complication.

3.5 Resolution of a tridiagonal matrix

The Thomas algorithm uses a tridiagonal matrix to solve systems of simultaneous equations containing three unknowns each.

Library functions which implement this algorithm are generally available. However, as this method was to feature heavily in the solution methods implemented in this research it was deemed worthwhile to grasp the theory and create an independent implementation. It was expected that this would be easier to incorporate than a black box Thomas algorithm.

The coefficients of the simultaneous equations, at solution points $i+1$, $i$ and $i-1$, are used to form a tridiagonal matrix. The matrix is manipulated in such a way that the bottom row becomes an equation containing a single unknown. This equation can then be solved and the solution used as a seed for the other rows of the matrix.

The tridiagonal matrix, $C$, the solution vector, $F$, and the previous solution value
vector, \( \mathbf{P} \) are combined as

\[
\mathbf{C}\mathbf{F} = \mathbf{P}.
\]  

(3.30)

Consider the central, first upper and first lower diagonals of \( \mathbf{C} \) referenced as vectors: \( \mathbf{D}, \mathbf{U} \) and \( \mathbf{L} \) which represent diagonal, upper and lower, respectively. All other matrix elements are zero. Equation (3.30) in expanded form yields

\[
\begin{bmatrix}
D_1 & U_1 & 0 & 0 & 0 & 0 & 0 \\
L_2 & D_2 & U_2 & 0 & 0 & 0 & 0 \\
0 & L_3 & D_3 & U_3 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & L_{I-2} & D_{I-2} & U_{I-2} & 0 \\
0 & 0 & 0 & 0 & L_{I-1} & D_{I-1} & U_{I-1} \\
0 & 0 & 0 & 0 & 0 & L_I & D_I
\end{bmatrix}
\begin{bmatrix}
F_1 \\
F_2 \\
F_3 \\
\vdots \\
F_{I-2} \\
F_{I-1} \\
F_I
\end{bmatrix}
= 
\begin{bmatrix}
P_1 \\
P_2 \\
P_3 \\
\vdots \\
P_{I-2} \\
P_{I-1} \\
P_I
\end{bmatrix}
\]

Let matrix \( \mathbf{C} \) be augmented with vector \( \mathbf{P} \). Gauss elimination is then used to remove the lower diagonal. The first three elements of the second row are transformed into

\[
\begin{align*}
C'_{21} &= C_{21} - C_{11} \frac{C_{21}}{C_{11}}, & L'_2 &= L_2 - D_1 \frac{L_2}{D_1}, \\
C'_{22} &= C_{22} - C_{12} \frac{C_{21}}{C_{11}}, & D'_2 &= D_2 - U_1 \frac{L_2}{D_1}, \\
C'_{23} &= C_{23} - C_{13} \frac{C_{21}}{C_{11}}, & U'_2 &= U_2 - 0.
\end{align*}
\]

In general, \( \mathbf{C} \) uses the following transformation formula:

\[
C'_{(r)(c)} = C_{(r)(c)} - C_{(r-1)(c)} \frac{C_{(r)(r-1)}}{C_{(r-1)(r-1)}}
\]

for all rows below the second where \( r \) and \( c \) represent row and column, respectively. The first column of the second row, \( C'_{21} (L_2) \), is replaced with a zero, the second term \( C'_{22} (D_2) \) is modified and the third term \( C'_{23} (U_2) \) and above remain unchanged. Vector \( \mathbf{P} \) is modified in the same way as the centre diagonal. This transformation is
cascaded down through the matrix completely replacing the lower diagonal with zeros:

\[
\begin{bmatrix}
    D_1 & U_1 & 0 & 0 & 0 & 0 & 0 \\
    0 & D_2 & U_2 & 0 & 0 & 0 & 0 \\
    0 & 0 & D_3 & U_3 & 0 & 0 & 0 \\
    \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
    0 & 0 & 0 & D_{I-2} & U_{I-2} & 0 & 0 \\
    0 & 0 & 0 & 0 & D_{I-1} & U_{I-1} & 0 \\
    0 & 0 & 0 & 0 & 0 & D_I & 0 \\
\end{bmatrix} \begin{bmatrix}
    F_1 \\
    F_2 \\
    F_3 \\
    \vdots \\
    F_{I-2} \\
    F_{I-1} \\
    F_I \\
\end{bmatrix} = \begin{bmatrix}
    P'_1 \\
    P'_2 \\
    P'_3 \\
    \vdots \\
    P'_{I-2} \\
    P'_{I-1} \\
    P'_I \\
\end{bmatrix}
\]

All dependencies from the last row are removed, thus

\[D'_I F_I = P'_I\]

can be solved directly to find \(F_I\). This value is located at the far bottom right of the solution domain and is also known as \(U_{I-1}\) in row \(I-1\), one up from the bottom. Hence, \(F_{I-1}\) can be solved using the seed solution \(F_I\). Similarly, each row above is solved until all values of the vector \(F\) are found.

**Implementation**

The Thomas algorithm used to resolve a tridiagonal matrix with indices, \(-I \to I\) was implemented as follows\(^3\):

1. For each row \(-I < r \leq I:\)
   - \(L_r = L_r / D_{r-1},\)
   - \(D_r = D_r - L_r U_{r-1},\)
   - \(P_r = P_r - L_r P_{r-1}.\)
2. \(F_I = P_I / D_I.\)
3. For each row \(I > r \geq -I:\)
   - \(F_r = (P_r - U_r F_{r+1}) / D_r.\)

\(^3\)as the \(L\) vector is removed, it is reused to store the row reduction multipliers
3.6 Leap frog time stepping method

The leap frog method is a time integration technique and as such is independent of the type of spatial method used and the dimensional order of the problem. As stated in §3.2.1, the standard two level time FDA has first order accuracy in time. To improve upon this, a three level method in time can be used. Crank and Nicolson found it possible to use an imaginary intermediate time level but this approach forces an otherwise explicit method to become implicit. The leap frog method can thus be used to advantage within explicit methods.

An FDA of $U_i$ using three levels of time produces a term accurate to the second order:

$$U_i = \frac{U_{i}^{n+1} - U_{i}^{n-1}}{2\tau} + O(\tau^2).$$  \hspace{1cm} (3.31)

Equation (3.31) is analogous to the FDA of $U_z$, (3.19). When combined with a general forward-time based spatial FDA, expressed in terms of $U_{i}^{n+1}$, the leap frog modified dependency grid is as shown in fig. 3.11. Although this is defined as a three level method, its implementation requires only two levels of storage.

The point dependency of $U_{i}^{n+2}$ on $U^n$ is a single point regardless of spatial operators. Thus the solution at $U_{i}^{n+2}$ can simply replace the old solution $U^n$, effectively removing the need to store a third solution domain.

Initialisation

Leap frog methods are complicated by requiring a two level (in time) startup procedure. To achieve this and retain second order accuracy is problematic. It may be possible to use an analytical technique to provide the first solution domain but this is clumsy if arbitrary initial conditions are used.
Two practical solutions are possible using the standard time stepping method. However as this is only accurate to the first order, second order accuracy is lost. The simpler approach is to accept first order accuracy for the first time step. Although this has accuracy implications for every following time step, it may produce more accurate results than sustained first order accurate evolution. A more complicated approach involves using a slow startup technique, whereby a first order two level method is used with a time step, \( \tau_2 \), considerably smaller than the main time step, \( \tau_3 \). With \( n\tau_2 = \tau_3 \), \( n \) evolutions are made using the two level method, after which point the leap frog method takes over.

**Odd-even behaviour**

The leap frog method is prone to odd-even decoupling where the error at every other time step becomes independent. Once this has occurred, the method may become unstable or produce very poor results unless the odd-even steps are re-coupled using a two level time step technique.
Part II

Linear problem
Chapter 4

Joint behaviour of IG and Rossby waves

A phenomenon known as blocking is believed to occur in the atmosphere due to the interaction of inertio-gravity and Rossby waves. A mathematical model describing this interaction is solved using techniques described in earlier chapters and the results presented.

4.1 Physical description

Using (1.49) found in [10], the influence of a geostrophically balanced Rossby perturbation, $\eta$, on the spread of a packet of inertia-gravity (IG) waves can be described by

$$2i \frac{\partial \psi}{\partial t} + 2i \left( \frac{\partial(1 + \eta)}{\partial x} \frac{\partial \psi}{\partial y} - \frac{\partial(1 + \eta)}{\partial y} \frac{\partial \psi}{\partial x} \right) + \nabla^2 (1 + \eta) \psi - \psi \nabla^2 \eta = 0$$

(4.1)

where the surface height is $h = 1 + \eta$. Solutions to equation (4.1) are sought to investigate the relationship between IG and Rossby waves.

4.2 Numerical description

Equation (4.1) is a 2D, homogeneous, parabolic, linear, complex PDE. It is a boundary value, propagation problem as illustrated in fig. 3.2(d). The solution is defined at a particular point in time, usually zero, and is marched forward by use of a numerical solution technique. The solution domain is of finite size with boundaries defined...
throughout time.

There are various methods that may be employed to solve such an equation. A 2D explicit method similar to the 1D FTCS method described in §3.2.1 is a possibility. However the advantage of simplicity is balanced by the disadvantage of conditional stability provided by such a method. A fully implicit scheme such as a Crank-Nicolson method in 2D can also be used, however this would cause the model to be transformed into a penta-diagonal matrix problem. Considering the finite level of compute power available and the existence of ADI methods, the use of a pure implicit method was dismissed. ADI methods offer an interesting alternative. They promise the unconditional stability associated with implicit methods but at a much cheaper computational cost. Of the ADI methods investigated [19, 29, 34], the most accurate appeared to be the component-by-component. This method overcame a subtle shortcoming in other ADI methods involving the commutativity of axial space derivatives. The component-by-component method was able to achieve unconditional second order accuracy.

As the research progressed, it was expected that feedback and progress reports to Falkovich would cause subsequent modifications to initial conditions and the Rossby perturbation, $\eta$. Consequently, conditions such as space derivative commutativity and the non-evolutionary nature of $\eta$ could not be assumed. Hence the desirability of a general, unconditionally stable method with second order accuracy.

The bracketed quantity in (4.1):

\[ 2i \left( \frac{\partial(1 + \eta)}{\partial x} \frac{\partial \psi}{\partial y} - \frac{\partial(1 + \eta)}{\partial y} \frac{\partial \psi}{\partial x} \right) \]

reduces to zero under conditions of complete symmetry. However, in this case, it is possible to further reduce the PDE to a single space dimension. For some experimentation, this was indeed the case. In two dimensions, (4.1) was used in its entirety to allow freedom for rotational effects to propagate.

The non-evolutionary function, $\eta$, could be integrated using either numerical or analytical methods. An analytical partial derivative potentially has a higher level of accuracy than a numerical approximation. However, the overall effect on the solution is negligible.

### 4.2.1 Solution accuracy

The component-by-component finite difference method is unconditionally stable. Accuracy, a measurement of the difference between the approximate numerical solution and the actual solution, is governed by a combination of:
The partial derivatives were approximated using Taylor series expansions, some of which are outlined in chapter 3. The FDAs produced using this approach each have an associated approximation or truncation error. Without truncation, continuous partial derivatives could not be represented in discrete terms. Truncation errors indicate the order of approximation provided by the method. The global error at each numerical solution point cannot be determined. Instead, the order of approximation provides a measure of the change in global error relative to a change in grid point spacing.

Resolution error is not an independent quantity. It describes the effect of truncation error on grid point spacing. Consider both a first and second order approximated FDE of the same PDE. Reducing grid spacing (increasing grid resolution) reduces the global error of the solution at a linear rate for the first order FDE and at a quadratic rate for the second order FDE. This leads to the conclusion that for a given solution domain resolution, the higher the order of the solution method, the higher the accuracy of the numerical solution. The propagation truncation error in $\tau$ is similarly linked to the accuracy of the individual time steps.

Using a symmetric Cartesian coordinate solution domain as an example, fig. 4.1, the highest possible frequency component that the grid can support without aliasing has a period of length $2h$. The movement of such a wave is severely restricted. It may only propagate horizontally or vertically. A high frequency component of period
\( \sqrt{(4h)^2 + (2h)^2} \) can propagate in four directions. Low frequency wave components are better modelled and can propagate in a more realistic fashion. The highest frequency components can introduce areas of non-physical behaviour into the solution. This problem can be addressed by using high resolution solution domains or employing an anti-alias filter to remove the highest frequencies. The later approach is usually only convenient if the solution domain is available in the frequency domain, however, it is subject to varying accuracy losses.

The problem was posed with Dirichlet boundary conditions. Under these conditions, boundary leakage is inevitable due to the use of a finite aperiodic solution domain. This kind of problem enforces an artificial limit on the scope of the solution at the boundaries of the solution domain. To minimise energy leakage through this physical anomaly, the absolute size of the solution domain should be large enough to stop the solution from sensing the edges. Boundary leakage can affect the accuracy of the solution, or if bad enough it can be responsible for destroying the physical laws underpinning the model. The results illustrated in §4.4 deliberately use a tight boundary to illustrate the onset of this effect after several units of time.

### 4.2.2 Solution speed

Every measure that can be employed to increase the accuracy of the solution except the order of the FDE has a negative impact on speed of numerical solution.

For a 2D PDE, a symmetric increase in the number of grid points is accompanied by a quadratic decrease in solution speed. Other things remaining equal, an increase in absolute grid size will not affect solution speed, but will reduce resolution and subsequently accuracy. Hence an increase in the absolute grid size is normally compensated by an increase in the number of grid points, producing a subsequent drop in solution speed.

A successful numerical simulation must balance accuracy with solution speed to determine suitable implementation characteristics. This usually requires a considerable amount of engineering.
4.3 Sequential implementation

Equation (4.1) is expanded in order to isolate the partial derivatives:

\[
\left[ 2i \left( \frac{\partial}{\partial t} + \frac{\partial \eta}{\partial x} \frac{\partial}{\partial y} - \frac{\partial \eta}{\partial y} \frac{\partial}{\partial x} \right) + (1 + \eta) \left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) + 2 \left( \frac{\partial(1 + \eta)}{\partial x} \frac{\partial}{\partial x} + \frac{\partial(1 + \eta)}{\partial y} \frac{\partial}{\partial y} \right) \right] \psi = 0. \tag{4.2}
\]

The component-by-component method is a two cycle method, each cycle marching forward a single time step. To fit this method, (4.2) must be re-written in the form

\[
\left( \hat{\Lambda}_t + \hat{\Lambda}_x + \hat{\Lambda}_y \right) \psi = 0.
\]

The individual operators acting upon \( \psi \) are

\[
\hat{\Lambda}_t = 2i \frac{\partial}{\partial t}, \\
\hat{\Lambda}_x = -2i \frac{\partial \eta}{\partial y} \frac{\partial}{\partial x} + (1 + \eta) \frac{\partial^2}{\partial x^2} + 2 \frac{\partial(1 + \eta)}{\partial x} \frac{\partial}{\partial x}, \\
\hat{\Lambda}_y = 2i \frac{\partial \eta}{\partial x} \frac{\partial}{\partial y} + (1 + \eta) \frac{\partial^2}{\partial y^2} + 2 \frac{\partial(1 + \eta)}{\partial y} \frac{\partial}{\partial y}.
\]

Initially, \( \eta \) described a non-evolutionary Gaussian wave packet:

\[
\eta = ae^{(-x^2 - y^2)}, -1 \leq a \leq +1
\]

where the sign of \( a \) was an indication of the pressure level of the wave packet. A cyclone is the result of using \( a < 0 \) and an anticyclone is modelled by \( a > 0 \). The partial derivatives of \( \eta \) were found analytically:

\[
\frac{\partial(1 + \eta)}{\partial x} = \frac{\partial \eta}{\partial x} = -2\eta x, \\
\frac{\partial(1 + \eta)}{\partial y} = \frac{\partial \eta}{\partial y} = -2\eta y
\]

and hard coded into the simulation. In retrospect, this was probably not the best method. The generality lost and inconvenience caused by using an analytical technique
outweighed the very slight computational penalty of numerical approximation:

\[
\begin{align*}
\frac{\partial \eta}{\partial x} &= \frac{\eta_{i+1,j} - \eta_{i-1,j}}{2h}, \\
\frac{\partial \eta}{\partial y} &= \frac{\eta_{i,j+1} - \eta_{i,j-1}}{2k}.
\end{align*}
\]

Indeed, later experiments were implemented using a numerical approximation method similar to this. Incorporating the partial derivatives of \(\eta\), the spatial operators in full are

\[
\begin{align*}
\hat{\lambda}_x &= 4\eta(\eta y - x) \frac{\partial}{\partial x} + (1 + \eta) \frac{\partial^2}{\partial x^2}, \\
\hat{\lambda}_y &= -4\eta(\eta x + y) \frac{\partial}{\partial y} + (1 + \eta) \frac{\partial^2}{\partial y^2}.
\end{align*}
\]

The method employs a 2D solution domain to model the surface of \(\psi\). Solution points are defined at grid points \(i\) and \(j\) with grid spacings, \(h\) and \(k\), respectively. A complete solution domain was computed at each time step \(n\), spaced through time by the constant, \(\tau\).

### 4.3.1 Time stepping

The following equation sets are organised in terms of step number from 1 to 4. Steps 1 and 2 form the first cycle of the method, steps 3 and 4, the second cycle. The relationship between step number and relative discrete time level is illustrated below:

- [Step 1] \(n - 1 \Rightarrow n - 1/2\),
- [Step 2] \(n - 1/2 \Rightarrow n\),
- [Step 3] \(n \Rightarrow n + 1/2\),
- [Step 4] \(n + 1/2 \Rightarrow n + 1\).

The FDEs for \(\psi\) at each time step are

\[
\begin{align*}
\text{[Step 1]} & \quad (\hat{\lambda}_t + \hat{\lambda}^n_x) \psi^{n-3/4} = 0, \\
\text{[Step 2]} & \quad (\hat{\lambda}_t + \hat{\lambda}^n_y) \psi^{n-1/4} = 0, \\
\text{[Step 3]} & \quad (\hat{\lambda}_t + \hat{\lambda}^n_y) \psi^{n+1/4} = 0, \\
\text{[Step 4]} & \quad (\hat{\lambda}_t + \hat{\lambda}^n_x) \psi^{n+3/4} = 0.
\end{align*}
\]
Modifying the standard time differential FDAs to fit the ADI scheme directly yields

\[
\begin{align*}
\text{[Step 1]} & \quad \tilde{A}_t \psi^{n-3/4} = 2i \frac{\psi_{i,j}^{n-1/2} - \psi_{i,j}^{n-1}}{\tau}, \\
\text{[Step 2]} & \quad \tilde{A}_t \psi^{n-1/4} = 2i \frac{\psi_{i,j}^{n} - \psi_{i,j}^{n-1/2}}{\tau}, \\
\text{[Step 3]} & \quad \tilde{A}_t \psi^{n+1/4} = 2i \frac{\psi_{i,j}^{n+1/2} - \psi_{i,j}^{n}}{\tau}, \\
\text{[Step 4]} & \quad \tilde{A}_t \psi^{n+3/4} = 2i \frac{\psi_{i,j}^{n+1} - \psi_{i,j}^{n+1/2}}{\tau}. 
\end{align*}
\]

To construct the spatial derivatives for each step, the intermediate values of \( \psi \) are replaced by an average of the two nearest real half time levels. This is represented by the function, \( \Psi \):

\[
\begin{align*}
\text{[Step 1]} & \quad \Psi^{n-3/4} = \frac{\psi^{n-1} + \psi^{n-1/2}}{2}, \\
\text{[Step 2]} & \quad \Psi^{n-1/4} = \frac{\psi^{n-1/2} + \psi^{n}}{2}, \\
\text{[Step 3]} & \quad \Psi^{n+1/4} = \frac{\psi^{n} + \psi^{n+1/2}}{2}, \\
\text{[Step 4]} & \quad \Psi^{n+3/4} = \frac{\psi^{n+1/2} + \psi^{n+1}}{2}. 
\end{align*}
\]

Using \( \Psi \) for brevity, with \( i \) and \( j \) indices promoted to the real \( \psi \) functions, the space component is created for each time step:

\[
\begin{align*}
\tilde{A}_x^n \psi^{n-3/4} = 4\eta_{h,j}(ij \kappa - ih) \left( \frac{\psi_{i+1,j}^{n-3/4} - \psi_{i-1,j}^{n-3/4}}{2h} \right) \\
&\quad + (1 + \eta_{i,j}) \left( \frac{\psi_{i+1,j}^{n-3/4} - 2\psi_{i,j}^{n-3/4} + \psi_{i-1,j}^{n-3/4}}{h^2} \right), \\
\tilde{A}_y^n \psi^{n-1/4} = -4\eta_{h,j}(ih \kappa + jh) \left( \frac{\psi_{i,j+1}^{n-1/4} - \psi_{i,j-1}^{n-1/4}}{2k} \right) \\
&\quad + (1 + \eta_{i,j}) \left( \frac{\psi_{i,j+1}^{n-1/4} - 2\psi_{i,j}^{n-1/4} + \psi_{i,j-1}^{n-1/4}}{k^2} \right),
\end{align*}
\]
CHAPTER 4. JOINT BEHAVIOUR OF IG AND ROSSBY WAVES

\[
\hat{\Delta}_y^n \Psi^{n+1/4} = -4\eta_{i,j}(ih + jk) \left( \frac{\Psi_{i,j+1}^{n+1/4} - \Psi_{i,j-1}^{n+1/4}}{2k} \right)
\]

[Step 3]

\[
+ (1 + \eta_{i,j}) \left( \frac{\Psi_{i,j+1}^{n+1/4} - 2\Psi_{i,j}^{n+1/4} + \Psi_{i,j-1}^{n+1/4}}{k^2} \right),
\]

\[
\hat{\Delta}_y^n \Psi^{n+3/4} = 4\eta_{i,j} (jk - ih) \left( \frac{\Psi_{i+1,j}^{n+3/4} - \Psi_{i-1,j}^{n+3/4}}{2h} \right)
\]

[Step 4]

\[
+ (1 + \eta_{i,j}) \left( \frac{\Psi_{i+1,j}^{n+3/4} - 2\Psi_{i,j}^{n+3/4} + \Psi_{i-1,j}^{n+3/4}}{h^2} \right).
\]

Substituting (4.7), (4.11) and (4.15) into (4.3) and collecting terms yields

\[
(-C_x + D_x + E_x)\Psi_{i+1,j}^{n-1/2}
\]

\[
+ (B - 2E_x)\Psi_{i,j}^{n-1/2}
\]

[Step 1]

\[
+ (C_x - D_x + E_x)\Psi_{i-1,j}^{n-1/2} = \left( C_x - D_x - E_x \right)\Psi_{i+1,j}^{n-1/2}
\]

\[
+ (B + 2E_x)\Psi_{i,j}^{n-1/2}
\]

\[
+ \left( -C_x + D_x - E_x \right)\Psi_{i-1,j}^{n-1/2}.
\]

Substituting (4.8), (4.12) and (4.16) into (4.4) and collecting terms yields

\[
(C_y + D_y + E_y)\Psi_{i,j+1}^{n-1/2}
\]

\[
+ (B - 2E_y)\Psi_{i,j}^{n-1/2}
\]

[Step 2]

\[
+ (-C_y - D_y + E_y)\Psi_{i,j-1}^{n-1/2} = \left( -C_y - D_y - E_y \right)\Psi_{i,j+1}^{n-1/2}
\]

\[
+ (B + 2E_y)\Psi_{i,j}^{n-1/2}
\]

\[
+ \left( C_y + D_y - E_y \right)\Psi_{i,j-1}^{n-1/2}.
\]

Substituting (4.9), (4.13) and (4.17) into (4.5) and collecting terms yields

\[
(C_y + D_y + E_y)\Psi_{i,j+1}^{n+1/2}
\]

\[
+ (B - 2E_y)\Psi_{i,j}^{n+1/2}
\]

[Step 3]

\[
+ (-C_y - D_y + E_y)\Psi_{i,j-1}^{n+1/2} = \left( -C_y - D_y - E_y \right)\Psi_{i,j+1}^{n+1/2}
\]

\[
+ (B + 2E_y)\Psi_{i,j}^{n+1/2}
\]

\[
+ \left( C_y + D_y - E_y \right)\Psi_{i,j-1}^{n+1/2}.
\]
Finally, substituting (4.10), (4.14) and (4.18) into (4.6) and collecting terms yields

\((-C_x + D_x + E_x)\psi_{i+1,j}^{n+1} + (B - 2E_x)\psi_{i,j}^{n+1}\)

[Step 4]  

\(+ (C_x - D_x + E_x)\psi_{i-1,j}^{n+1} = (C_x - D_x - E_x)\psi_{i+1,j}^{n+1/2} + (B + 2E_x)\psi_{i,j}^{n+1/2} + (-C_x + D_x - E_x)\psi_{i-1,j}^{n+1/2}\)

where

\[ B = \frac{2}{\tau}, \]

\[ C_x = iD_x \frac{k}{h}, \quad C_y = iD_y \frac{k}{h}, \]

\[ D_x = -i\eta_{i,j}, \quad D_y = -f\eta_{i,j}, \]

\[ E_x = \frac{1 + m_{i,j}}{2h^2}, \quad E_y = \frac{1 + m_{i,j}}{2k^2}. \]

The coefficients: \( C_{(x,y)}, D_{(x,y)}, E_{(x,y)} \) and the local perturbation, \( \eta \), were all assigned a 2D lookup table. These non-evolutionary values were initialised prior to time stepping. In initial experiments, the coefficients were created as functions that received solution domain coordinates as arguments, and returned the computed value. Although this method was extremely memory efficient, it meant that identical values were computed repeatedly. To recoup some of the memory lost by the lookup table approach, it is possible to optimize the table for symmetric functions. Consider a 2D symmetric function, \( f \). All requests for \( f_{i+1, j} \) can be satisfied by \( f_{i, j} \) thus saving 3/4 of the memory used by the complete table. A significant increase in solution speed occurred after the employment of functional lookup tables.

**4.3.2 Applying the method**

The following routine was used to apply the component-by-component method. See fig. 4.2 for an illustration.

1. For each \( j \) 1D solution strip spanning \( x \) (fig. 4.2(a)) at time \( n - 1 \):
   
   (a) Apply equation (4.19) at each interior point in strip

   (b) Resolve strip for time level \( n - 1/2 \) using the Thomas algorithm

2. For each \( i \) 1D solution strip spanning \( y \) (fig. 4.2(b)) at time \( n - 1/2 \):
CHAPTER 4. JOINT BEHAVIOUR OF IG AND ROSSBY WAVES

Figure 4.2: Component by component solution domain sweeping

(a) Apply equation (4.20) at each interior point in strip

(b) Resolve strip for time level $n$ using the Thomas algorithm

3. For each 1D solution strip spanning $y$ (fig. 4.2(b)) at time $n$:

(a) Apply equation (4.21) at each interior point in strip

(b) Resolve strip for time level $n + 1/2$ using the Thomas algorithm

4. For each 1D solution strip spanning $x$ (fig. 4.2(a)) at time $n + 1/2$:

(a) Apply equation (4.22) at each interior point in strip

(b) Resolve strip for time level $n + 1$ using the Thomas algorithm

4.3.3 Collecting results

The numerical simulation produced a valid solution every two time steps as suggested by the method, however a single simulation would typically complete somewhere in the range of $100 \rightarrow 10000$ time steps. The size of a solution domain was typically of the order of several megabytes ($x$-pts x $y$-pts x 8bytes), hence it was undesirable, and indeed unnecessary, to store every solution on disk. Typically, ten solutions spaced equally throughout the life of the experiment were stored on disk.

Two values were monitored every double time step to ensure that the solution was progressing without problems:
• A single solution value, usually the centre point of the domain: $|\psi(0,0,t)|^2$

• The energy level of the numerical model: $\int (1 + \eta) |\psi|^2 \, dx \, dy$ (see §4.5.1)

If either of these values changed too rapidly, the simulation terminated prematurely and an email was despatched to the author including relevant data regarding the nature of the problem. Indeed, email was also used to indicate successful completion or abnormal termination of the numerical simulations. This feature allowed unattended experimentation with immediate notification if attention was required.

To enable the continuation of a completed simulation from the point at which it ended, the current solution and relevant state data was always saved when either the final solution was computed or the program caught a termination signal sent by the OS. The latter case was a necessary safeguard when running simulations lasting days at a time where regular solution domains were dumped at very sparse intervals of wall-clock time. The saved solution data was in a form suitable to be post-processed by:

• 3D visualisation programs using the FORTRAN 77 UNIRAS graphics library.

• 2D to 1D data splitting programs for 2D graphing software.

• The Poisson solver, used to measure conservation of enstrophy (see §4.5.3)

• The simulation program, to be used as an initial condition.

### 4.4 Results

Results were obtained from a number of experiments which led to further insight into the interactions of Rossby and IG waves. In general, the results showed that neither a bound state nor a collapsing wave cavern occur during evolution, implying that the high and low frequency interaction is not the sole cause of atmospheric blocking.

#### 4.4.1 Capture of an IG wave by a cyclone

**Symmetric Gaussian wave packet in the cyclone**

Initial conditions were set to model the evolution of a broad packet of high frequency IG waves under the influence of a narrow low frequency cyclonic Rossby well:

$$\psi(x, y, 0) = e^{-\frac{x^2}{\alpha^2}},$$

$$\eta(x, y, t) = ae^{-x^2-y^2}, \quad \alpha = \pm 1/2.$$
Numerical solution parameters of $\tau = h = k = 0.05$ were used with a solution grid of size $255^2$ points ($x = \pm 6.35, \ y = \pm 6.35$). Fig. 4.3 illustrates the evolution of $|\psi(0,0,t)|^2$ at the centre of the well through five units of dimensionless time. The evolution shows that the IG packet is captured by the cyclone ($a = -1/2$) and grows in amplitude for a unit of time. In contrast, the IG packet is repelled by the anticyclone ($a = 1/2$), with a fairly constant fall in amplitude over time. Physically, this behaviour can be explained in terms of velocities. Inside the cyclone, the velocity of the IG waves grows due to the depth of the cyclone. The waves spend more time inside the cyclone because of this. It is noted that in the late stages of evolution, the wave packet spreads faster in the presence of the cyclone. A slice through the IG wave packet at $\theta = 0$ is shown in fig. 4.4 for each unit of time. Figures 4.5 through 4.10 are images of the IG wave as it evolves in the presence of a cyclone. Similarly, figures 4.11 through 4.16 show the IG wave in the presence of an anticyclone.

**Asymmetric wave packet near the cyclone**

A second experiment was conducted using an initially asymmetric IG wave packet in the presence of similar cyclonic wells:

$$\psi(x,0) = xe^{-x^2/36},$$
$$\eta(x,t) = ae^{-x^2}.$$  

This problem was solved using a 1D implicit Crank-Nicolson method due to the absence of 2D rotational effects noted in the first experiment. Numerical solution parameters of $\tau = 0.005$ and $h = 0.15$ were used with a solution of size 1025 points ($x = \pm 76.8$). The evolution of the IG wave packet influenced by the cyclone is shown in fig. 4.17 and by the anticyclone in fig. 4.18. The labels represent values of dimensionless time. These results show that the behaviour of the IG wave packet is somewhat different near a Rossby well compared to being actually in it. At initial stages of evolution, the IG packet is attracted to the anticyclone, but soon turns back, all the time increasing in amplitude until it reaches its initial position. From this point on, it moves away from the anticyclone, becoming broader in size and decreasing in amplitude through time. Conversely, the cyclone appears to push the packet away initially. After several units of time, the packet is drawn back toward the cyclone, where it remains.
CHAPTER 4. JOINT BEHAVIOUR OF IG AND ROSSBY WAVES

Figure 4.3: Amplitude of IG wave packet at the centre of the Rossby well

Figure 4.4: Form of IG wave packet inside the cyclone
Figure 4.5: The IG wave packet inside the cyclone at $t = 0$

Figure 4.6: The IG wave packet inside the cyclone at $t = 1$
Figure 4.7: The IG wave packet inside the cyclone at $t = 2$

Figure 4.8: The IG wave packet inside the cyclone at $t = 3$
CHAPTER 4. JOINT BEHAVIOUR OF IG AND ROSSBY WAVES

Figure 4.9: The IG wave packet inside the cyclone at $t = 4$

Figure 4.10: The IG wave packet inside the cyclone at $t = 5$
CHAPTER 4. JOINT BEHAVIOUR OF IG AND ROSSBY WAVES

Figure 4.11: The IG wave packet inside the anticyclone at \( t = 0 \)

Figure 4.12: The IG wave packet inside the anticyclone at \( t = 1 \)
CHAPTER 4. JOINT BEHAVIOUR OF IG AND ROSSBY WAVES

Figure 4.13: The IG wave packet inside the anticyclone at \( t = 2 \)

Figure 4.14: The IG wave packet inside the anticyclone at \( t = 3 \)
CHAPTER 4. JOINT BEHAVIOUR OF IG AND ROSSBY WAVES

Figure 4.15: The IG wave packet inside the anticyclone at $t = 4$

Figure 4.16: The IG wave packet inside the anticyclone at $t = 5$
Figure 4.17: Form of IG wave packet near the cyclone

Figure 4.18: Form of IG wave packet near the anticyclone
Symmetric wave packet influenced by a cyclone-anticyclone dipole

The evolution of an initially symmetric Gaussian IG wave packet was simulated in the
presence of a cyclone-anticyclone dipole:

\[
\psi(x, y, 0) = e^{-\frac{x^2 + y^2}{\sigma}}, \\
\eta(x, y, t) = xe^{-x^2 - y^2}
\]

Numerical solution parameters of \( \tau = h = k = 0.05 \) were used with a grid of size \( 255^2 \)
points \((x = \pm6.35, \ y = \pm6.35)\). The results show that the wave packet adjusts its
position to centralise over the cyclone. There it evolves, decreasing in amplitude and
becoming broader through time. Fig. 4.19 shows the evolution of a slice through the
waveform at \( y = 0 \). Figures 4.20 through 4.25 show images of the IG wave packet as
it evolves through each unit of time.

4.4.2 Production of a cyclone-anticyclone pair by IG packet

Having investigated the effect of a stationary Rossby well on a packet of IG waves, it
is also interesting to consider evolutionary Rossby waves affected by the Coriolis force
in the form of \( \beta \) and corresponding feedback from the evolution of the IG wave packet.
CHAPTER 4. JOINT BEHAVIOUR OF IG AND ROSSBY WAVES

Figure 4.20: The IG wave packet over the cyclone-anticyclone dipole at $t = 0$

Figure 4.21: The IG wave packet over the cyclone-anticyclone dipole at $t = 1$
Figure 4.22: The IG wave packet over the cyclone-anticyclone dipole at \( t = 2 \)

Figure 4.23: The IG wave packet over the cyclone-anticyclone dipole at \( t = 3 \)
CHAPTER 4. JOINT BEHAVIOUR OF IG AND ROSSBY WAVES

Figure 4.24: The IG wave packet over the cyclone-anticyclone dipole at $t = 4$

Figure 4.25: The IG wave packet over the cyclone-anticyclone dipole at $t = 5$
The feedback equation (1.52) was modified in order to provide a moving frame of reference able to track the solution and keep it relative to the centre of the solution grid:

$$\frac{\partial \psi}{\partial t} = \beta \psi \frac{\partial \eta}{\partial x} - \beta \frac{\partial \psi}{\partial x} + i \frac{1}{2} \left( (\eta + 1) \frac{\partial^2 \psi}{\partial x^2} - \psi \frac{\partial^2 \eta}{\partial x^2} \right).$$

The nonlinear evolutionary equation in $\eta$ (1.51) was linearised and also modified to take account of the moving reference frame:

$$\frac{\partial \eta}{\partial x} = \beta \frac{\partial}{\partial x} |\psi|^2.$$

This simplified model was used in the production of a numerical simulation. Having only a single space dimension, a Crank-Nicolson technique was used to solve the problem. Initial conditions consisted of

$$\psi(x, 0) = \frac{1}{2} e^{-x^2},$$

$$\eta(x, 0) = 0.$$

Numerical solution parameters of $\tau = 0.005$ and $h = 0.12$ were used with a grid of size 2049 points ($x = \pm 122.88$). The reference frame moves with velocity, $\beta = 1/8$. Fig. 4.26 shows the evolution of the model, fig. 4.26(a) illustrating the IG wave packet and fig. 4.26(b), the Rossby dipole. As the IG packet evolves, a persistent cyclone-anticyclone dipole is formed. The cyclone attracts the IG packet whilst it spreads and loses amplitude over time. This is consistent with behaviour observed in other experiments.

4.5 Validating the results

There are several potential sources of error in the implementation of a numerical method. These include

- Technical
  - Typographical - an incorrect sign or grid index
  - Programming - exceeding array bounds

- Numerical
Figure 4.26: Interaction between IG and evolutionary Rossby wave packets
CHAPTER 4. JOINT BEHAVIOUR OF IG AND ROSSBY WAVES

- algorithmic - bad implementation or method

- Experimental
  - Stability
  - Consistency - poor grid size or point spacing
  - Accuracy - any of the above errors

To address these problems, several methods are available. In order of complexity, simplest first, these are

1. Monitor energy variance.

2. Compare the numerical and analytical solutions of a PDE that has a known analytical solution.

3. Monitor potential enstrophy variance.

4.5.1 Energy conservation

The conservation of energy can be used to monitor the evolution of a solution with reference to implementation errors and solution accuracy. Energy contained by the model is described by

\[ E = \int \int (1 + \eta) |\psi|^2 \, dx \, dy. \]

In practice this value was computed after the completion of each dual cycle, at each time level where second order accuracy was achieved. To calculate \( E \), a quadrature method was used. Simpson’s rule proved to be a suitable method which was implemented in two dimensions to yield a multi level function:

\[ E = \frac{k}{3} \left[ E_x(-J) + E_x(J) + 4 \sum_{j=1-J}^{J-1,2} E_x(j) + 2 \sum_{j=2-J}^{J-2,2} E_x(j) \right] \]

where

\[ E_x(j) = \frac{h}{3} \left[ E_{xy}(-I,j) + E_{xy}(I,j) + 4 \sum_{i=1-I}^{I-1,2} E_{xy}(i,j) + 2 \sum_{i=2-I}^{I-2,2} E_{xy}(i,j) \right] \]
and

$$\mathcal{E}_{xy}(i,j) = (1 + \eta_{ij}) |\psi_{ij}|^2.$$ 

Solution domain boundaries are represented by $i = \pm I, j = \pm J$.

Unfortunately, the use of $\mathcal{E}$ as a measure of success was inadequate, although it did provide a good indication of simulation problems. A sharp rise in $\mathcal{E}$ over time would indicate an implementation error. In contrast, a sharp fall in $\mathcal{E}$, above a few percent per unit of time would indicate an accuracy problem which might be rectified by an increase in grid size and resolution, and/or a reduction in the size of the time step.

Stable energy in the model proved to be a necessary but nevertheless insufficient condition to verify the results. This was proved by the discovery of an implementation error after the production of results that conserved energy over time.

The mistake was related to computing coefficients of $\psi$ at incorrect grid points. Consider as an example, the solution point equation:

$$\alpha \psi_{i+1,j}^{n+1} + b \psi_{i,j}^{n+1} + c \psi_{i-1,j}^{n+1} = \alpha \psi_{i+1,j}^n + b \psi_{i,j}^n + \gamma \psi_{i-1,j}^n.$$ 

To resolve this equation, the coefficients $\alpha, \beta$ and $\gamma$ should be computed at grid point $i, j$. However, the coefficients $\alpha$ and $\gamma$ had instead been computed at grid points $i+1, j$ and $i-1, j$, respectively. This had a dramatic effect on the results produced but did not show up as an energy problem. As the invariance of energy could not be relied upon as a definite measure of success, further checking procedures were sought.

4.5.2 Using a non-homogeneous equation

A considerable amount of effort was spent implementing a solution method similar to that established in §3.3.2. A PDE is assigned an arbitrary solution and the resulting non-homogeneous PDE is solved numerically. The numerical and analytical solutions are compared to provide insight into the accuracy of the method in general, however the implementation suffered from stability problems which led to inconclusive results.

A simple equation was prepared:

$$(\alpha + \beta y) \frac{\partial^2 U}{\partial x^2} + (\lambda + \gamma z) \frac{\partial^2 U}{\partial y^2} + \frac{\partial U}{\partial t} = g(x, y, t)$$  \hspace{1cm} (4.23)
CHAPTER 4. JOINT BEHAVIOUR OF IG AND ROSSBY WAVES

and assigned an arbitrary solution, \( f \):

\[
U(x, y, t) = f(x, y, t) = ae^{(-x^2-y^2-\epsilon t)}.
\] (4.24)

The partial derivatives were found analytically:

\[
\begin{align*}
U_t &= -\epsilon f, \\
U_{xx} &= 4x^2 f, \\
U_{yy} &= 4y^2 f
\end{align*}
\]

and the initial condition derived from (4.24):

\[
U(x, y, 0) = f(x, y, 0) = ae^{(-x^2-y^2)}.
\]

Hence,

\[
g(x, y, t) = (4x^2(\alpha + \beta y) + 4y^2(\lambda + \gamma x) - \epsilon) f.
\]

The component-by-component method was modified to take account of the RHS of (4.23) and the numerical solutions compared to the real analytical solution. As mentioned above, the numerical solution became unstable after less than a unit of time. However, the solutions were fairly comparable with the numerical solution propagating more rapidly than the analytic solution.

In an attempt to solve the instability problem, the implementation was rewritten using the modelled PDE as the target. It was assigned a similar arbitrary solution and turned into a non-homogeneous PDE. This implementation again proved to be unstable with worse results and a higher degree of instability than the first attempt. Due to these problems, the results of this experimentation could not be used for either positive or negative verification purposes.

4.5.3 Conservation of potential enstrophy

As a final test of accuracy and physical consistence, the law of enstrophy conservation was applied to the model yielding

\[
N = \iint \left( \frac{\nabla^2}{(1 + \eta)} \left( \frac{\partial}{\partial x} - i \frac{\partial}{\partial y} \right) (1 + \eta) \psi \right) \, dx \, dy.
\] (4.25)
Equation (4.25) can be rewritten in terms of a Poisson equation:

\[ \nabla^2 g = \left( \frac{\partial}{\partial x} - i \frac{\partial}{\partial y} \right) (1 + \eta) \psi \tag{4.26} \]

allowing the integral to become

\[ N = \int \int \left| \frac{g}{(1 + \eta)} \right| dx \, dy. \tag{4.27} \]

There are two common finite difference methods available to solve non-propagating elliptic PDEs such as the Poisson equation (4.26) on a finite solution grid. These are:

- ADI
- Iteration

In general terms, ADI is the more pleasing approach and is likely to be faster at producing a solution to a given accuracy. However, iteration is a simpler scheme and is therefore easier and quicker to implement and less prone to implementation errors. As the solution of (4.26) was completely independent of the numerical evolution of the model, the solution speed was not critical.

The Poisson equation was solved using an SOR technique as described in §3.4.1. Converting to FDAs, (4.26) becomes

\[ \frac{g_{i+1,j} - 2g_{i,j} + g_{i-1,j}}{h^2} + \frac{g_{i,j+1} - 2g_{i,j} + g_{i,j-1}}{k^2} = f \]

where

\[ f = \frac{((1 + \eta) \psi)_{i+1,j} - ((1 + \eta) \psi)_{i-1,j}}{2h} - \frac{((1 + \eta) \psi)_{i,j+1} - ((1 + \eta) \psi)_{i,j-1}}{2k}. \]

Rewriting with respect to \( g_{i,j} \) gives

\[ g_{i,j} = \frac{2k^2(g_{i+1,j} + g_{i-1,j}) + 2h^2(g_{i,j+1} + g_{i,j-1}) - 2hk^2f}{4(h^2 + k^2)}. \tag{4.28} \]

**Implementation**

Adopting a sweep strategy transforms (4.28) into

\[ g_{i,j}^{m+1} = g_{i,j}^m + \omega \Delta g_{i,j}^m \]
with \( \omega \) representing the over-relaxation factor and \( m \), the iteration level. The iterative part, \( \Delta g_{i,j}^m \), is determined by:

\[
\Delta g_{i,j}^m = \frac{2k^2(g_{i+1,j}^m + g_{i-1,j}^{m+1}) + 2h^2(g_{i,j+1}^m + g_{i,j-1}^{m+1}) - 2h^2k^2f}{4(h^2 + k^2)} - g_{i,j}^m.
\]

The method required an initial function, \( g^0 \), in the form of a solution grid and an approximation cut-off condition to end the iteration. In the text, final values of \( g \) and \( N \) are represented by \( g^M \) and \( N^M \), respectively. The value of the \( g^0 \) function proved to be a critical factor determining the amount of iteration required to reach \( g^M \) and hence, \( N^M \). Implementation of the SOR adopted a two stage process to optimize the \( g^0 \) solution grid.

Initial functions of \( \psi \) and \( \eta \) were used to compute a value of \( g^0 \) close to the expected final value. This value, \( \tilde{g}^0 \) was used for the second stage of the SOR. As an example, the first stage of the SOR would use

\[
\psi^0 = e^{-x^2 - y^2}/\alpha, \quad \eta^0 = a e^{-x^2 - y^2}, \quad g^0 = 0
\]

to produce a secondary initial grid: \( \tilde{g}^0 \). The computed grid, \( \tilde{g}^0 \) could then be used as an initial condition with evolved solution data of the same domain size and similar \( \psi^0 \) and \( \eta^0 \).

The approximation cut-off point was determined by a two level trigger. Quadrature used to solve (4.27):

\[
N = \frac{k}{3} \left[ N_x(-J) + N_x(J) + 4 \sum_{j=1-J}^{J-1,2} N_x(j) + 2 \sum_{j=2-J}^{J-2,2} N_x(j) \right],
\]

\[
N_x(j) = \frac{h}{3} \left[ N_{xy}(-I,j) + N_{xy}(I,j) + 4 \sum_{i=1-I}^{I-1,2} N_{xy}(i,j) + 2 \sum_{i=2-I}^{I-2,2} N_{xy}(i,j) \right], \quad (4.29)
\]

was computationally expensive, thus it was not performed within the iteration cycle.
until a predefined accuracy trigger level was met:

$$\sum_{\text{all } i} \left[ \sum_{\text{all } j} |\omega \Delta g^{n+1}_i| \right] < 0.1.$$

Subsequent iteration then involved computing the enstrophy integral using the system (4.29). The final cut-off trigger level was determined by the change in enstrophy level per iteration:

$$|N^m - N^{m-1}| < 0.01.$$  

A sanity limit of 10000 iterations was also set, beyond which the program was terminated prematurely.

The over-relaxation factor, $\omega$, also had a substantial effect on the amount of iteration required to achieve an arbitrary level of accuracy. See fig. 4.27 for an illustration. Using too low a value of $\omega$ caused $g$ to approach $g^M$ too slowly, whereas too high a value of $\omega$ caused $g$ to overshoot $g^M$ and oscillate about it with decreasing amplitude per iteration. Empirically, $\omega = 1.90$ proved to be the best general value. This was used by all subsequent enstrophy calculations as individual optimum values dependent on $\psi$, $\eta$ and other simulation characteristics would have taken more research time to find than they would have saved.
Stage two of the implemented SOR method used the pre-computed $\tilde{g}^0$ to compute an enstrophy level for the first $\psi$ solution domain in the results data file. The $\tilde{g}^M$ computed was saved and re-used as the initial value of $g$ for the next $\psi$ solution domain. This continued until a value of $N^M$ was determined for each $\psi$ solution domain.

Results

The results of enstrophy testing proved positive. The level of variance was well below a single percent per unit of time for the results presented in §4.4.

4.6 Parallel implementation

To provide the possibility of producing finer grained results on a larger solution grid, a parallel implementation was considered. Using parameters of $\tau = 0.05$ on a solution containing $256^2$ grid points, 10 units of dimensionless time (200 steps) would take approximately 220 minutes to compute\(^1\). However, simply raising the grid size to a modest $1024^2$ points increases the solution time to 58 hours.

Consideration was given to domain decomposition, a method that enables variable grid point spacing. An increase in solution speed is accomplished by concentrating the solution mesh at areas of high flux whilst reducing the point density in static areas. Hence a reduction in the number of solution points is achieved without an increase in the overall granularity of the solution. This method is most useful in situations where the form of the solution is known in advance. However, even implementing a static variable-spaced solution mesh is a non-trivial exercise. Domain optimization methods were considered unsuitable for this particular problem.

Parallel computing offered a potential solution. Various parallel systems were investigated in §2.4, and as a first step in this direction, PVM was chosen. To gain an increase in solution speed using a distributed environment with high communication costs such as PVM, the first step is to conceptually minimise dependencies in the problem. Once this has been done, it is possible to estimate the degree of parallelisation, if any, that can be exploited.

4.6.1 Example for explicit method

Before examining the parallelisation of an ADI method, it is interesting to consider a simple 2D explicit method. An explicit method has immediate neighbour dependencies\(^1\) assuming a solution rate of 1000 points per second on a lightly loaded Sun SPARCstation 10/30.
CHAPTER 4. JOINT BEHAVIOUR OF IG AND ROSSBY WAVES

Figure 4.28: Geometric data partitioning of a 2D explicit method

only and thus the problem can be resolved using a geometric data partitioning scheme. Emphasis is placed on the actual method used to partition the data with minimisation of nodal interfaces as the goal.

Assuming an $n$ node homogeneous distributed parallel system, the solution domain is split into $n$ equal portions, spread amongst the nodes. The allocated portion of the solution domain remain with each node throughout the simulation. Edge data from each portion must be communicated to the neighbouring nodes at each time step. This causes barrier synchronisation which could cause problems in a heterogeneous environment. However, such problems can be countered by a dynamic implementation whereby a controller process watches the processes and redistributes the solution grid when necessary to minimise nodal idle time. Fig. 4.28 illustrates a partitioning and communication scheme using $n = 4$ and a solution grid of size $8^2$ points.

4.6.2 ADI

Viewed from a parallel perspective, ADI methods pose a completely different problem. Each row or column of the solution grid is dependent on its current and previous state only. Individual rows are completely independent of all other rows in the solution grid.

Considering a single implicit axial sweep direction in isolation, the problem appears to be highly parallel. Nodes can be assigned arbitrary 1D strips of the solution grid and the processing can be accomplished with no inter-nodal communication costs at
all, as there are no dependencies exterior to any node. However, there is a significant amount of communication at the sweep direction changeover. Fig. 4.29 illustrates the movement of data at a directional sweep change with \( n = 4 \) and fig. 4.30 shows the quantity of the solution data that must be exchanged for a given number of nodes.

Due to the nature of a basic ADI method, there is little or no opportunity to interleave the communication with the solution processing. The implementation would inevitably have to process the strips and then transfer data to other nodes whilst the other nodes were trying to transfer data themselves. In PVM, this would create communication saturation. The nature of the communication medium compounds the problem by forcing all of the communication messages to be serialised, hence nodes would spend most of their time waiting for data to process rather than processing data.

To enable any solution speedup using a loosely coupled distributed parallel environment such as PVM, the following problems have to be solved:

- Increase granularity
- Interleave communication with computation

4.6.3 Method parallelisation

As mentioned above, the direct implementation of an ADI method produces a very fine grained problem which is unable to exploit interleaving of communication with computation. The amount of data processing between axial sweeps must be increased at each node to increase granularity. This can be achieved by

1. A reduction in the number of nodes
2. An increase in the overall number of grid points

3. An alternate solution method

The former two methods both constitute a severe decrease in potential speedup. However, a general modification to the solution method, or more drastically a complete change of method, are both possible solutions.

Consider an increase in the number of implicit axial sweeps performed between direction alternation. A general ADI method performs a single sweep per direction, the component by component method performs 2 sweeps per direction. If the number of consecutive axial sweeps could be increased by a factor of 10 or 100, then the overall granularity would be increased by the same amount. In a sequential implementation, such a modification would have no effect on solution speed.

The 2D sequential implementation was modified to allow an arbitrary number of axial sweeps to be performed. This generalisation changed the sequence of sweeps from

\[ X, 2X, 2X, \ldots \] to \[ sX, 2sY, 2sX, \ldots \]

where \( s \geq 1 \).

A direct consequence of this modification is that snapshots of valid solution data are only available every \( 2s \) time steps. The modification is similar in effect to using
the standard method with a time step of $s\tau$, however the accuracy of the solution is likely to be similar to that of using a time step of size $\tau$.

Values of $s \leq 100$ were tested and produced results that matched those found with $s = 1$. Although values of $s > 1$ proved to be empirically correct, the mathematical consequences were not fully investigated. The use of a high $s$ factor provides the opportunity for communication to be interleaved amongst computation. As the individual strips are completely independent of each other, it is possible to process each strip $s$ times before processing the next strip. In this way, once a strip has been fully processed, it may be immediately transferred to the other nodes. Using this model, it is possible to achieve very little process latency under favourable conditions.

### 4.6.4 Speedup realisation

To ascertain the conditions necessary for an increase in solution speed using PVM, the data in table 4.1 may be considered. This information is based upon a $1024^2$ point solution grid with a moderate sweep factor, $s = 20$. Network transfer times are theoretical maximums. Not surprisingly, the worse case scenario is found when using the maximum amount of nodes connected via Ethernet, in this case 16. The necessary transfer of solution data between direction alternations takes 6 seconds to complete. A satisfactory implementation requires that the nodal processing time between changeovers exceeds 6 seconds. For this to occur, the solution processing speed must not exceed

$$\frac{1.25}{6}(1024^2) \approx 220000 \text{ pts/s}.$$ 

Using the best case, 4 nodes connected via FDDI, this value increases to

$$\frac{5}{0.48}(1024^2) \approx 11000000 \text{ pts/s}.$$
Sequential simulations run on a lightly loaded, 4 processor, Sun SPARCstation 1000 compute server with 384Mb of physical memory produced results at a rate of approximately 5000 points per second. It is thus demonstrated that the compute time exceeds the communication time by a factor of 44 using the worse case example above with a moderate s factor.

4.6.5 Implementation

An efficient implementation could be organised as:

Controller

- Spawn children.
- For each time level:
  - Block receive
    * Store solution strip in solution domain.
  - Optionally compute energy and enstrophy.
  - Optionally dump solution domain to file.
- Kill children.
- Dump final solution domain.

Children

- Initialize:
  - Join PVM group.
  - Find out how many children exist and hence which part of the solution domain is to be processed.
  - Calculate time independent functions.
  - Calculate $\psi^0$.
- Strip processing:
  - For each strip:
    * Process $s$ times
    * Send results of processing to relevant peers and controller.
* Receive any data in input message queue. Store for later.
  - Block receive for data until all has arrived.
* Continue strip processing the next time level.

The actual implementation attempted was somewhat simplified. Instead of each child passing individual portions of data to each of its peers, the entire strip was sent to every process including the controller. Not only was this easier to implement, but it should not have had an impact on communication by use of an OS/hardware multicast mechanism. A multicast allows a node to pass an identical message to all nodes using a single network transfer. Unfortunately, the multicast capabilities of PVM did not invoke an OS multicast message in the version used which meant that every message was duplicated \( n \) times. Another simplification let each child hold a full copy of the solution domain that it received prior to allocating the desired portion.

A combination of problems caused the PVM implementation to be permanently shelved. These included the multicast problem mentioned above and a lack of homogeneous workstations available on a single subnet with sufficient memory. As an example, the memory occupied by a solution domain containing \( 1024^2 \) points totals 8Mb. Adding space for storing the previous solution domain and lookup tables for time independent functions increases this memory requirement to 36Mb. Empirically, a figure closer to 54Mb was seen to be consumed during such a simulation. This may be explained by speed optimizations in the compiler causing an increase in run time memory requirements.

Using PVM decreases the array space required for processing on each node by around a factor of the nodes used. However, each pvmSd must store incoming data for the local process. Depending upon the particular method employed, this may yet raise the memory requirement by the size of another solution domain.

### 4.6.6 Conclusions

The component by component ADI method in 2D can withstand high levels of direction implicit computation. Indeed, stability cannot be affected by this generality as using \( s = 100 \) and a time step of \( \tau \) can be compared with using a single time step of size \( 100\tau \). Although a working implementation was not achieved, the research suggests that with appropriate equipment and conditions, it is perfectly possible. However, the scalability of such an implementation must be considered. Unless the size of the solution grid is increased at a rate proportional to added nodes, a point will be reached where adding
a further node will produce either no increase in speed, or finally a decrease. The computational time spent by each node must exceed the interleaved communication time for there to be appreciable speedup. Once this limit is broken, nodal idle time spent in blocking receive mode for data will increase and efficiency will fall.

4.7 Conclusions

4.7.1 Physical behaviour

The results showed no evidence of a bound state or a collapsing wave cavern during evolution. It was hoped that the results would provide evidence to support the theory that IG and Rossby wave interaction is a cause of atmospheric blocking. However, it was shown that their interaction, in isolation of other atmospheric effects, does not itself explain the phenomenon. Had the results proved more exciting, further 2D simulations would have been justified, making use of a more complicated model containing nonlinear elements. Although this did not happen, the results obtained were positive, in that several behaviours were observed that require physical interpretation. Such interpretation is beyond the scope of this thesis. These behaviours include the appearance of a local IG wave packet maximum in the presence of a cyclone, which then reduces in amplitude at a faster rate than a similar IG packet in the presence of an anticyclone. Another notable effect is the production of a sustained cyclone-anticyclone dipole during the evolution of an IG wave packet taking account of the Coriolis effect.

4.7.2 Numerical methodology

The use of various numerical methods, implemented in FORTRAN 77 code proved successful. These included the Crank-Nicolson, Thomas, component by component ADI and SOR iteration methods. The increased effort involved in implementing an ADI-based method over an explicit scheme was rewarded by the unconditional stability of the method. This completely removed stability as an issue while debugging the implementation. Numerical integration and the SOR method used to solve the Poisson problem posed by the conservation of enstrophy aided in the verification of results. Failure of the non-homogeneous ADI solution technique to verify the solution implementation was a disappointment, however the conservation of both energy and enstrophy by the simulations, demonstrate that the results are physically consistent. The lengthy research into both 1D and 2D finite difference methods paid dividends with the intro-
duction of the 1D model with evolutionary $\eta$. Modification of the 2D implementation simply required reference to [32], prepared after the first year of research.

4.7.3 Parallel implementation

If a model of higher complexity had been warranted, the use of a parallel solution technique would have been imperative. At the time of this experimentation, the fastest local machine, a Sun SPARCstation 10/30 was only capable of sustaining a solution rate of around 1000 points per second using the implemented ADI method on a linear model. A 2D system with evolutionary $\eta$ would have reduced this rate by a factor of at least two. Introducing a nonlinearity into the model would have compounded the problem by a factor of ten at the very least, due to the employment of an iteration technique at each time step. Hence a single time step using a $1024^3$ point solution grid would have taken at least six hours to compute under these conditions. A modest 100 time steps would have required a month of sustained computation.

Although PVM was the most convenient introduction to a parallel system, the techniques learned and used with this system are directly transferable to more tightly coupled distributed parallel systems. Modification of the component by component ADI method by the introduction of a sweep factor, $s$, provided a way to conceptually achieve parallelism even under the limited communication speed of a loosely connected, distributed, virtual parallel system.

In retrospect, the nature of the problem suggests that it may have been better suited to a shared memory type system. On such systems, the concept of passing data is not an issue. In this case, the problem associated with the ADI direction alternation would not have existed. However, problems with memory throughput due to cache size limitations could have arisen instead.
Part III

Nonlinear problem
Chapter 5

IG waves in the presence of 2D turbulence

To investigate further the role of IG waves in the atmosphere, a nonlinear model of fluid motion was sought. A system described by Farge in [15] has been solved previously [13] using pseudo-spectral techniques on a Cray supercomputer. Her work describes the effect of IG waves and rotation on a decaying 2D flow.

The numerical aim of this research was to solve the system using finite difference methods and local compute power. Physically, the aim was to establish the effect of both forced and unforced inverse cascades of IG waves in the presence of rotational turbulence.

5.1 Physical description

The model derived in §1.6.1 consists of the Barré de Saint-Venant equations written in terms of geopotential, \( \phi \), vorticity, \( \omega \) and divergence, \( \delta \):

\[
\frac{\partial \phi}{\partial t} + \nabla \cdot (\omega + f) \mathbf{v} = 0, \quad (5.1)
\]

\[
\frac{\partial \delta}{\partial t} - \nabla \times (\omega + f) \mathbf{v} + \nabla^2 \left( \phi + \frac{v^2}{2} \right) = 0, \quad (5.2)
\]

\[
\frac{\partial \phi}{\partial t} + \phi \delta + \mathbf{v} \cdot \nabla \phi = 0 \quad (5.3)
\]
where $\vec{v}$ represents the velocity vector and $f$, the Coriolis parameter. This model is supplemented by the nonlinear invariants, energy ($\mathcal{E}$) and potential enstrophy ($N$),

$$\mathcal{E} = \frac{1}{2} \iint (\phi^2 + \phi u^2) \, dx \, dy,$$

$$N = \frac{1}{2} \iint \left( \frac{\omega + f}{\phi} \right)^2 \, dx \, dy.$$

It is capable of describing incompressible fluid motion in a rotating stratified shallow water layer. In the simulations performed below, velocity magnitude was kept below 10% of $c_s (\sqrt{\phi})$ to ensure incompressibility and the variation in $\phi$ was kept below 5% of $\bar{\phi}$ to justify the assumption of shallowness.

5.2 Numerical description

5.2.1 The velocity vector field

A velocity vector field, $\vec{v}$, is used by the model to evolve the geopotential, vorticity and divergence fields, although $\vec{v}$ itself is not evolved directly. Consequently, to produce a self-sustaining numerical simulation, it must be restored from the numerically evolved fields. Vorticity and divergence can be described in terms of velocity:

$$\omega = \nabla \times \vec{v},$$

$$\delta = \nabla \cdot \vec{v}.$$

A suitable vector identity

$$\nabla \times (\nabla \times \vec{v}) \equiv \nabla (\nabla \cdot \vec{v}) - \nabla^2 \vec{v}$$

is then combined with (5.6) and (5.7) to create a non-evolutionary Poisson type restoration equation for $\vec{v}$:

$$\nabla^2 \vec{v} = \nabla \delta - \nabla \times \omega.$$

In scalar form, (5.8) yields a pair of elliptic PDEs:

$$\nabla^2 u = \frac{\partial \omega}{\partial y} + \frac{\partial \delta}{\partial x},$$

$$\nabla^2 v = \frac{\partial \delta}{\partial y} - \frac{\partial \omega}{\partial x}.$$
CHAPTER 5. IG WAVES IN THE PRESENCE OF 2D TURBULENCE

The model

In scalar form, (5.1), (5.2) and (5.3) form a set of nonlinear parabolic PDEs:

\[
\frac{\partial \omega}{\partial t} = -\frac{\partial (\omega + f)u}{\partial x} - \frac{\partial (\omega + f)v}{\partial y},
\]

\[
\frac{\partial \delta}{\partial t} = \frac{\partial (\omega + f)v}{\partial x} + \frac{\partial (\omega + f)u}{\partial y} - \nabla^2 \left( \phi + \frac{v^2}{2} \right),
\]

\[
\frac{\partial \phi}{\partial t} = -\phi \delta - \frac{\partial \phi}{\partial x} - \frac{\partial \phi}{\partial y}
\]

where \(v^2 = u^2 + v^2\). To complete the model, time invariants (5.4) and (5.5) linearised around a rest state of \(\delta = \omega = 0\) and \(\phi = \phi^*\) become:

\[
\mathcal{E} = \frac{1}{2} \iint (\phi^2 + \phi \nu^2) \, dx \, dy,
\]

\[
N = \frac{1}{2} \iint \frac{\omega^2}{\phi} \, dx \, dy.
\]

respectively.

5.2.2 Methodology

Very few of the methods developed and implemented for use in the solution of the model in part II could be reused in the numerical evolution of this model.

The problem was posed as a periodic initial value problem, illustrated in fig. 3.2(b). This precluded the application of the Thomas algorithm to resolve tridiagonal matrices. Each solution field wraps at the edges, producing circular implicit dependencies within every 1D strip.

The high degree of nonlinearity in the system favoured the use of explicit spatial integration methods over implicit methods. Explicit finite difference numerical methods always produce linear FDEs, whereas implicit methods can produce nonlinear FDEs containing products of unknown solution points. In our case, this problem is exacerbated due to the coupling of \(\delta\) with both \(\omega\) and \(\delta\). However, the potential instability of a nonlinear system and the conditional stability of explicit methods can produce an undesirable cocktail of stability issues.

To solve the elliptic Poisson PDEs produced by the process of velocity restoration, the slow but simple iterative approach used in part II is not a viable option. Solution of \(u\) and \(v\) is part of the evolution process and is therefore time critical. A fast, direct, non-iterative method is the best solution for such a problem. The use of a periodic
grid enables the option of using a Fourier transform technique whereby the PDE is transformed into a simple function. This function gives the spectral solution at each point in the solution domain. Transforming back yields the spatial solution.

5.2.3 Managing the model

To conceive a solution scheme for this model, it was first necessary to break the whole problem down into separate manageable pieces:

1. Initialise
   (a) Find vorticity and divergence fields from initial velocity vector.
   (b) Create initial geopotential field.

2. Evolve model
   (a) Solve (5.11) to produce a new vorticity field.
   (b) Solve (5.12) to produce a new divergence field.
   (c) Solve (5.13) to produce a new geopotential field.

3. Restore \( \vec{v} \)
   (a) Solve (5.9) to produce \( u \).
   (b) Solve (5.10) to produce \( v \).

4. Iterate
   (a) Use \( u = (u^n + u^{n+1})/2 \) and \( v = (v^n + v^{n+1})/2 \) to recalculate \( \omega, \delta \) and \( \phi \).
   (b) Cease iteration upon suitable level of velocity convergence.
   (c) Propagate model through time by accepting the new solution.

The periodicity of the problem effectively eliminated boundaries and allowed correct representation in the frequency domain. On the negative side, a periodic solution mesh adds additional problems when computing edge data. Special care must be taken to ensure that relative to \( i = I \), point \( i + 1 \) references point \(-I\). At the corners of the mesh, two of the four points at \( i \pm 1, j \pm 1 \) wrap.

On an individual basis, each PDE appears to be linear, however when considered in terms of the system as a whole, each is nonlinear due to the implicit dependence on the velocity vector. To avoid nonlinearity in the individual PDEs, explicit 2D methods were chosen for spatial integration of the evolutionary fields.
Both standard and leapfrog (§3.6) time integration methods were considered and a generic implementation was achieved to incorporate either method into the numerical scheme. The elliptic Poisson PDEs, (5.9) and (5.10) were solved using a periodic FFT technique established previously in §3.4.2 and implemented below in §5.3.2.

5.2.4 Initialisation

Initial data was available in the form of $\phi$, $f = |2\Omega|$ and $\vec{v}(x,y,0)$ as a combination of $u$ and $v$. All functions were doubly periodic in space to be solved on a periodic solution grid of size $(2\pi)^2$.

By an application of simple vector arithmetic, initial values for vorticity and divergence can be derived from velocity:

$$\omega = \nabla \times \vec{v} = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y},$$
$$\delta = \nabla \cdot \vec{v} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}.$$  

Numerically, $\omega$ and $\delta$ can be solved using standard second order accurate finite differences:

$$\omega_{i,j} = \frac{v_{i+1,j} - v_{i-1,j}}{2h} - \frac{u_{i,j+1} - u_{i,j-1}}{2k},$$  
$$\delta_{i,j} = \frac{u_{i+1,j} - u_{i-1,j}}{2h} + \frac{v_{i,j+1} - v_{i,j-1}}{2k}.$$  

where $h$ and $k$ represent the spatial point separations as usual.

5.2.5 Evolution

Time integration

To address the two level startup problem with the leapfrog method and leave the choice of two or three level method open, it is necessary to give time integration special attention. By taking account of the three level leapfrog method at an early stage of implementation, it is possible to provide enough generality to enable the choice to be deferred until compile time. Consider tables 5.1 and 5.2 with reference to

$$\frac{\partial f}{\partial t} = \frac{f_{i,j}^{n+1} - f_{i,j}^{n-1}}{2d}.$$
Chapter 5. IG Waves in the Presence of 2D Turbulence

They illustrate leapfrog and standard time integration methods, respectively. A 3D solution array is used with the third dimension indexing the 2D solution at relative levels of time.

Leapfrog and standard methods only require two levels of storage, however, as the leapfrog method reuses the old time level to store the new time level, it is a one shot operation which cannot be iterated. Table 5.3 illustrates the generic scheme extended further to allow iteration with the leapfrog method. In this case, three levels of storage are required.
CHAPTER 5. IG WAVES IN THE PRESENCE OF 2D TURBULENCE

Vorticity

Using the generic method for time integration and $O(h^2, k^2)$ accurate, 2D FDAs for the space integration, (5.11) is transformed into

$$
\frac{\omega_{i,j}^{n+1} - \omega_{i,j}^{n-1}}{2d} = -\frac{1}{2h} \left[ (\omega + f)u_{i+1,j}^n - (\omega + f)u_{i-1,j}^n \right] - \frac{1}{2k} \left[ (\omega + f)v_{i,j+1}^n - (\omega + f)v_{i,j-1}^n \right].
$$

(5.16)

With respect to $\omega_{i,j}^{n+1}$, (5.16) becomes

$$
\omega_{i,j}^{n+1} = \omega_{i,j}^{n-1} - \frac{d}{h} \left[ (\omega + f)u_{i+1,j}^n - (\omega + f)u_{i-1,j}^n \right] - \frac{d}{k} \left[ (\omega + f)v_{i,j+1}^n - (\omega + f)v_{i,j-1}^n \right].
$$

Divergence

Discretised in the same way as vorticity, (5.12) is transformed into

$$
\frac{\delta_{i,j}^{n+1} - \delta_{i,j}^{n-1}}{2d} = -\frac{1}{2h} \left[ (\omega + f)v_{i+1,j}^n - (\omega + f)v_{i-1,j}^n \right] - \frac{1}{2k} \left[ (\omega + f)v_{i,j+1}^n - (\omega + f)v_{i,j-1}^n \right] - \frac{\phi + v^2/2}{h^2} \frac{\phi_{i+1,j}^n - 2\phi_{i,j}^n + \phi_{i-1,j}^n}{h^2} - \frac{\phi + v^2/2}{k^2} \frac{\phi_{i,j+1}^n - 2\phi_{i,j}^n + \phi_{i,j-1}^n}{k^2}.
$$

(5.17)

With respect given to $\delta_{i,j}^{n+1}$, (5.17) becomes:

$$
\delta_{i,j}^{n+1} = \delta_{i,j}^{n-1} + \frac{d}{h} \left[ (\omega + f)v_{i+1,j}^n - (\omega + f)v_{i-1,j}^n \right] - \frac{d}{k} \left[ (\omega + f)v_{i,j+1}^n - (\omega + f)v_{i,j-1}^n \right] - \frac{2d}{h^2} \left[ (\phi + v^2/2)_{i+1,j}^n - 2(\phi + v^2/2)_{i,j}^n + [\phi + v^2/2]_{i-1,j}^n \right] - \frac{2d}{k^2} \left[ (\phi + v^2/2)_{i,j+1}^n - 2(\phi + v^2/2)_{i,j}^n + [\phi + v^2/2]_{i,j-1}^n \right].
$$
CHAPTER 5. IG WAVES IN THE PRESENCE OF 2D TURBULENCE

Geopotential

The product $\phi \delta$ in (5.13) was initially considered special. By evolving the fields in the specified order, the entire $\delta^{n+1}$ solution is available prior to solving for $\phi$. The use of $\phi^{n+1}$ coupled to divergence produces a linear FDE. However $\phi \delta = \phi^{n+1} \delta^{n+1}$ was considered unsuitable due to its dependence on field ordering and unknown behaviour in the context of an otherwise explicit method. A similar Crank-Nicolson approach was also proposed:

$$\phi \delta = \frac{1}{2} (\phi^{n+1} \delta^{n+1}) + \frac{1}{2} (\phi^n \delta^{n+1}).$$  \hspace{1cm} (5.18)

Indeed, a geopotential FDE incorporating (5.18) was used in initial experimentation, however the evolution was highly unstable. Severe reductions in $\tau$ did not improve the instability, thus it could not be attributed to conditional explicit behaviour alone. This was not surprising as the production of a consistent FDE with (5.18) requires that the spatial derivatives are also represented by FDAs at the intermediate time level. Implementation of this would have created an implicit FDE requiring a 2D periodic solution scheme.

To ensure consistence in the model as a whole and to allow arbitrary ordering of field evolution, a standard 2D explicit method was implemented. Equation (5.13) is transformed into

$$\frac{\phi_{i,j}^{n+1} - \phi_{i,j}^{n-1}}{2d} = -\phi_{i,j} \delta_{i,j} - u_{i,j}^n \phi_{i+1,j}^{n+1} + \phi_{i-1,j}^{n+1} - \phi_{i,j+1}^n - \phi_{i,j-1}^n. $$

With respect to $\phi_{i,j}^{n+1}$, this becomes

$$\phi_{i,j}^{n+1} = \phi_{i,j}^{n-1} - d \phi_{i,j} \delta_{i,j} - \frac{d}{h} u_{i,j}^n (\phi_{i+1,j}^n - \phi_{i-1,j}^n) - \frac{d}{k} v_{i,j}^n (\phi_{i,j+1}^n - \phi_{i,j-1}^n).$$

Velocity

Expanding (5.9) and (5.10), using $O(h^2, k^2)$ accurate finite differences to compute the differentials of vorticity and divergence yields

$$\nabla^2 u = f^u,$$  \hspace{1cm} (5.19)

$$\nabla^2 v = f^v$$  \hspace{1cm} (5.20)
where for each discrete point:

\[ f_{i,j}^u = \frac{\omega_{i,j} + 1 - \omega_{i,j} - 1}{2k} + \frac{\delta_{i+1,j} - \delta_{i-1,j}}{2h}, \]  
\[ f_{i,j}^v = \frac{\omega_{i,j} + 1 - \omega_{i,j} - 1}{2k} - \frac{\omega_{i+1,j} - \omega_{i-1,j}}{2h}. \]  

5.3 Implementation

5.3.1 Programming style

As mentioned earlier, the implementation of this problem could not reuse any previously implemented algorithms. Although this was unfortunate, it did provide an opportunity to address some of the problems that became apparent during earlier programming.

In terms of actual coding style, an attempt was made to modularise the algorithms using far more subroutines and functions which were themselves more general. Instead of hand optimizing each equation, thus removing generality and reducing readability, the equations were coded almost directly as written on the page, with function calls implementing standard operators.

**Periodic boundaries**

For instance, to deal with periodic boundary conditions, a single subroutine was used to solve every solution plane. This routine was passed the address of a function holding the actual FDE to be solved at each point in the solution mesh. The routine was responsible for passing the correct values of \( i \pm 1 \) and \( j \pm 1 \) at each point. This kind of modularity greatly reduced the amount of trivial programming errors that were made. It also meant that a bug would manifest itself uniformly allowing easier identification. With hand optimized subroutines a bug can remain very well hidden, due to complicated manifestations.

5.3.2 Restoration of velocity

The first part of the problem to be implemented was the final stage in the evolution of the model.

As velocity is a vector quantity, the 2D model requires two velocity component fields, each representing a component of velocity at each point in the solution domain. Hence, the \( u \) domain contains all of the \( x \) components of velocity at each point and
the \( v \) domain contains all \( y \) components of velocity. Due to stratification, the model does not incorporate vertical motion.

Although there are several solution fields used in the numerical evolution of the model, the velocity field is the physical realisation of the model. The energy in the model is proportional to \( u^2 \).

**FFT solution**

The Poisson equations, (5.19) and (5.20) are solved using a Fourier transform:

\[
\hat{f}(\nabla^2 u) = \hat{f}(f^u),
\]

\[-\kappa_x^2 p = g^p,\]

\[-\kappa_y^2 q = g^q.\]

The functions produced are expanded to obtain the individual point solutions:

\[
p_{i,j} = \frac{g_{i,j}^y}{-(\kappa_x^2 + \kappa_y^2)},
\]

\[
q_{i,j} = \frac{g_{i,j}^x}{-(\kappa_x^2 + \kappa_y^2)}.
\]

where \( \kappa_x = i \) and \( \kappa_y = j \) due to the doubly periodic solution domain.

**Method**

The following list describes the stages required to attain a solution for \( u \):

1. Use (5.21) to produce 2D \( f^u \) array.

2. Perform forward 2D Fourier transform, \( \hat{f}(f^u) = g^p \):
   - 1D FFT in \( z \):
     - For each 1D strip in \( x \), do real forward FFT.
     - Recode the real spectral data into a half sized complex 2D array.
   - 1D FFT in \( y \):
     - For each part transformed 1D strip in \( y \), do complex forward FFT.
   - Postprocess the spectral image.
     - Sort spectral image into array of ascending frequency components.
     - Build the negative \( x \) frequency image using complex conjugate.\(^1\)

\(^1\)used only for visualisation purposes as the negative \( x \) spectrum contains no additional information.
3. Solve (5.23) at each point in the half sized complex domain.

4. Apply a high frequency truncation filter\(^2\)

5. Perform reverse 2D Fourier transform, \(u = \hat{\mathcal{F}}^{-1}(p)\):
   - Pre-process the spectral image.
     - Sort spectral array back into native order.
   - 1D FFT\(^{-1}\) in \(y\):
     - For each 1D strip in \(y\), do complex reverse FFT.
   - 1D FFT\(^{-1}\) in \(x\):
     - Recode the complex spectral data into full sized real 2D array.
     - For each 1D strip in \(x\), do real reverse FFT.

A similar method was used to solve \(v\). The real and complex forward and reverse 1D FFTs were performed using standard FFT library routines.

The entire solver was tested using some sample functions for \(u\) and \(v\) from which \(\omega\) (5.14) and \(\delta\) (5.15) were obtained. These fields were then passed through the solver to regenerate \(u\) and \(v\). Several problems relating to periodic boundaries were fixed and the correct function of the solver verified.

### 5.3.3 Time stepping

As established in §5.2.5, the vorticity, divergence and geopotential PDEs were discretised using explicit methods. In space, standard FDAs were used, in time, a generic method was used to allow testing with both two and three level time stepping techniques.

At first, a two storage level leapfrog method was tested, using a standard first order accurate kick start. Consequently, no iteration was performed. Without reference to early stability problems\(^3\), odd-even decoupling was strongly present. This decoupling is a well known symptom of the leapfrog method and is very much dependent upon the characteristics of the model. Adding iterative procedures for the nonlinearities could have been implemented by increasing the storage requirement of the leapfrog method to three levels. However, due to the strong decoupling behaviour and the lack of a second order or higher startup and re-coupling technique, the leapfrog method was

\(^2\)or anti-alias filter
\(^3\)indicated by high energy and potential enstrophy variance
replaced with the standard $O(\tau)$ accurate, two time level FDA. This method was used with success. Stability problems continued, though further reductions in $\tau$ were able to stabilise the simulation for several units of dimensionless time.

### 5.3.4 Nonlinearities

The solution of a nonlinear problem is never straightforward. There are many ways in which the nonlinearities can evolve. Inevitably, the solution of a nonlinear equation is dependent upon itself, either directly or indirectly. In our model, there are a number of indirect nonlinearities. All three parabolic PDEs are dependent on the velocity in the system. However, the velocity in the system is determined by a numerical combination of vorticity and divergence. To add to this level of interdependence, the vorticity and geopotential PDEs have mutual dependencies. Regardless of the iterative feedback mechanism used \cite{53, 47, 35}, there are several ways in which to consider the nonlinearities:

1. At the individual PDE level.
2. On mutually dependent sets of PDEs.
3. On the system of PDEs as a whole.

To deal with the nonlinearities individually poses an almost insurmountable problem. Consider solving for vorticity and using this new field to recompute the velocity. This process could be iterated until appropriate convergence was obtained. Following this the other PDEs would require similar treatment. As the solution of each individual PDE affects the velocity field, the entire system of individual iterations would have to loop until convergence.

A similar problem occurs when considering mutual dependencies. If velocity is rewritten as a function of vorticity and divergence: \( \bar{v} = g(\omega, \delta) \), it can be seen that every PDE is directly dependent on itself and every other PDE, except for velocity which has no direct dependence\(^{4}\) on geopotential.

This leaves the later approach, which was the favoured implementation. Each PDE is solved explicitly with implicit iteration applied to the entire model. A simple iterative strategy was used whereby the new value of \( \bar{v} \) was the average value of the two most recently computed. Iteration was performed until an arbitrary level convergence

\(^{4}\)Indirect dependence via divergence
was reached:

$$\max \left( \sum \sum |u_{i,j}^{m} - u_{i,j}^{m+1}|, \sum \sum |v_{i,j}^{m} - v_{i,j}^{m+1}| \right) < 1.0,$$

$$\max(A, B) = \begin{cases} A & \text{if } A \geq B, \\ B & \text{if } A < B. \end{cases}$$

5.4 Results

Results were collected in a similar manner to that established in part II. The individual solution fields of vorticity ($\omega$), divergence ($\delta$), geopotential ($\phi$), velocity ($u, v$) and energy ($v^2$) were dumped to disk at equally spaced intervals through time. In the event of abnormal termination, full state data was saved to enable restarting from the termination point. The dumped solution fields were converted to ASCII suitable for gnuplot which was used to visualise the fields and generate hardcopy.

Two preliminary sets of results are included below. They should be considered simply as tests of the numerical model. Significance of the results, if any, has yet to be established.

5.4.1 Small scale geopotential perturbation

A geopotential perturbation was introduced,

$$\phi^0 = 20 + \frac{1}{10} \cos(10x + 10y)e^{-2(x^2+y^2)},$$

to an otherwise unforced system and the resultant wave behaviour observed. A time step of $\tau = 10^{-4}$ was used on a $256^2$ point grid,

$$-\pi \leq x < \pi,$$
$$-\pi \leq y < \pi.$$

Figures 5.2 through 5.4 represent the evolution of the resultant wave packet in the system with figures 5.5 through 5.8 illustrating the geopotential field. Normalised energy and potential enstrophy levels throughout the simulation are shown in fig. 5.1.
CHAPTER 5. IG WAVES IN THE PRESENCE OF 2D TURBULENCE

Figure 5.1: Physical invariants, energy ($E$) and enstrophy ($N$)

![Figure 5.1: Physical invariants, energy ($E$) and enstrophy ($N$)](chart)

Figure 5.2: The energy ($v^2$) of the wave packet at $t = 0.4$
CHAPTER 5. IG WAVES IN THE PRESENCE OF 2D TURBULENCE

Figure 5.3: The energy ($v^2$) of the wave packet at $t = 0.8$

Figure 5.4: The energy ($v^2$) of the wave packet at $t = 1.2$
CHAPTER 5. IG WAVES IN THE PRESENCE OF 2D TURBULENCE

Figure 5.5: Geopotential field ($\phi$) at $t = 0.0$

Figure 5.6: Geopotential field ($\phi$) at $t = 0.4$
Figure 5.7: Geopotential field ($\phi$) at $t = 0.8$

Figure 5.8: Geopotential field ($\phi$) at $t = 1.2$
The resultant wave packet splits into two equal packets that move in opposite directions. Geopotential levels are affected in a similar manner. By the time the periodic packets interact with one another, the geopotential field has completely broken down. This breakdown causes the simulation to eventually lose stability. It is possible that the nonlinear interaction of the two wave packets contributes to the instability. A similar experiment was performed with the initial condition offset by $\pi/2$ in both directions. This was done to determine if the instability was caused by incorrect handling of solution domain boundaries. However, identical behaviour was seen thus ruling out such problems.

5.4.2 Influence of a small scale variability on a large scale pattern

The interaction of a small scale geopotential perturbation with a large scale IG wave packet is modelled. Initial conditions were defined as

\begin{align*}
  u^0 &= \frac{1}{10} \sin(x + 2y)e^{-x^2 - y^2}, \\
  v^0 &= \frac{1}{5} \sin(3x + 4y)e^{-x^2 - y^2}, \\
  \phi^0 &= 20 + \frac{1}{10} \cos \left( \frac{x^2 + y^2}{3} \right).
\end{align*}

A time step of $\tau = 10^{-4}$ was used on the same $256^2$ point grid. Figures 5.10 through 5.13 illustrate the evolution of the wave packet and figures 5.14 through 5.17 illustrate the geopotential field. Energy and enstrophy variance is shown in fig. 5.9.

5.5 Validating the results

The sources of error in a numerical simulation were outlined in §4.5. Removal of the highest frequency components was implemented using FFT transforms. This was also discussed in the aforementioned section. Methods of validation used here were based upon invariance of the physically conserved properties: energy and potential enstrophy. Any variance exceeding 10% during the lifetime of a simulation would cause premature termination.
CHAPTER 5. IG WAVES IN THE PRESENCE OF 2D TURBULENCE

Figure 5.9: Physical invariants, energy ($\mathcal{E}$) and enstrophy ($N$)

Figure 5.10: The energy ($v^2$) of the wave packet at $t = 0.0$
CHAPTER 5. IG WAVES IN THE PRESENCE OF 2D TURBULENCE

Figure 5.11: The energy \( (v^2) \) of the wave packet at \( t = 0.4 \)

Figure 5.12: The energy \( (v^2) \) of the wave packet at \( t = 0.8 \)
CHAPTER 5. IG WAVES IN THE PRESENCE OF 2D TURBULENCE

Figure 5.13: The energy ($v^2$) of the wave packet at $t = 1.2$

Figure 5.14: Geopotential field ($\phi$) at $t = 0.0$
Figure 5.15: Geopotential field ($\phi$) at $t = 0.4$

Figure 5.16: Geopotential field ($\phi$) at $t = 0.8$
5.5.1 Conserved properties

Both $\mathcal{E}$ and $N$ were tested for conservation after each full time step. The trapezoid rule was used for numerical quadrature. In 2D, this resembles

$$Q = \frac{k}{2} \left[ Q_x(-J) + Q_x(J) + 2 \sum_{j=1-J}^{J-2} Q_x(j) \right],$$

$$Q_x(j) = \frac{h}{2} \left[ Q_{xy}(-I,j) + Q_{xy}(I,j) + 2 \sum_{i=1-I}^{I-2} Q_{xy}(i,j) \right].$$

For energy, $Q_{xy} = \mathcal{E}_{xy}$ and for enstrophy, $Q_{xy} = N_{xy}$ where

$$\mathcal{E}_{xy}(i,j) = \phi_{i,j}^2 + \bar{\phi}(u_{i,j}^2 + v_{i,j}^2),$$

$$N_{xy}(i,j) = \frac{\omega_{i,j}^2}{\phi_{i,j}}.$$  

\footnote{as Simpson's rule is applicable only to grids with an even number of intervals}
As shown in §5.4 above, these properties were conserved to a high degree during simulation for several units of dimensionless time.

### 5.6 Conclusions

#### 5.6.1 Physical behaviour

The results presented above are somewhat cursory in nature due to the constraints of research time. Evolution of the waves is consistent with expected behaviour and physical invariants are conserved whilst stability is held.

Had further research time been available, results could have been obtained from more meaningful simulations using smaller time steps and/or larger solution grids.

#### 5.6.2 Numerical methodology

The Barré de Saint-Venant equations produce a difficult model to solve numerically. There are various finite difference methods available to achieve this, however they all have relative drawbacks. The pseudo-spectral technique that Farge implemented used a small $128^2$ point grid and demanded the power of a Cray supercomputer. Perhaps this is a true indication of the complexity of the numerical problem posed.

The elliptic equations produced by the restoration of velocity were solved successfully using FFT spectral techniques. Simple explicit schemes were used to solve the evolutionary equations, however the non-deterministic conditional stability offered by such schemes was highly undesirable. At times, this led to difficulty in determining the cause of unstable behaviour. The use of explicit methods added an extra source of potential error which would not have been present using unconditionally stable methods.

An interesting approach would have been to use second order accurate, unconditionally stable ADI methods, as with the linear problem posed in part II. However, the periodicity of the problem would have required an efficient cyclic reducer to resolve the sets of simultaneous equations produced. Estimating the complexity of such a problem using ADI methods is made difficult due to the dynamic levels of iteration used. Approximately $10(3t(\text{ADI}) + t(\text{FFT}))$ more wall-clock time would have been necessary to compute a single numerical time step for the nonlinear model posed here, as compared with the linear model posed in the part II. With $t(\text{ADI})$ representing slightly over one and $t(\text{FFT})$, a yet unmeasured quantity. The final factor would likely be near the order of 100. Hence, the use of ADI methods, albeit desirable would not have been practical
using local compute power.

5.6.3 Parallel implementation

Expanding explicit 2D solution methods into the realms of parallel computation was discussed in §4.6.1. However, implementation using a loosely coupled system would not be trivial. The restoration of velocity produces an axial problem similar to that found with ADI methods. The FFT is split into dimensional components with a transformation in $y$ following directly after a transformation in $x$. This would make a simple farm technique, favoured by explicit methods, highly inappropriate to solve the problem as a whole.

Using either explicit or ADI methods to solve this model would pose a serious problem with respect to parallelisation. Further investigation in this area would be required to shed light on such an implementation.
Chapter 6

General conclusions

The broad nature of this work allowed several goals to be identified:

1. Enhance scientific knowledge in the area of atmospheric wave interactions.
2. Achieve numerical solutions of atmospheric models.
3. Investigate parallelisation in terms of numerical solution methods.

The aim, as with all research, was to achieve some degree of originality in these areas.

6.1 Introductory work

The first part of this thesis introduced some of the base concepts and theory required to achieve the numerical solution of atmospheric models. In terms of science, at least a general understanding of the physical laws governing wave behaviour was required.

Programming experience and a familiarity with computer systems was necessary. This applies to the implementation of algorithms and to the visualisation of results. Parallel techniques were investigated as a potential optimization method. Although time constraints hindered the full exploitation of such methods, their investigation was justified. Even using the simplest explicit methods to solve the nonlinear problem took close to a week to complete on sequential machines. If time had allowed, a parallel solution technique for this problem would have posed a very interesting challenge.

Finite difference methods were used as the basis for numerical integration. As the authors previous experience of such methods was non-existent, a considerable amount of methodology was investigated and presented with example implementations at various stages.
6.2 Numerical methodology

Fig. 6.1 illustrates where the various numerical methods were used in relation to each other and to the problems that were solved. Keywords appearing in boxes denote the numerical solution of keyword. The meaning of “Standard” refers to the $O(r)$ accurate, two level time integration method.

6.3 Linear equation

The second part of this thesis used Falkovich’s linear approximation of the shallow water equations as a basis for research. Novelty was immediately apparent due to the lack of any previous numerical solutions. The 2D model proved to be a non-trivial problem in terms of compute power. Relatively painless unconditional stability was achieved using an ADI solution method.

Results produced by the simulations warranted publication. However, they failed to provide evidence of behaviour suggesting atmospheric blocking. As a consequence, further investigation using this model or a nonlinear variant was not attempted.

To enable the use of larger solution grids, speed optimization in the form of parallel techniques were investigated. A loosely coupled virtual parallel environment was chosen as a convenient test-bed. The nature of the ADI solution method suggested that modifications of the method were necessary. The novel technique chosen was shown to work successfully in place of the unmodified technique. The actual implementation using PVM failed to work properly, however it was suggested that the problem was of a practical nature rather than theoretical.

6.4 Nonlinear system

The final part of this thesis presents a solution of the full nonlinear Barré de Saint-Venant model.

Farge’s solution of the problem used pseudo-spectral methods and required the application of an Asselin filter at each time step to damp the fastest IG waves. The solution grid contained $128^2$ points.

A novel solution method using a combination of finite difference and FFT spectral techniques is presented here. Although a solution grid containing four times as many points was used, simulation remained possible using local compute power. The cursory nature of the results produced was simply evidence of the finite amount of research
Figure 6.1: Methodology chart
time available. Since the preparation of the results in part III, further simulations using a smaller time step have yielded stable results at evolutionary periods in time exceeding those presented here.

6.5 Achievements

As discussed here and in previous chapters, all three of the goals identified above were met with varying degrees of success. Scientific knowledge was increased by the publication of results from part II.

Implementation, and in some cases modification, of numerical solution methods resulted in the solution of a 2D linear equation and a 2D nonlinear system, both describing atmospheric motions with varying degrees of approximation.

Parallel methods were thoroughly investigated and a technique was proposed to achieve speedup of an ADI method in a loosely coupled distributed virtual parallel environment. The application of parallelisation to the nonlinear solution scheme was briefly discussed.
Bibliography


BIBLIOGRAPHY


[48] UNIRAS A/S. *AGL series user and reference manuals*.

[49] UNIRAS A/S. *FGL/GRAPHICS user and reference manuals*.


Appendix A

Program code

A.1 Code implementing problem posed in Part II

A.1.1 2D ADI solver

Header (progs/2dparams37.h)

```c
/* Grid parameter values */
#define xp_val 127             /* +/- No. of x-dir_n grid points */
#define yp_val xp_val          /* +/- No. of y-dir_n grid points */
#define h_val 0.05             /* Value of dx, space spacing */
#define k_val h_val             /* Value of dy, space spacing */
#define t_val 0.05             /* Value of tau, time spacing */
#define tp_val 50             /* No. of time levels */

/* Equation parameter values */
#define a_val 0.5             /* Value of eq_n constant a */
#define initial_psi_val cmplx(exp((-x**2-y**2)/6.), 0.0)
#define eta_val a*exp(-x**2-y**2)

/* Disk I/O parameter values */
#define in_file '/dev/null'   /* File prefix of initialization data */
#define out_file 'n37a_2dren3c' /* File prefix of output files */

/* Misc parameter values */
#define no_of_graphs 5        /* No. of data outputs per run */
#define bailout_on
#define antisymmetric
```

Driver (progs/n37a-2d.F)

C Author: Wilf (eepg9w)
C creation date: 26-5-93 (5/26/93)
APPENDIX A. PROGRAM CODE

C last revision date: %G%
C revision from: newS3atmos.df & n37a_1d.F
C Notes: Complete re-hash of our fave 2D program!
C objects: filadmnS simpson2 tridiagS abort bailout2
C finish sig1 sig2 sigl5 intprg mailme
C libraries: maths.e etc.s a signals.s

#include "2dparams37.h"

C Using an alternating direction-implicit scheme
C Component by component splitting up method
C psi is a 2D array which at any time i holds the solution

program Latmos

integer xp, yp, tp, n, iv, step, sig1, sig2, sigl5, i, j, signal, p
real t, h, k, a, simpE, energy, maxnrg, minnrg, ninvar
integer*2 one, two
integer m, range

parameter(tp=tp_val, xp=xp_val, yp=yp_val)
parameter(t=t_val, h=h_val, k=k_val, a=a_val)
parameter(iv=tp/no_of_graphs)
parameter(range=2)

character*50 epofl, psiofl, kilfile
logical int
external sig1, sig2, sigl5

common /psi/ psi
common /kill/ oldpsi, n, /int/ int

data int .false./
data one, two /1, 2/

print*, 'Setting up internal constants...

epoff=out_file//'FileEP.data'
psiofl='B'//out_file//'FileAllPsi.data'
kilfile='B'//out_file//'FAP_DEAD.data'

C Set appropriate IEEE flag(s);
* call abrupt_underflow()

C Open up the output files (args: 0 for ascii, 1 for binary):
call filadmn3(epoff, psiofl, 1)
APPENDIX A. PROGRAM CODE

C initialise the output files;

    write(11)'!', xp, yp, t, h, k, a
    write(10, tp, -xp, -yp, yp, t, h, k

10 format(' Time points: ', i6, /,
       + ' X range: ', i4, ' -> ', i4, /,
       + ' Y range: ', i4, ' -> ', i4, /,
       + ' Parameters (t, h, k): ', 3(e7.2, x))

C Sort out some signal handling;

    open(8, file=kilfile, status='unknown', form='unformatted')
    write(8)'!', xp, yp, t, h, k, a
    i=signal(1, sig1, -1)
    i=signal(2, sig2, -1)
    i=signal(15, sig15, -1)

C find initial condition;

    print*, 'Initialised signal handling, Calculating time constants . . . '
    call tcsetup()
    print*, 'Finding initial condition . . . '
    call init2D(step)
    call writfil(step)
    call simpson2(one, h, k, xp, yp, energy)

= call fftdriver()
* call simpson2(two, h, k, xp, yp, ninvar)
*
    print*, 'NINVAR: ', ninvar
    maxnrg=energy*1.1
    minnrg=energy*0.9
    print*, 'Calculating Psi . . . '

C forall time;

    p=iv
    do n=step+1, tp+step, 2

C calc time steps 1, 2;

    do m = 1, range/2
       call xarray()
    enddo

    do m = 1, range
       call yarray()
    enddo

    do m = 1, range/2
APPENDIX A. PROGRAM CODE

```
call xarray()
enddo

C Save value of psi in 'kill' common block

do i=-xp, xp
  do j=-yp, yp
    oldpsi(i, j)=psi(i, j)
  enddo
enddo

C check const. E;

call simpson2(one, h, k, xp, yp, simpE)
* call fftdriver()
* call simpson2(two, h, k, xp, yp, minvar)
  ninvar=0.

#ifdef bailout_on
  if (simpE.gt.maxnrg.or.simpE.lt.minnrg)
    call bailout2(n, t, energy, simpE)
#endif

print*, 'Step: ', n, ' const: ', simpE,
  + 'N_invar: ', ninvar, ' ', cabs(psi(0, 0))**2
write(10, *)n, simpE, ninvar, cabs(psi(0, 0))**2

C store main array (if reqd) and finish if interrupted;

if (int) then
  call writfil(n)
call finish()
endif

p=p-1
if (p.eq.0) then
  call writfil(n)
p=iv
endif

enddo

call writfil(n)
call finish()

C return to OS;

end

C   
```
APPENDIX A. PROGRAM CODE

subroutine xarray()

C

integer xp, yp, i, j
real b
parameter(xp=xp_val, yp=yp_val, b=2.0/t_val)
complex ml(-xp+1:xp), md(-xp:xp), mu(-xp:xp-1)
complex mb(-xp:xp), mbd(-xp:xp), mbu(-xp:xp-1)
complex psi(-xp:xp, -yp:yp)

common /psi/, /xtc/ cx, dx, ex

C forall y:

do j = -yp, yp

C calc matrix vars mu, md, ml @ interior points

do i = -(xp-1), xp-1

mbu(i) = complex(dx(i, j)-ex(i, j), -cx(i, j))
mbd(i) = complex(2.0*ex(i, j), b)
mb(i) = complex(dx(i, j)+ex(i, j), cx(i, j))
md(i) = complex(-2.0*ex(i, j), b)
mb(i) = psi(i+1, j)*mbu(i)+psi(i, j)*mbd(i)-psi(i-1, j)*mb(i)
endo

C calc matrix vars mu, md, ml and mb @ boundaries

mu(-xp) = complex(dx(-xp, j)+ex(-xp, j), cx(-xp, j))
md(-xp) = complex(-2.0*ex(-xp, j), b)
mbu(-xp) = complex(dx(-xp, j)-ex(-xp, j), -cx(-xp, j))
mbd(-xp) = complex(2.0*ex(-xp, j), b)
mb(-xp) = psi(-xp, j)*mb(-xp)+psi(-xp+1, j)*mbu(-xp)
ml(xp) = complex(dx(xp, j)+ex(xp, j), cx(xp, j))
md(xp) = complex(-2.0*ex(xp, j), b)
mb(xp) = complex(dx(xp, j)-ex(xp, j), -cx(xp, j))
mbd(xp) = complex(2.0*ex(xp, j), b)
mb(xp) = psi(xp, j)*mb(xp)+psi(xp-1, j)*mbd(xp)
endo

C calc next time level;

call tridiag3(-xp, xp, ml, mb, md, mu, future)

C store in main array;

do i = -xp, xp

psi(i, j) = future(i)
endo
enddo

end

C

subroutine yarray()

C integer xp, yp, i, j
real b
parameter(xp=xp_val, yp=yp_val, b=2.0/t_val)
complex ml(-yp:yp), md(-yp:yp), mu(-yp:yp-1)
complex mb(-yp:yp), mbd(-yp:yp), mbu(-yp:yp-1)
complex ps(-xp:xp, -yp:yp)
common /psi/ psi, /yc/ cy, dy, ey

C forall x;
do i=-xp, xp

C calc matrix vars mu, md, ml @ interior points

do j=-(yp-1), yp-1
  mbu(j)=complex(-dy(i, j)-ey(i, j), -cy(i, j))
  mbd(j)=complex(2.0*ey(i, j), b)
  ml(j)=complex(dy(i, j)-cy(i, j), cy(i, j))
  mu(j)=complex(dy(i, j)+ey(i, j), cy(i, j))
  md(j)=complex(-2.0*ey(i, j), b)
  ml(j)=complex(-dy(i, j)+ey(i, j), -cy(i, j))
  mb(j)=psi(i, j+1)*mbu(j)+psi(i, j)*mbd(j)+psi(i, j-1)*ml(j)
enddo

C calc matrix vars mu, md, ml and mb @ boundaries

mu(-yp)=complex(dy(i, -yp)+ey(i, -yp), cy(i, -yp))
md(-yp)=complex(-2.0*ey(i, -yp), b)
mbu(-yp)=complex(-dy(i, -yp)-ey(i, -yp), -cy(i, -yp))
mbd(-yp)=complex(2.0*ey(i, -yp), b)
mb(-yp)=psi(i, -yp)*mbd(-yp)+psi(i, -yp+1)*mbu(-yp)

ml(yp)=complex(dy(i, yp)+ey(i, yp), cy(i, yp))
md(yp)=complex(-2.0*ey(i, yp), b)
mb(yp)=complex(-dy(i, yp)-ey(i, yp), -cy(i, yp))
mbd(yp)=complex(2.0*ey(i, yp), b)
mb(yp)=psi(i, yp)*mbd(-yp)+psi(i, yp-1)*ml(yp)

C calc next time level;
call tridiag3(-yp, yp, ml, mb, md, mu, future)

C store in main array;
APPENDIX A. PROGRAM CODE

do j=-yp, yp
   psi(i, j)=future(j)
endoj
endoj
end

C
---

subroutine init2D(step)
---

integer xp, yp, i, step, xpp, ypp
parameter(xp=xp_val, yp=yp_val)
character dum*5, pling*1
complex psi(-xp:xp, -yp:yp)
real t, h, k, a
logical found
character infile*40
common /psi/ psi

infile='B'/in_file//'FileAllPsi.data'

C Is there an initial condition?;
inquire(file=infile, exist=found)

C If not, make one;
   if (.not.found) then
      print*, 'No initial condition found - producing own at t=0'
      call idatemn2()
      step=0
      return
   endif

C If so, then read it in!
   open(unit=12, file=infile,
        +form='unformatted', status='old')
   read(12)pling, xpp, ypp, t, h, k, a
   do i=1, 50
      read(12, end=10, err=20)dum, step
      read(12, end=20, err=20)psi
      print*, dum, step
   end do
   10 print*, 'End of file encountered...'
   print*, 'Initial Condition retrieved...'
close(unit=12)
return
APPENDIX A. PROGRAM CODE

20 close(unit=12)
call abort()
end

C subroutine idatmem2()

integer xp, yp, i, j
real a, h, k, x, y
parameter(xp=xp_val, yp=yp_val, a=a_val, h=h_val, k=k_val)
complex psi(-xp:yp, -yp:yp)
common /psi/ psi

print*, 'Computing Init. Cond...' 
do i=-xp, xp
 do j=-yp, yp
 x=real(i)+h
 y=real(j)+k
 psi(i, j)=initial_psi_val
 enddo
enddo
end

C subroutine writfil(n)

integer xp, yp, n
parameter(xp=xp_val, yp=yp_val)
complex psi(-xp:yp, -yp:yp)
common /psi/ psi

print*, 'About to write time level ', n, ' to disk.'
write(11)'Step=', n
write(11)'psi'
call flush(11)
print*, 'done.'
end

C subroutine tcsetup()

integer xp, yp, i, j, tp
real h, k, a, mem, t, divish, divisk, x, y
parameter(xp=xp_val, yp=yp_val, a=a_val, h=h_val, k=k_val)
parameter(t=t_val, tp=tp_val, divish=2.0*h_val+2, divisk=2.0*k_val+2)
real eta(-xp:yp, -yp:yp), cx(-xp:yp, -yp:yp)
real dx(-xp:yp, -yp:yp), dy(-xp:yp, -yp:yp)
real ex(-xp:xp, -yp:yp), ey(-xp:xp, -yp:yp)

comm on /e ta/ eta, /x tc/ cx, dx, ex, /y tc/ cy, dy, ey

do i=-xp, xp
   do j=-yp, yp
      x=real(i)*h
      y=real(j)*k
      eta(i, j)=eta_val
      dx(i, j)= -real(i)*eta(i, j)
      dy(i, j)= -real(j)*eta(i, j)
      ex(i, j)=(1.0+eta(i, j))/divish
      ey(i, j)=(1.0+eta(i, j))/divisk
      #ifdef antisymmetric
         cx(i, j)= -dy(i, j)*k/h
         cy(i, j)=dx(i, j)*h/k
      #else
         cx(i, j)=0.0
         cy(i, j)=0.0
      #endif
   enddo
endo

C display check values;

C mem = sarray_size * sarray_size * no_of_bytes_for_real * no_of_arrays
mem=(2.0*xp+1)*(2.0*yp+1)*4*9/1024
print*, ' Estimated memory usage for array structures: ', mem, 'K'
end

C real function integral(choice, i, j)

integer xp, yp, i, j
integer*2 choice
parameter(xp=xp_val, yp=yp_val)
real eta(-xp:xp, -yp:yp), fft(-xp:xp, -yp:yp)
complex psi(-xp:xp, -yp:yp)
comm on /psi/ psi, /eta/ eta, /fft/ fft

C choice is not used at present - may be used when second
C stability condition is implemented

goto (1, 2), choice

1 integral=(1.0+eta(i, j))*cabs(psi(i, j))**2
   return
2 integral=fft(i, j)
   return
end
APPENDIX A. PROGRAM CODE

C

subroutine killprg()

integer xp, yp, n
parameter(xp=xp_val, yp=yp_val)
complex oldpsi(-xp:xp, -yp:yp)
common /kill/ oldpsi, n

C Dump Psi in aptly titled file;

print*, 'Received signal TERM(15)'
print*, 'Terminating immediately,'
write(8)'Step=', n
write(8)oldpsi
close(8)

print*, 'DEAD.data file written to disk. (Phew!)

C Tell wilf that I have been killed;

open(9, file='message', form='formatted')
write(9, 'Terminated due to signal TERM(15)'
write(9, 'Step', n, ' was reached'
call mailme('Has been killed', 'message')
close(9, status='delete')

C Shutdown gracefully;

close(11)
close(10)
stop

end

C

A.1.2 1D Crank-Nicolson solver

Header (progs/n37a-1d.h)

/ * Grid parameter values */
#define xp_val 1024 /* +/- No. of x-dir. grid points */
#define yp_val 0 /* +/- No. of y-dir. grid points */
#define h_val 0.12 /* Value of dx, space spacing */
#define k_val h_val /* value of dy, space spacing */
APPENDIX A. PROGRAM CODE

```fortran
#define t_val .5e-2  /* Value of tau, time spacing */
#define tp_val 2000  /* No. of time levels */

/* Equation parameter values */
#define initial_psi_val cmplx(0.5*exp(-x**2/6.0), 0.0)
#define initial_eta_val 0.0 /* (-0.5*exp(-(x-2.0)**2) */
#define BETA_VAL (1.0/8.0)

/* Disk I/O parameter values */
#define in_file '/dev/null'   /* File prefix of initialization data */
#define out_file 'n37a_tldrun11c' /* File prefix of output files */

/* Misc parameter values */
#define no_of_graphs 20 /* outputs per run */

Driver (progs/n37a-tld.F)

C Author: Wulf (eep2gw)
C creation date: 4-6-93 (6/4/93)
C last revision date: %G%
C revision from: new35atmos.df > n37a_2d.F
C Notes: Complete re-hash of our face 2D program!
C objects: filadmn2 simpson2 tridiag3 abort bailout
C libraries: maths.a etc.a

#include "n37a_tld.h"

C Using an alternating direction-implicit scheme
C Component by component splitting up method
C psi is a 2D array which at any time t holds the solution

C -------
program tld_atm
C -------

integer xp, yp, tp, n, iv, graphs, step
real t, h, k, simpE, energy, maxnrg, minnrg

parameter(tp=tp_val, xp=xp_val, yp=yp_val)
parameter(t=t_val, h=h_val, k=k_val)
complex psi(-xp:xp, -yp:yp)
integer*2 one

common /psi/ psi
data one /1/

print*, 'Setting up variables...
graphs=no_of_graphs
```
iv=tp/graphs

C Set appropriate IEEE flag(s);

#define SUN
    call abrupt_underflow()
#endif

C Open up the output files (args: 0 for ascii, 1 for binary);

call fileadm4(out_file//'FileEP.data', 10,
+ 'B//'out_file//'FileAllPsi.data', 11, 1)
call fileadm4('/dev/null', 15, 'B//'out_file//'eta.data', 13, 1)
write(11)'!', xp, yp, t, h, k, BETA_VAL
write(13)'E', xp, yp, t, h, k, BETA_VAL

C find initial condition;

print*, 'Calculating time constants. . .
call initialise(step)
call writfil(step)
call simpson2_1D(one, h, xp, energy)
maxnrg=energy*1.1
minnrg=energy*0.9

print*, 'Calculating time levels. . .

C forall time;

do n=step+1, tp+step, 1
C calc time step;

call xarray()
call feedback()

C check const. E;

call simpson2_1D(one, h, xp, simpE)
if (simpE.gt.maxnrg.or.simpE.lt.minnrg)
+  call bailout2(n, t, energy, simpE)
    print*, 'Step: ', n, ' const: ', simpE, ' ', cabs(psi(0, 0))**2
    write(10, *)n, simpE, cabs(psi(0, 0))**2

C store main array (if reqd);

if (mod(n+1, iv).eq.0)call writfil(n+1)

enddo

close(unit=11)
close(unit=10)
print*, 'Done.'  90
C return to OS;
end

C subroutine xarray()

integer xp, yp, i, j
real h, k
parameter(xp=xp_val, yp=yp_val, h=h_val, k=k_val)
real b, a, beta, c(-xp:xp)
parameter(beta=BETA_VAL, a=beta/(2.0*h), b=1.0/t_val)
complex mb(-(xp-1):xp), mbd(-xp:xp), mbu(-xp:xp-1)
complex psi(-xp:xp, -yp:yp)
real eta(-xp:xp, -yp:yp)

C initialise c(i);
do i=-xp, xp
    c(i)=(eta(i, 0)+1.0)/(2.0*h**2)
enddo

C forall y;
do j=-yp, yp
    C calc matrix vars mu, md, ml @ interior points
    do i=-(xp-1), xp-1
        mu(i)=complex(a, -c(i))
        md(i)=complex(b, 2.0*c(i))
        ml(i)=complex(-a, -c(i))
        mbu(i)=complex(-a, c(i))
        mbd(i)=complex(b + a*(eta(i+1, j)-eta(i-1, j)), -2.0*c(i) + (eta(i+1, j)-2.0*eta(i, j)+eta(i-1, j))/(4.0*h))
    enddo

    C calc matrix vars mu, md and mb @ boundaries
    mu(-xp)=complex(a, -c(-xp))
    md(-xp)=complex(b, 2.0*c(-xp))
    mbd(-xp)=complex(-a, c(-xp))
end subroutine xarray()
APPENDIX A. PROGRAM CODE

\[ mbd(-xp) = \text{complx}(b + a * (\eta(-xp+1, j) - 0), \]
\[ + -2.0 * e(-xp) - \]
\[ + (\eta(-xp+1, j) - 2.0 * \eta(-xp, j) + 0) / (4.0 * h) \]
\[ mb(-xp) = psi(-xp, j) * mbd(-xp) + psi(-xp+1, j) * mbu(-xp) \]

\[ ml(xp) = \text{complx}(-a, -c(xp)) \]
\[ md(xp) = \text{complx}(b, 2.0 * c(xp)) \]
\[ mb(xp) = \text{complx}(a, c(xp)) \]
\[ mbd(xp) = \text{complx}(b + a * (0 - \eta(xp-1, j)), \]
\[ + -2.0 * e(xp) - \]
\[ + (0 - 2.0 * \eta(xp, j) + \eta(xp-1, j)) / (4.0 * h) \]
\[ mb(xp) = psi(xp, j) * mbd(xp) + psi(xp-1, j) * mbl(xp) \]

\text{calc next time level;}
\text{call tridiag3(-xp, xp, ml, mb, md, mu, future)}

\text{store in main array;}
\text{do i=-xp, xp}
\text{psi(i, j)=future(i)}
\text{enddo}
\text{enddo}
\text{end}

\text{---}

\text{subroutine feedback()}
\text{--- feedback}

\text{--- evolve \eta using last calculated psi;}

\text{integer i, xp, yp}
\text{real beta, tau, h}
\text{parameter (xp=xp_val, yp=yp_val, beta=BETA_VAL)}
\text{parameter (tau=t_val, h=h_val)}
\text{complex psi(-xp:xp, -yp:yp)}
\text{real eta(-xp:xp, -yp:yp)}

\text{common /psi/ psi, /eta/ eta}
\text{do i=-xp+1, xp-1}
\text{eta(i, 0) = tau * beta / (2.0 * h) *}
\text{+ (cabs(psi(i+1, 0))**2 - cabs(psi(i-1, 0))**2) +}
\text{+ eta(i, 0) \}
\text{enddo}

\text{eta(xp, 0) = eta(xp, 0)}
\text{eta(-xp, 0) = eta(-xp, 0)}
\text{end}

\text{---}
subroutine initialise(step)

integer xp, yp, i, step, xpp, ypp
parameter(xp=xp_val, yp=yp_val)
character dum*5, pling*l
complex psi(-xp:xp, -yp:yp)
real eta(-xp:xp, -yp:yp)
real t, h, k, a
logical psifound, etafound
character*40 psiinfile, etainfile
common /psi/ psi, /eta/ eta

psiinfile= 'FileAllPsi.data'
etainfile= 'FileEta.data'

C Is there an initial condition?
inquire(file=psiinfile, exist=psifound)
inquire(file=etainfile, exist=etafound)

C If not, make one;
if  (.not.psifound) .and. (.not.etafound) then
   print*, 'Initial conditions not found - producing own at t=0'
call psisetupO
   call etasetupO
   step=0
   return
endif

C If so, then read it in!
open(unit=12, file=psiinfile, +form = 'unformatted', status= 'old')
read(12)pling, xpp, ypp, t, h, k, a
print*, 'Reading in previous PSI file...'
do i=1, 100
   read(12, end=5, err=20)dum, step
   read(12, end=20, err=20)psi
   print*, dum, step
end do
C shouldnt reach this bit!
stop

open(unit=12, file=etainfile, +form = 'unformatted', status= 'old')
read(12)pling, xpp, ypp, t, h, k, a
print*, 'Reading in previous eta file...'
do i=1, 100
APPENDIX A. PROGRAM CODE

```fortran
read(12, end=10, err=20)dum, step
read(12, end=20, err=20)eta
print*, dum, step
end do
C shouldn't reach this bit!
stop
10 print*, 'End of file encountered...
print*, 'Initial Condition retrieved...
close(unit=12)
return
20 close(unit=12)
call abort()
end

C --------
subroutine psisetup()
C
integer xp, yp, i
real h, x
parameter(xp=xp_val, yp=yp_val, h=h_val)
complex psi(-xp:yp, -xp:yp)
common /psi/ psi
print*, 'Computing initial psi...
do i=-xp, xp
x=real(i)*h
psi(i, 0)=initial_psi_val
enddo
eend

C --------
subroutine etasetup()
C
integer xp, yp, i
real h, x
parameter(xp=xp_val, yp=yp_val, h=h_val)
real eta(-xp:yp)
common /eta/ eta
print*, 'Computing initial eta...
do i=-xp, xp
x=real(i)*h
eta(i, 0)=initial_eta_val
enddo
eend
```
APPENDIX A. PROGRAM CODE

C          subroutine writfil(n)
          integer n
          call write_psi(n)
          call write_eta(n)
          end

C          subroutine write_psi(n)

C          integer xp, yp, n
            parameter(xp=xp_val, yp=yp_val)
            complex psi(-xp:xp, -yp:yp)
            common /psi/ psi
          write(11)'Step=', n
          write(11)psi
          end

C          subroutine write_eta(n)

C          integer xp, yp, n
            parameter(xp=xp_val, yp=yp_val)
            real eta(-xp:xp, -yp:yp)
            common /eta/ eta
          write(13)'Step=', n
          write(13)eta
          end

C          real function itgrall(choice, i, j)

C          integer xp, yp, i, j, choice*2
            parameter(xp=xp_val, yp=yp_val)
            real eta(-xp:xp, -yp:yp)
            complex psi(-xp:xp, -yp:yp)
            common /psi/ psi, /eta/ eta

C choice is not used at present - may be used when second
C stability condition is implemented

C goto (1, 2), choice
itgrall=(1.0+eta(i, 0))*cabs(psi(i, 0))**2

A.1.3 2D ADI non-homogeneous PDE solver

Broken driver (progs/n52.1-2d.F)

Author: Wilf (sep2gw)
creation date: 26-5-93 (5/26/93)
last revision date: %G%
revision from: new35atmos.df & n37a_1d.F
revision: n52-1_2d No.: 52.1
include file revision: 52.1

Notes: Complete re-hash of our fave 2D program!

objects: filadmn3 simpson2 tridiag3 getout bailout2
objects: finish sig1 sig2 sig15 intergr mailme absreals

libraries: maths.a etc.a signals.a

Using an alternating direction-implicit scheme
Component by component splitting up method
psi is a 2D array which at any time t holds the solution

I/O:
unformatted killfile [ Psi(x, y, N) ]
temporary message file
formatted output file [ energy, Psi(0, 0, t) ]
unformatted output file [ Psi(x, y, t) ]
unformatted input file [ Psi(x, y, ?) ]

#include "n52-1_2d.h"

program L_difuse

integer xp, yp, tp, a, iv, step, i, j
integer*2 one
real t, h, k, a, simpE, energy, maxnrg, minnrg, diff, value, timebit

parameter(tp=tp_val, xp=xp_val, yp=yp_val)
parameter(t=t_val, h=h_val, k=k_val, a=a_val)
parameter(iv=tp/no_of_graphs)

character*50 epofl, psiofl, kilfile
complex psi(-xp: xp, -yp: yp)
common /psi/ psi
APPENDIX A. PROGRAM CODE

```c
#ifdef INTERRUPTS
    integer sig1, sig2, sig15
    external sig1, sig2, sig15
    complex oldpsi(-xp:xp, -yp:yp)
    logical int
    common /kill/ oldpsi, n, /int/ int

#ifdef SUN
    integer signal
#endif
#endif

data one /1/

print*, 'Setting up internal constants...

epofi=out_file//'EP.data'
psiofl='B'//out_file//'Psi.data'
kifile='B'//out_file//'Killed_Psi.data'

C Set appropriate IEEE flag(s);

#ifdef SUN
    call abrupt_underflow()
#endif

call filadmn3(epofi, psiofl, .true.)
C initialise the output files;

write(11)'!', xp, yp, t, h, k, a
write(*, 10)xp, -xp, -yp, yp, t, h, k

10 format('Time point : ', i6, '/',
    'X range : ', i4, '<->', i4, '/',
    'Y range : ', i4, '<->', i4, '/',
    'Parameters (t, h, k) : ', 3(e7.2, x))

#elifdef INTERRUPTS
C Sort out some signal handling;

data int /false,/
open(8, file=kifile, status='unknown', form='unformatted')
write(8)'!', xp, yp, t, h, k, a
#endif
```

# ifdef SUN
i=signal(1, sig1, -1)
APPENDIX A. PROGRAM CODE

i=signal(2, sig2, -1)
i=signal(15, sig15, -1)

# elseif LINUX
  call signal(1, sig1)
call signal(2, sig2)
call signal(15, sig15)
# endif

C find initial condition;

  print*, 'Initialised signal handling'

#endif

C We need to setup the constants before finding the I.C. in case we
C need to generate it. BUT one of the constants is dependent on step,
C found by init2D(), so we now have a pre-init2D() to find x, y and their
C squares, followed by init2D() which uses x, y, x2, y2 and finds step, which
C tcsetup() uses to initialise g(x, y, i): the RHS. Hopefully no more
C interdependence. (20-10-93)
C tcsetup() call replaced by rhs() call. (28-11-93)

call pre_init2D()

print*, 'Finding initial condition...
call init2D(step)

print*, 'Calculating time constants...
call timeconsts()

call simpson2(one, h, k, xp, yp, energy)
maxnrg=energy*1.1
minnrg=energy*0.9

C forall time;

do n=step+2, tp+step, 2

C calc time steps 1, 2;

print*, ' ' 
call rhs(n-1)
timebit=EPSILON*(t*real(n))**2

#ifdef INFO

print*, ' C Psi @ (0, 0): ', psi(0, 0)
print*, ' Value @ (0, 0): ', cabs(psi(0, 0))**2
print*, 'step 1...
call xarray()
print*, ' C Psi @ (0, 0): ', psi(0, 0)
print*, ' Value @ (0, 0): ', cabs(psi(0, 0))**2
print*, 'step 2...

#else

print*, ' C Psi @ (0, 0): ', psi(0, 0)
print*, ' Value @ (0, 0): ', cabs(psi(0, 0))**2
print*, 'step 1...
call xarray()
print*, ' C Psi @ (0, 0): ', psi(0, 0)
print*, ' Value @ (0, 0): ', cabs(psi(0, 0))**2
print*, 'step 2...'
#endif
call yarray2()
prints, ' C Psi @ (0, 0): ', psi(0, 0)
prints, ' Value @ (0, 0): ', cabs(psi(0, 0))**2
prints, 'step 3...'
call yarray3()
prints, ' C Psi @ (0, 0): ', psi(0, 0)
prints, ' Value @ (0, 0): ', cabs(psi(0, 0))**2
prints, 'step 4...'
call yarray()
#else
    call xarray()
call yarray2()
call yarray3()
call xarray()
#endif
C Save value of psi in 'kill' common block
#if define INTERUPTS
    do i = -xp, xp
        do j = -yp, yp
            oldpsi(i, j) = psi(i, j)
        enddo
    enddo
#endif
#if define INTERUPTS
    do i = -xp, xp
        do j = -yp, yp
            oldpsi(i, j) = psi(i, j)
        enddo
    enddo
#endif
C check const. E;
*    call simpson2(t, h, k, xp, yp, simpE)
value = cabs(cmplex(a_val*exp(-timebit), 0.))**2
prints, 'Time level: ', n
prints, ' C Psi @ (0, 0): ', psi(0, 0)
prints, ' R Ana ......... : ', a_val*exp(-timebit)
prints, ' Value @ (0, 0): ', cabs(psi(0, 0))**2
prints, ' Analytic ..... : ', value
prints, ' Diff @ (0, 0): ', value - cabs(psi(0, 0))**2
prints, ' Energy : ', 'Not calculated'
#endif
#if define BAILOUT
    if (simpE.gt.maxnrg.or.simpE.lt.minnrg)
        call bailout2(n, t, energy, simpE)
#endif
#if define COMPARE
    call compare(timebit, diff)
prints, ' Overall absolute avaraged ratio: ', diff
APPENDIX A. PROGRAM CODE

write(10, *) n, value, cabs(psi(0, 0))**2, diff
#else
write(10, *) n, value, cabs(psi(0, 0))**2
#endif

C store main array (if reqd) and finish if interrupted;
if (mod(n, iv).eq.0) then
call writfil(n)
#endif INTERUPTS
elseif (int) then
call writfil(n)
call finish()
#endif
endif
endif
enddo
#endif INTERUPTS
call finish()
#endif

C return to OS;
end

subroutine pre_init2D()
C Just deals with setting up x, y and their squares as arrays
C for speed access
C see also absreals()
C SUPPLIES /space2/: squares of x, y
integer xp, yp, i, j
parameter(xp=xp_val, yp=yp_val)
real x2(-xp:xp), y2(-yp:yp)
common /space2/ x2, y2

do i = -xp, xp
x2(i) = (real(i)*h_val)**2
enddo

do j = -yp, yp
y2(j) = (real(j)*k_val)**2

APPENDIX A. PROGRAM CODE

enddo

end

C

subroutine init2D(step)

C Initialises Psi either by an input file or set of supplied conditions
C if file does not exist.
C see also incond()

C TAKES step: current time step

integer xp, yp, i, step, xpp, ypp
parameter(xp=xp_val, yp=yp_val)
character dum*5, pling*l
complex psi(-xp:xp, -yp:yp)
real t, h, k, a
logical found
character infile*40
common /psi/ psi

infile='B'/in_file//'Psi.data'

C Is there an initial condition?
C inquire(file=infile, exist=found)

C If not, make one;

if (.not.found) then
  print*, 'No initial condition found - producing own at t=0'
call incond()
  step=0
  call writfil(step)
  return
endif

C If so, then read it in;

open(unit=12, file=infile,
+form='unformatted', status='old')
read(12)pling, xpp, ypp, t, h, k, a

if (xpp.ne.xp.or.ypp.ne.yp) then
  print*, 'Problem with initial condition: ',
  'grid size does not match'
  stop
endif

do i=1, 100
  read(12, end=10, err=20)dum, step
  read(12, end=20, err=20)psi
```
print*, dum, step
end do

10 print*, 'End of file encountered...
print*, 'Initial Condition retrieved.
close(unit=12)
return

20 close(unit=12)
print*, 'Big problems with binary input file :-0'
call getout()
end

C ----

subroutine inicond()

C Used if input file not found. Sets psi to parameter set in cpp file.
C Requires pre_init2D() to have set up x, y arrays.
C REQUIRES /space2/
C SUPPLIES /psi/

integer xp, yp, i, j
parameter(xp=xp_val, yp=yp_val)
complex psi(-xp*xp, -yp*yp)
real x2(-xp*xp), y2(-yp*yp)

common /psi/ psi, /space2/ x2, y2

print*, 'Computing Init. Cond.

do i=-xp, xp
do j=-yp, yp
psi(i, j)=initial_psi_val
enddo
enddo

C ----

subroutine rhs(step)

C Produces an exact g(x, y, t) [rhs] for a given (step) time level.
C TAKES step: current step level
C REQUIRES /space2/, /eta/
C PROVIDES /g/: rhs

integer xp, yp, i, j, step
real tau, time, f, timebit
parameter(xp=xp_val, yp=yp_val, tau=t_val)
```
APPENDIX A. PROGRAM CODE

```fortran
real x2(-xp:xp), y2(-yp:yp)
complex g(-xp:xp, -yp:yp)
real eta(-xp:xp, -yp:yp)

common /eta/ eta
common /g/ g
common /space2/ x2, y2

C Calculate g(x, y, t) @ t=step*tau;

print*, ' Calculating space values of g(x, y, t) @ t step', step

time=tau*real(step)
timebit=EPSILON*time**2

do i=-xp, xp
  do j=-yp, yp

  #ifdef HOMOGENEOUS
    g(i, j)=complex(0., 0.)
  #else
    f=a_val*exp(-x2(i)-y2(j)*(-timebit))
    g(i, j)=complex( +4.*f*(1.+eta(i, j))*(x2(i)+y2(j)1. )+ +8.*eta(i, j)*f*(x2(i)+y2(j) )+ -4.*EPSILON*f*time )
  #endif

  enddo
  enddo

C

C subroutine timeconsts()

C

#ifdef 1
  call tcsetup()
#else

C Calculates time constants for use throughout the simulation.
C REQUIRES /space2/
C PROVIDES /xu/: tc's for x sweep, /yu/: tc's for y sweep
C PROVIDES /eta/: eta

integer xp, yp, i, j
parameter(xp=xp_val, yp=yp_val)
real eta(-xp:xp, -yp:yp), w(-xp:xp, -yp:yp)
```

real x2(-xp:xp), y2(-yp:yp), x, y

common /eta/ eta, /xu/ xu, /yu/ yu, /w/ w
common /space2/ x2, y2

C !!! eta_val needs access to x2(i) and y2(j) !!!

do i=—xp, xp
   do j = —yp, yp
       x=real(i)*h_val
       y=real(j)*k_val
       eta(i, j)=eta_val
       w(i, j)=1.0+eta(i, j)
       xu(i, j)=complex( -4.0*x*eta(i, j), 4.0*y*eta(i, j) )
       yu(i, j)=complex( -4.0*y*eta(i, j), -4.0*x*eta(i, j) )
   enddo
enddo

#else

cx(i, j)=-dy(i, j)*k/h
    cy(i, j)=cx(i, j)*h/k
else

cx(i, j)=0.0
    cy(i, j)=0.0
APPENDIX A. PROGRAM CODE

```c
#include

#ifndef
enddo
enddo

C display check values;

C mem = xarray_size * yarray_size * no_of_bytes_for_real * no_of_arrays

mem=(2.0*xp+1)*(2.0*yp+1)+4*9/1024
printf( "Estimated memory usage for array structures: \( \text{mem} \) \[K\]\)

end

C

real function integral(choice, i, j)

integer xp, yp, i, j
integer*2 choice
parameter(xp=xp_val, yp=yp_val)
real eta(-xp:xp, -yp:yp)
complex psi(-xp:xp, -yp:yp)
common /psi/ psi, /eta/ eta

C choice is not used at present - may be used when second
C stability condition is implemented
*
  goto (1, 2), choice

  integral=(1.0+eta(i, j))*cabs(psi(i, j))**2

end

C

subroutine writfil(n)

integer xp, yp, n
parameter(xp=xp_val, yp=yp_val)
complex psi(-xp:xp, -yp:yp)
common /psi/ psi

write(11) 'Step=', n
write(11) psi

end

C

subroutine xarray()
```

```c
```
integer xp, yp, i, j
real b
parameter(xp=xp_val, yp=yp_val, b=2.0/t_val)
real cx(-xp:yp, -xp:yp), dx(-xp:yp, -xp:yp), ex(-xp:yp, -xp:yp)
complex mb(-xp:yp), future(-xp:yp)
complex psi(-xp:yp, -xp:yp)
common /psi/ psi, /xtc/ cx, dx, ex

C forall y;
do j=-yp, yp

C calc matrix vars mu, md, ml @ interior points

do i=-(xp-1), xp-1
  mbu(i)=cmplx(-dx(i, j)-ex(i, j), -cx(i, j))
  mbd(i)=cmplx(2.0*ex(i, j), b)
  mb(i)=cmplx(dx(i, j), cx(i, j))
  md(i)=cmplx(-2.0*ex(i, j), b)
  mb(i)=psi(i+1, j)*mbu(i)+psi(i, j)*mbd(i)+psi(i-1, j)*mb(i)
endo

C calc matrix vars mu, md, ml and mb @ boundaries

  mu(-xp)=cmplx(dx(-xp, j)+ex(-xp, j), cx(-xp, j))
  md(-xp)=cmplx(-2.0*ex(-xp, j), b)
  mbu(-xp)=cmplx(-dx(-xp, j)-ex(-xp, j), -cx(-xp, j))
  mbd(-xp)=cmplx(2.0*ex(-xp, j), b)
  mb(-xp)=psi(-xp, j)*mbd(-xp)+psi(-xp+1, j)*mbu(-xp)

  ml(xp)=cmplx(dx(xp, j)+ex(xp, j), cx(xp, j))
  md(xp)=cmplx(-2.0*ex(xp, j), b)
  mb(xp)=cmplx(-dx(xp, j)-ex(xp, j), -cx(xp, j))
  mbd(xp)=cmplx(2.0*ex(xp, j), b)
  mb(xp)=psi(xp, j)*mbd(-xp)+psi(xp-1, j)*mb(xp)

C calc next time level;
call tridiag3(-xp, xp, ml, mb, md, mu, future)

C store in main array;
do i=-xp, xp
  psi(i, j)=future(i)
endo
endo
APPENDIX A. PROGRAM CODE

---

end

subroutine yarray2()

integer xp, yp, i, j
real b
parameter(xp=xp_val, yp=yp_val, b=2.0/t_val)
real tau
parameter(tau=t_val)
complex ml(-yp+1:yp), md(-yp:yp), mbl(-yp:yp-1)
complex mbd(-yp:yp), mbu(-yp:yp-1)
complex psi(-xp:xp, -yp:yp)
complex g(-xp:xp, -yp:yp)

common /g/ g
common /psi/ psi, /yc/ cy, dy, ey

forall z;
do i=-xp, xp
C calc matrix vars mu, md, ml @ interior points
   do j=-(yp-1), yp-1
      mbu(j)=complex(-dy(i, j)-ey(i, j), -cy(i, j))
      mbd(j)=complex(2.0*ey(i, j), b)
      mb(i)=complex(dy(i, j)+ey(i, j), cy(i, j))
      md(j)=complex(-2.0*ey(i, j), b)
      mbl(j)=complex(-dy(i, j)-ey(i, j), cy(i, j))
      mb(i)=psi(i, j+1)*mbu(j)+psi(i, j)*mbd(j)+psi(i, j-1)*mbl(j)+
      +mu(j)+tau*g(i, j+1)+md(j)+tau*g(i, j)+ml(j)+tau*g(i, j-1)
   enddo

C calc matrix vars mu, md, ml and mb @ boundaries
   mu(-yp)=complex(dy(i, -yp)+ey(i, -yp), cy(i, -yp))
   md(-yp)=complex(-2.0*ey(i, -yp), b)
   mbu(-yp)=complex(-dy(i, -yp)-ey(i, -yp), cy(i, -yp))
   mbd(-yp)=complex(2.0*ey(i, -yp), b)
   mb(-yp)=psi(i, -yp)*mbd(-yp)+psi(i, -yp+1)*mbu(-yp)+
   +mu(-yp)+tau*g(i, -yp+1)+
   +md(-yp)+tau*g(i, -yp+1)
   ml(yp)=complex(dy(i, yp)+ey(i, yp), cy(i, yp))
   md(yp)=complex(-2.0*ey(i, yp), b)
   mbd(yp)=complex(-dy(i, yp)-ey(i, yp), cy(i, yp))
   mb(yp)=psi(i, yp)*mbd(yp)+psi(i, yp-1)*mbu(yp)+
APPENDIX A. PROGRAM CODE

+ md(yp)*tau*g(i, yp) +
+ ml(yp)*tau*g(i, yp - 1)

C calc next time level;

call tridiag3(-yp, yp, ml, mb, md, mu, future)

C store in main array;

do j = -yp, yp
  psi(i, j) = future(j)
enddo
enddo
end

C subroutine yarray3()

integer xp, yp, i, j
real b
parameter(xp=xp_val, yp=yp_val, b=2.0/t_val)
real tau
parameter(tau=t_val)
complex ml(-yp:yp), md(-yp:yp), mu(-yp:yp - 1)
complex mb(-yp:yp), mbd(-yp:yp), mbu(-yp:yp - 1)
complex mb(-yp:yp), future(-yp:yp)
common /psi, /cy, /dy, /ey
common /g,

C forall x;

do i = -xp, xp

C calc matrix vars mu, md, ml @ interior points

do j = -(yp - 1), yp - 1
  mbu(j) = cmplx(-dy(i, j) - ey(i, j), cy(i, j))
  mbd(j) = cmplx(2.0*ey(i, j), b)
  mb(j) = cmplx(dy(i, j) - ey(i, j), cy(i, j))
  mu(j) = cmplx(dy(i, j) + ey(i, j), cy(i, j))
  ml(j) = cmplx(-2.0*ey(i, j), b)
  mbl(j) = cmplx(-dy(i, j) + ey(i, j), -cy(i, j))
  mb(j) = psi(i, j + 1) + mbu(j) + mbd(j) + psi(i, j - 1) + mbd(j) +
          mbu(j) + mbl(j) +
          mbd(j) + mbl(j) +
          ml(j) + mbl(j) +
APPENDIX A. PROGRAM CODE

enddo

C calc matrix vars mu, md, ml and mb @ boundaries

mu(-yp)=cmplx(dy(i, -yp)+ey(i, -yp), cy(i, -yp))
md(-yp)=cmplx(-2.0*ey(i, -yp), b)
mbu(-yp)=cmplx(-dy(i, -yp)-ey(i, -yp), -cy(i, -yp))
mbd(-yp)=cmplx(2.0*ey(i, -yp), b)
mb(-yp)=psi(i, -yp)*mbd(-yp)+psi(i, -yp+1)*mbu(-yp)+
+ mbu(-yp)*tau*g(i, -yp+1) +
+ mbd(-yp)*tau*g(i, -yp)

ml(yp)=cmplx(dy(i, yp)+ey(i, yp), cy(i, yp))
md(yp)=cmplx(-2.0*ey(i, yp), b)
mbl(yp)=cmplx(-dy(i, yp)-ey(i, yp), -cy(i, yp))
mbd(yp)=cmplx(2.0*ey(i, yp), b)
mb(yp)=psi(i, yp)*mbd(-yp)+psi(i, yp-1)*mbl(yp)+
+ mbd(yp)*tau*g(i, yp) +
+ mb(yp)*tau*g(i, yp-1)

C calc next time level;
call tridiag3(-yp, yp, ml, mb, md, mu, future)

C store in main array;
do j = -yp, yp
   psi(i, j)=future(j)
enddo
endo
don

#define INTERUPTS

subroutine killprg()

integer xp, yp, n
parameter(xp=xp_val, yp=yp_val)
complex oldpsi(-xp:xp, -yp:yp)
common /kill/ oldpsi, n

C Dump Psi in aptly titled file;
print*, 'Received signal TERM(15)'
print*, 'Terminating immediately.'
write(8)'Step=', n
write(8)oldpsi
close(8)
print*, ' ?DEAD.data file written to disk. (Phew!) '

C Tell wilf that I have been killed;
open(9, file='message', form='formatted')
write(9, *)'Terminated due to signal TERM(15)'
write(9, *)'Step ', n, ' was reached'
call mailme('Has been killed', 'message')
close(9, status='delete')

C Shutdown gracefully;
close(11)
close(10)
stop
end

C

endif
#endif COMPARE

subroutine compare(timebit, correspond)

C Sorts out the relative correspondence between theory and practice.
C TAKES timebit: time invariant piece of f(x, y, t)
C REQUIRES /psi/, /space2/
C GIVES correspond: (see above)

integer xp, yp, i, j
parameter(xp=xp_val, yp=yp_val)
real x2(-xp:xp), y2(-yp:yp)
real correspond, normalise, timebit
complex actual, psi(-xp:xp, -yp:yp)

common /psi/, psi, /space2/, x2, y2

correspond=0.
normalise=0.
do i=-xp, xp
do j=-yp, yp
actual=compx(a_val*exp(-x2(i)*y2(j)*timebit), 0.)
correspond=correspond+cabs(actual-psi(i, j))**2
normalise=normalise+cabs(actual)**2
enddoendo

correspond=correspond/normalise
A.1.4 2D SOR Poisson solver

Stage one: create optimised $g^0$ (progs/sor.F)

```fortran
program SOR-TEST

real h, k, integral, t, a
parameter(t=t_val, a=a_val)
integer xp, yp, i, j, n, iter
parameter(xp=xp_val, yp=yp_val, h=h_val, k=k_val)
complex f(-xp:xp, -yp:yp), newf(-xp:xp, -yp:yp)
real eta(-xp:xp, -yp:yp), x, y, diff
complex c1, c2, c3, c4, c6
integer*2 two
real w
complex delta
common /newf/ newf, /eta/ eta

data two /2/

define the constants

cl = cmplx(2.0*k**2, 0.)
c2 = cmplx(2.0*h**2, 0.)
c3 = cmplx(-h*k**2, 0.)
c4 = cmplx(0., h**2+k)
c6 = cmplx(4.0*(k**2+h**2), 0.)

open(10, file = 'sor10000-19d.data', status='new')

start with w set to 1.0 (Gauss-Seidel)
```

APPENDIX A. PROGRAM CODE

w=1.9

C setup the test data;

    do i = -xp, xp
        do j = -yp, yp
            x=real(i)*h
            y=real(j)*k
            newf(i, j)=cmath(0., 0.)
            f(i, j)=cmath(0., 0.)
            psi(i, j)=initial_psi_val
            eta(i, j)=eta_val
            alpha(i, j) = psi(i, j)*cmath(1.0+eta(i, j), 0.)
        enddo
    enddo

C start the iteration process;

    do iter = 1, 10000

C now lets do something;

        diff=0.0

        do j=-yp+1, yp-1
            do i=-xp+1, xp-1

                delta = w*(
                 + c1* ( newf(i-1, j) + f(i+1, j ) )+
                 + c2* ( newf(i, j-1) + f(i, j+1 ) )+
                 + c3* ( alpha(i+1, j) - alpha(i-1, j ) )+
                 + c4* ( alpha(i, j+1) - alpha(i, j-1 ) )
                 + ) / c5 - f(i, j)
                 +

                 newf(i, j)=f(i, j)+delta
                 diff=diff+cabs(delta)

            enddo
        enddo

        print*, 'w=', real(w), ' iteration=', iter,
        + ' diff=', diff
        write(10, *) iter, diff
        if (diff.lt.0.01) then
            open (11, file = 'sor-19d.data', status='unknown',
                 + form='unformatted')
            write(11)'!', xp, yp, t, h, k, a
            write(11)'Step=', n
            write(11)newf
            close(11)
        close (10)
APPENDIX A. PROGRAM CODE

stop
endif
do i = —xp, xp
do j = —yp, yp
  f(i, j)=newf(i, j)
  enddo
dendo
dendo
end

C —

Stage two: potential enstrophy solver (progs/sor2.F)

C Author: Wilf (eep2gw)
C creation date: 1-3-94
C last revision date:
C revision from:
C revision: sor No.: 2
C Notes: SOR driver for grid 255x255
C objects: simpson2
C libraries: maths.a

#include "2dparams37.h"

program SOR_MAIN
real h, k, integral, t, a
integer xp, yp, i, j, n, step
parameter(xp=xp_val, yp=yp_val, h=h_val, k=k_val, a=a_val)
complex f(-xp:xp, -yp:yp)
real eta(-xp:xp, -yp:yp), x, y
complex w, c1, c2, c3, c4, c5
character*256 prog
common /psi/ psi, /eta/ eta

call openfile(t, prog)

C define the constants

w = cmplx(1.9, 0.)
c1 = cmplx(2.0*k**2, 0.)
c2 = cmplx(2.0*h**2, 0.)
c3 = cmplx(-h*k**2, 0.)
c4 = cmplx(0., h**2+k)
c5 = cmplx(4.0*(k**2+h**2), 0.)
APPENDIX A. PROGRAM CODE

open(10, file = 'sor-'/prog, status='new', form='formatted')

C setup the test data;
   do i = -xp, xp
      do j = -yp, yp
         x=real(i)*h
         y=real(j)*k
         eta(i, j)=eta_val
      enddo
   enddo

C do the job for each infile 'f';
   do n = 1, 1000
      C get the new psi();
      call nextpsi(step)
      C encode alpha();
      do i = -xp, xp
         do j = -yp, yp
            alpha(i, j) = psi(i, j)*complex(1.0+eta(i, j), 0.)
         enddo
      enddo
      call sor(w, c1, c2, c3, c4, c5, alpha, integral)
      write(10, *) real(step)*t , integral
      call flush(10)
   enddo

end

real function integral(choice, i, j)
integers xp, yp, i, j
integers choice
parameter(xp=xp_val, yp=yp_val)
real eta(-xp:xp, -yp:yp)
complex f(-xp:xp, -yp:yp)
common / f, / eta

goto (1, 2), choice
1 return
2 integral=cabs(f(i, j))/complex(1.0+eta(i, j), 0.)
return
end
APPENDIX A. PROGRAM CODE

C subroutine openfile(t, argv)

real t, h, k, a
integer xp, yp, i, large, xp, yp, rxp, ryp
parameter(rxp=xp_val, ryp=yp_val)
character argv*255, pling*l, step*5
complex f(-rxp:rxp, -ryp:ryp)
common /f/ f

i=argc()
if (i.ne.1) then
    print*, 'Please specify in-filename (only) on command line.'
    stop
endif

call getarg(i, argv)
open(11, file=argv, status='old', form='unformatted')
read(11) pling, xp, yp, t, h, k, a
if (xp.ne.rxp.or.yp.ne.ryp) then
    print*, 'grid parameters do not match.'
    print*, 'change params to xp_val=*, xp, and yp_val=*, yp'
    stop
endif
print*, 'grid: ', xp, 'x ', yp, ' t= ', t, ' h=', h, ' k=', k, ' a=', a

C initialize the poisson grid;
open(12, file='sor-19d.data', status='old', form='unformatted')
read(12) pling, xp, yp, t, h, k, a
print*, 'grid: ', xp, 'x ', yp, ' t= ', t, ' h=', h, ' k=', k, ' a=', a
read(12) step, i
read(12) f
return

end

C subroutine nextpsi(istep)

integer xp, yp, istep
parameter(xp=xp_val, yp=yp_val)
complex psi(-xp:xp, -yp:yp)
complex step
character step*5
APPENDIX A. PROGRAM CODE

common /psi/ psi
read(11, end=10, err=20) step, istep
read(11, end=20, err=20) psi
print*, 'step ', istep, ' retrieved'
return

10 print*, 'End of file encountered...

150 close(unit=11)
return

20 close(unit=12)
call abort()
end

subroutine sor(w, c1, c2, c3, c4, c5, alpha, intnew)

integer xp, yp
parameter(xp=xp_val, yp=yp_val)
real h, k
parameter(h=h_val, k=k_val)
real diff, idiff, intnew, intold
integer iter, i, j
complex c1, c2, c3, c4, c5, w, delta
complex alpha(-xp:xp, -yp:yp), f(-xp:xp, -yp:yp)
integer*2 two

common /f/ f
data two /2/
intold=0.0
intnew=0.0

do iter= 1, 10000

diff=0.0

190 do j=-yp+1, yp-1
  do i=-xp+1, xp-1
    delta = w*(
      + (c1* ( f(i-1, j) + f(i+1, j ) )+ 
      + c2* ( f(i, j-1) + f(i, j+1) )+ 
      + c3* ( alpha(i+1, j) - alpha(i-1, j) )+ 
      + c4* ( alpha(i, j+1) - alpha(i, j-1) )
    }
  

APPENDIX A. PROGRAM CODE

\[ f(i, j) = f(i, j) + \delta \]
\[ \text{diff} = \text{diff} + \text{cabs}(\delta) \]

\text{C now lets integrate over the entire grid for our result;}

\[ \text{if (diff .LT. 0.1) then} \]
\[ \text{call simpson2(two, h, k, xp, yp, intnew)} \]
\[ \text{idiff} = \text{abs(intnew} - \text{intold)} \]
\[ \text{intold} = \text{intnew} \]
\[ \text{print*, 'i:', iter, ' overall_diff:', diff, ' idiff:', idiff, ' integral:', intnew} \]
\[ \text{if (idiff .LT. 0.01 .AND. diff .LT. 0.1) return} \]
\[ \text{else} \]
\[ \text{print*, 'i:', iter, ' overall_diff:', diff} \]
\[ \text{endif} \]

\text{endif} \]

\text{print*, 'Iteration limit exceeded (10000)'}
\text{stop}
\text{end}

A.1.5 PVM test programs

Header (pvm/bigbro1.h)

#define PSIMSG 3 /* msgtag for psi transfer */
#define TIDMSG 4 /* msgtag for tid array transfer */
#define NPROC 4 /* number of child processes */
#define SWEEPS 10 /* sweeps per direction (2+) */
#undef SYMMETRIC /* if symmetric */

/* Grid parameter values */
#define XP 7 /* No. of x-dir grid points */
#define YP 7 /* No. of y-dir grid points */
#define H 1 /* Value of dx, space spacing */
#define K H /* value of dy, space spacing */
#define TAU 0.001 /* Value of tau, time spacing */
#define TIME_LEVELS 1000 /* No. of time levels */

/* Equation parameter values */
#define A -0.5 /* Value of eq-n constant a */
#define INITIAL_PSI_VAL cmplx( exp( ( — x**2 — y**2 ) ), 0.0)
#define ETA (A * exp ( — x**2 — y**2 ))

/* Disk I/O parameter values */
#define in_file 'null' /* File prefix of initialization data */
#define out_file 'bigbrol' /* File prefix of output files */

/* Misc parameter values */
#define NO_OF_GRAPHS 20 /* No. of data outputs per run */
#define bailout_on

Controller/parent (pvm/bigbro1.F)

program bigbro
C
#include "bigbro1.h"

#include PPVM_H

integer nproc, i, xp, yp, tp
real h, k, a, t
parameter(xp = XP, yp = YP, tp = TIME_LEVELS)
parameter(h = H, k = K, a = A, t = TAU)
parameter(nproc = NPROC)
integer mytid, info
integer tids(nproc)
integer step
character*50 epofl, psiofl, kilfile

epofl = out_file//'EP.data'
psiofl = 'B'//out_file//'Psi.data'
kilfile = 'B'//out_file//'DEAD.data'
call filadm3(epofl, psiofl, 1)

C initialise the output files

write(11) '!', xp, yp, t, h, k
write(*, 10) tp, -xp, xp, -yp, yp, t, h, k

10 format(' Time points: ', i6, /,
+ ' I range: ', i4, ' -> ', i4, /,
+ ' Y range: ', i4, ' -> ', i4, /,
+ ' Parameters (t, h, k): ', 3(e7.2, x))

C join pvm;

call pvmfmytid( mytid )

print*, ' controller = ', mytid
APPENDIX A. PROGRAM CODE

C Fire up the children;

print*, 'spawning ', nproc, ' children'
call pvmfspawn('/home/wilf/f77/childl', PVMDEFAULT, '*', nproc, tids, info)
if (info.lt.0) call fatal('pvmfspawn: childl')
print*, 'spawn successful'

C tcsetupQ is the childrens job saves commun.

C initialise and send psi, commit it to disk;

call init2D(step)
call send_psi(tids, nproc)
call writfil(step)
print*, 'entering blocking receive mode'
call blockrec(step)

C wipe out the children;

do i = 1, nproc
   call pvmkill(tids(i), info)
   if (info.lt.0) call fatal('unable to kill a child')
end do

C leave pvm;

call pvmfexit(info)

C return to OS;

end

C subroutine send_psi(tids, nproc)

C include FPVM_H

integer i, j, xp, yp, bufid, info, nproc
integer tids(nproc)
parameter(xp = XP, yp = YP)
complex strip(-xp:xp)
complex psi(-xp:xp, -yp:yp)

common /psi/ psi

C initialise send buffer;

call pvmfinitsend(PVMRAW, bufid)
if (bufid.lt.0) call fatal('pvmfinitsend: psi')

C first task is to let the children know who they are;
APPENDIX A. PROGRAM CODE

call pvmfpkint(tids, aproc, 1, info)
if (info.lt.0) call fatal('pvmfpkint: tid array')
call pvmfmcast(aproc, tids, TIDMSG, info)
if (info.lt.0) call fatal('pvmfmcast: The tids')

print*, 'sent tids to children'
C send psi in strips to all child processes;
do j = -yp, yp
  do i = -xp, xp
    strip(i) = psi(i, j)
  end do
end do
C the strip number;
call pvmfinitsend(PVMRAW, bufid)
call pvmfpkint(i, 1, 1, info)
if (info.lt.0) call fatal('pvmfpkint: strip number')
C the data itself;
call pvmfpkcmplx(strip, 2*xp + 1, 1, info)
if (info.lt.0) call fatal('pvmfpkcmplx: strip data')
call pvmfmcast(aproc, tids, PSIMSG, info)
if (info.lt.0) call fatal('pvmfmcast: The strip')
ifdef DEBUG
  print*, 'multicast strip ', i, ' ', j
endif
end subroutine blockrec(step)
C
integer step, bufid, bytes, msgtag, tid, info, sweeps
integer i, j, xp, yp, strip_no, iv, tp, graphs, n
parameter(xp = XP, yp = YP, tp = TIME-LEVELS)
parameter(graphs = NO_OF_GRAPHS)
parameter(sweeps = SWEEPS)
parameter(iv = tp / (sweeps * graphs) )
complex xstrip(-xp: xp), ystrip(-yp:yp)
complex psi(-xp:xp, -yp:yp)
real energy, h, k
parameter(h = H, k = K)
* integer2 one
common /psi/ psi
* data one /1/

do n = step + 1, tp + step

C REMEMBER, no values are valid until both directions have been swept!

do j = -yp, yp

    call pvmrecv(-1, -1, bufid)
call pvmbufinfo(bufid, bytes, msgtag, tid, info)

if (msgtag.eq.PSIMSG) then
    call pvmfupkint(strip_no, 1, 1, info)
    if (info.lt.0) call fatal('pvmfupkint')
call pvmfupkcomplex(ystrip, 2*yp + 1, 1, info)
    if (info.lt.0) call fatal('pvmfupkcomplex')
call load_yarray(xp, yp, psi, ystrip, strip_no)
    else
        print*, 'received message tag: ', msgtag
call fatal('unknown msgtag in receive')
endif
#endif DEBUG
print*, 'got strip ', strip_no, ' from ', tid
print*, ystrip
#endif

end do

C interim;

    do i = -xp, xp

        call pvmrecv(-1, -1, bufid)
call pvmbufinfo(bufid, bytes, msgtag, tid, info)

    if (msgtag.eq.PSIMSG) then
        call pvmfupkint(strip_no, 1, 1, info)
        if (info.lt.0) call fatal('pvmfupkint')
call pvmfupkcomplex(xstrip, 2*xp + 1, 1, info)
        if (info.lt.0) call fatal('pvmfupkcomplex')
call load_xarray(xp, yp, psi, xstrip, strip_no)
        else
            print*, 'received message tag: ', msgtag
call fatal('unknown msgtag in receive')
endif
#endif DEBUG
print*, 'got strip ', strip_no, ' from ', tid
print*, xstrip
#endif

end do

C Now we can take a peek at the results;
* call simpson2(one, h, k, xp, yp, energy)

print*, 'Step: ', n, ' const: ', energy,
+ cabs: ', cabs(psi(0, 0) ) ** 2

write(10, *) n, energy, cabs( psi(0, 0) ) ** 2

if (mod(n, iv).eq.0) then
  call writfil(n)
endif
end do
return
end

C —

subroutine init2D(step)

integer xp, yp, i, step, xpp, ypp
parameter(xp = XP, yp = YP)
character dum*5, pling*1
complex psi(-xp:xp, -yp:yp)
real t, h, k, a
logical found
character infile*40

common /psi/ psi

infile='B'/in_file//'Psi.data'

C Is there an initial condition?

inquire(file = infile, exist = found)

C If not, make one;

if (.not.found) then
  print*, ' No initial condition found - producing own at t=0'
  call idatmem2()
  step = 0
  return
endif

C If so, then read it in!

open(unit = 12, file = infile,
+ form = 'unformatted', status = 'old')
read(12) pling, xpp, ypp, t, h, k, a

do i = 1, 50
  read(12, end = 10, err = 20) dum, step
  read(12, end = 20, err = 20) psi
APPENDIX A. PROGRAM CODE

print*, dum, step
end do

10 print*, 'End of file encountered...
print*, 'Initial Condition retrieved...
close(unit = 12)
return

20 close(unit = 12)
call abort()
end

C —

subroutine idatmem2()

integer xp, yp, i, j
parameter(xp = XP, yp = YP)
real a, h, k, x, y
parameter(a = A, h = H, k = K)
complex psi(-xp:xp, -yp:yp)
common /psi/ psi

print*, 'Computing Init. Cond...

do i = -xp, xp
  do j = -yp, yp
    x = real(i) * h
    y = real(j) * k
    psi(i, j) = INITIAL_PSI_VAL
    #ifdef DEBUG
       print*, cabs( psi(i, j) )
    #endif
  end do
end do

C —

subroutine writfil(n)

integer xp, yp, n
parameter(xp = XP, yp = YP)
complex psi(-xp:xp, -yp:yp)
common /psi/ psi

write(11) 'Step=', n
write(11) psi
print*, 'written step ', n, ' to disk'
Worker/child *(pvm/child1.F)*

```fortran
C program child1
C
#
include "bigbro1.h"
include FPVM_H
integer mytid, ptid, nproc
parameter(nproc = NPROC)
integer tids(nproc)

C sort out the pvm side;

 call pvmfmytid( mytid )
call pvmfparent( ptid )
if  (ptid.eq.PvmNoParent) call fatal('child orphan, must have parent. ')

C pvmfrecv() the tids of other children from parent;

call get_tids(tids, ptid)
print*, mytid, ': got tids'
call tcsetup()
print*, mytid, ': set up tc''s'

C we dont come back from this one;

call main_loop(mytid, tids, ptid)
end

C subroutine get_tids(tids, ptid)

integer nproc, bufid, ptid, info
parameter(nproc = NPROC)
integer tids(nproc)

call pvmfrecv(ptid, TIDMSG, bufid)
call pvmfupkint(tids, nproc, 1, info)
if (info.lt.0) call fatal('pvmfupkint: unpacking tid array')

return
end
C
```
subroutine main_loop(mytid, tids, ptid)
  C
  parameter(nproc = NPROC)
  integer tids(nproc)
  parameter(xp = XP, yp = YP)
  integer my_xlength, my_ylength
  integer my_xbit(2), my_ybit(2)
  complex xstrip(-xp: xp), ystrip(-yp: yp)
  complex psi(-xp: xp, -yp: yp)

  common /psi/ psi

  do n = 1, nproc
    if (tids(n).eq.mytid) me = n
  end do
  call which_bit(my_xbit, xp, me)
  call which_bit(my_ybit, yp, me)
  my_xlength = my_xbit(2) - my_xbit(1) + 1
  my_ylength = my_ybit(2) - my_ybit(1) + 1
  print*, 'startxbit: ', my_xbit(1), ', endxbit: ', my_xbit(2),
  ' length: ', my_xlength
  print*, 'startybit: ', my_ybit(1), ', endybit: ', my_ybit(2),
  ' length: ', my_ylength

  C go around and around until parent kills us;
  continue

  C receive (wait for) all of the x strips and load into psi;
  do i = -xp, xp - my_xlength
    call recv_xstrip()
  end do
  print*, 'got all x strips'

  C only process relevant xstrips
  do i = my_xbit(1), my_xbit(2)

  C load a y strip from psi;
  do j = -yp, yp
    ystrip(j) = psi(i, j)
  end do

  C do a sweep in the x direction;
APPENDIX A. PROGRAM CODE

\begin{verbatim}
call ysweep(ystrip, i)
C send the y strip;
    call send_strip(ystrip, yp, i, tids, ptid)
end do
print*, 'finished y sweep'
C receive (wait for) all of the y strips and load into psi;
    do j = -yp, yp - my_ylength
        call recv_ystrip()
    end do
print*, 'got all y strips'
C only process relevant y strips;
    do j = my_ybit(1), my_ybit(2)
    C load an x strip from psi;
        do i = -xp, xp
            xstrip(i) = psi(i, j)
        end do
    C do a sweep in the y direction;
        call xsweep(xstrip, j)
        call send_strip(xstrip, xp, j, tids, ptid)
    C send the x strip;
    end do
print*, 'finished x sweep'
C start all over again;
    goto 10
end
C

subroutine which_bit(bits, p, me)

  integer nproc, bits(2), n, p, me, strips, slide
  parameter(nproc = NPROC)
  integer bit(nproc), startbit(nproc), endbit(nproc)

  C calculate everyone's strips;
\end{verbatim}
APPENDIX A. PROGRAM CODE

strips = p*2 + 1
slide = -p

do n = 1, nproc
    bit(n) = strips / (nproc + 1 - n)
    strips = strips - bit(n)
    startbit(n) = slide
    slide = slide + bit(n)
    endbit(n) = slide - 1
end do

C assign me my strips;
bits(1) = startbit(me)
bits(2) = endbit(me)

return
end

C subroutine recv_xstrip()
---------------

integer xp, yp, strip_no, bufid, info
parameter(xp = XP, yp = YP)
complex strip(-xp:xp)
complex psi(-xp:xp, -yp:yp)

common /psi/ psi

call pvmfrecv(-1, PSIMSG, bufid)
if (bufid.lt.0) call fatal('pvmfrecv: PSIMSG')
call pvmfupkint(strip_no, 1, 1, info)
if (info.lt.0) call fatal('pvmfupkint: strip_no')
call pvmfupkcomplex(strip, 2*xp + 1, 1, info)
if (info.lt.0) call fatal('pvmfupkcomplex: xstrip')
call load_xarray(xp, yp, psi, strip, strip_no)

return
end

C subroutine recv_ystrip()
---------------

integer xp, yp, strip_no, bufid, info
parameter(xp = XP, yp = YP)
complex strip(-yp:yp)
complex psi(-xp:xp, -yp:yp)

common /psi/ psi
call pvmrecv(-1, PSIMSG, bufid)
if (bufid.lt.0) call fatal('pvmrecv: PSIMSG')

call pvmfupkint(strip_no, 1, 1, info)
if (info.lt.0) call fatal('pvmfupkint: strip_no')
call pvmfupkcmplx(strip, 2*yp + 1, 1, info)
if (info.lt.0) call fatal('pvmfupkcmplx: ystrip')

call load_yarray(xp, yp, psi, strip, strip_no)
return
end

C
--

subroutine send_strip(strip, p, strip_no, tids, ptid)
include PPVM_H

integer strip_no, p, nproc, ptid, bufid, info
parameter(nproc = NPROC)
complex strip(-p:p)
integer tids(nproc)

call pvmfinitsend(PVMRAW, bufid)
if (bufid.lt.0) call fatal('pvmfinitsend: send_strip')
call pvmfupkint(strip_no, 1, 1, info)
if (info.lt.0) call fatal('pvmfupkint: send_strip')
call pvmfupkcmplx(strip, 2*p + 1, 1, info)
if (info.lt.0) call fatal('pvmfupkcmplx: send_strip')
call pvmfmcast(nproc, tids, PSIMSG, info)
if (info.lt.0) call fatal('pvmfmcast: send_strip (children)')
call pvmfmcast(1, ptid, PSIMSG, info)
if (info.lt.0) call fatal('pvmfmcast: send_strip (parent)')
return
end

C
--

subroutine xsweep(xstrip, j)

integer xp, yp, i, j, sweep, sweeps
real b, t
parameter(t = TAU, sweeps = SWEEPS)
parameter(xp = XP, yp = YP, b = 2.0/t)
real dx(-xp:xp, -yp:yp), ex(-xp:xp, -yp:yp)
complex ml(-(xp-1):xp), md(-xp:xp), mu(-(xp-1))
complex mbl(-(xp-1):xp), mbd(-xp:xp), mbu(-(xp-1))
complex mb(-xp:xp)
complex xstrip(-xp:xp)

ifndef SYMMETRIC
real cx(-xp:xp, -yp:yp)
APPENDIX A. PROGRAM CODE

common /cx/ cx
#else
#define cx(i, j) 0.0
#endif
common /dx/ dx, /ex/ ex

C for strip number j, sweeps times;

do sweep = 1, sweeps

C calc matrix vars mu, md, ml @ interior points

do i = -(xp-1), xp-1
  mbu(i)=cmplx(-dx(i, j)-ex(i, j), -cx(i, j))
  mbd(i)=cmplx(2.0*ex(i, j), b)
  mbl(i)=cmplx(dx(i, j)-ex(i, j), cx(i, j))
  mu(i)=cmplx(dx(i, j)+ex(i, j), cx(i, j))
  md(i)=cmplx(-2.0*ex(i, j), b)
  ml(i)=cmplx(-dx(i, j)+ex(i, j), -cx(i, j))
  mbu(i)=cmplx(dx(i, j)+ex(i, j), -cx(i, j))
  mbd(i)=cmplx(-dx(i, j)+ex(i, j), cx(i, j))
  mbl(i)=cmplx(dx(i, j)-ex(i, j), cx(i, j))
  mub(i)=xstrip(i+1)*mbu(i)+xstrip(i)*mbd(i)+xstrip(i-1)*mbl(i)
end do

C calc matrix vars mu, md, ml and mb @ boundaries

mu(-xp)=cmplx(dx(-xp, j)+ex(-xp, j), cx(-xp, j))
md(-xp)=cmplx(-2.0*ex(-xp, j), b)
mbn(-xp)=cmplx(dx(-xp, j)-ex(-xp, j), -cx(-xp, j))
mbd(-xp)=cmplx(2.0*ex(-xp, j), b)
mb(-xp)=xstrip(-xp)*mbd(-xp)+xstrip(-xp+1)*mbu(-xp)

ml(-xp)=cmplx(dx(-xp, j)+ex(-xp, j), cx(-xp, j))
md(-xp)=cmplx(-2.0*ex(-xp, j), b)
mb(-xp)=xstrip(-xp)*mbd(-xp)+xstrip(-xp-1)*mbl(-xp)

C calc next time level;

call tridiag3(-xp, xp, ml, mb, md, mu, xstrip)
enddo

return
end

C —

subroutine ysweep(ystrip, i)

integer xp, yp, i, j, sweep, sweeps
real b, t
parameter(t = TAU, sweeps = SWEEPS)
parameter(xp = XP, yp = YP, b = 2.0/t)
real dy(-xp:xp, -yp:yp), cy(-xp:xp, -yp:yp)
complex ml(-(yp-1):yp), md(-yp:yp), nu(-yp:yp,-1)
complex mb(-(yp-1):yp), mbd(-yp:yp), mbu(-yp:yp,-1)
complex mb(-yp:yp)

#ifdef SYMMETRIC
real cy(-xp:xp, -yp:yp)
#endif
#include /cy/ cy

#define cy(i, j) 0.0
#endif
#include /dy/ dy, /ey/ ey

C for strip number i, sweeps times;
do sweep = 1, sweeps

C calc matrix vars mu, md, ml & interior points

do j = -(yp-1), yp-1
mbu(j)=complx(-dy(i, j)-cy(i, j), -cy(i, j))
mbd(j)=complx(2.0*ey(i, j), b)
mb(j)=complx(dy(i, j)-cy(i, j), cy(i, j))
md(j)=complx(-2.0*ey(i, j), b)
mu(j)=complx(dy(i, j)+ey(i, j), cy(i, j))
mb(j)=ystrip(j+1)*mbu(j)+ystrip(j)*mbd(j)+ystrip(j-1)*mb(j)

enddo

C calc matrix vars mu, md, ml and mb @ boundaries

mu(-yp)=complx(dy(i, -yp)+ey(i, -yp), cy(i, -yp))
md(-yp)=complx(-2.0*ey(i, -yp), b)
mbu(-yp)=complx(-dy(i, -yp)-ey(i, -yp), -cy(i, -yp))
mbd(-yp)=complx(2.0*ey(i, -yp), b)
mb(-yp)=ystrip(-yp)*mbd(-yp)+ystrip(-yp+1)*mbu(-yp)

ml(yp)=complx(dy(i, yp)+ey(i, yp), cy(i, yp))
md(yp)=complx(-2.0*ey(i, yp), b)
mb(yp)=complx(-dy(i, yp)-ey(i, yp), -cy(i, yp))
mbd(yp)=complx(2.0*ey(i, yp), b)
mb(yp)=ystrip(yp)*mb(yp)+ystrip(yp-1)*mbd(yp)

C calc next time level;
call tridiag3(-yp, yp, ml, mb, md, mu, ystrip)

enddo

return

to  end

C
APPENDIX A. PROGRAM CODE

```c
subroutine tcsetup()

integer xp, yp, i, j
real h, k, a, t, divish, divisk, x, y
parameter(xp = XP, yp = YP, a = A, h = H, k = K)
parameter(t = TAU)
parameter(divish = 2.0 * h ** 2, divisk = 2.0 * k ** 2)
real eta(-xp;xp, -yp:yp)
real dx(-xp:xp, -yp:yp), dy(-xp:xp, -yp:yp)
real cx(-xp:xp, -yp:yp), cy(-xp:xp, -yp:yp)

#ifndef SYMMETRIC
real cx(-xp:xp, -yp:yp), cy(-xp:xp, -yp:yp)
#endif

common /cx/, /cx/, /cy/, /cy/
common /dx/, /dx/, /ex/, /ex/
common /dy/, /dy/, /ey/, /ey/

do i = -xp, xp
  do j = -yp, yp
    x = real(i)*h
    y = real(j)*k
    eta(i, j) = ETA
    dx(i, j) = real(i)*eta(i, j)
    dy(i, j) = real(j)*eta(i, j)
    ex(i, j) = (1.0 + eta(i, j))/divish
    ey(i, j) = (1.0 + eta(i, j))/divisk
  #ifndef SYMMETRIC
    cx(i, j) = -dy(i, j)*k/h
    cy(i, j) = dx(i, j)*h/k
  #endif
  enddo
enddo
return
end
```

A.1.6 2D binary to 1D ASCII data converter

Translator for xmgr (progs/1dconvert.F)

C Author: Wilf (ep2gw)
C creation date: 22-3-93
APPENDIX A. PROGRAM CODE

C last revision date: %G% 11-8-93
C revision from: n 2 disp 2.1
C Notes: To convert 2D arrays into 1D slice graph for xmgr

C objects:
C libraries:

#include "ldconvert.h"

program display

integer n, xp, yp, r xp, r yp, step, i, yslice(11:61), filetype
integer conv, cons
parameter(xp=xp_val, yp=yp_val)
complex psi(-xp:xp, -yp:yp)
real eta(-xp:xp, -yp:yp)
real t, h, k, a
character dum*5, bigtext*69, test*l

data bigtext/' ' /
call input(filetype, cons, yslice)

C Read in the ID of the infile and convert to nice $;
read(10, end=10, err=30)test, r xp, r yp, t, h, k, a
if (xp.ne.rxp.or.yp.ne.ryp) then
   print*, 'Grid size values are bad - check include file'
stop
endif

write(bigtext,
   'h=', h, ' k=', k, ' t= ', t, ' a=', a, ' grid= ', 2*r xp, 'x ', 2*ryp
print*, bigtext

do conv=11, cons+10
   write(conv, '(a 2 , a 6 9 )')'# ', bigtext
enddo

C Read in the 2D data from each graph;
do n=0, no_of_graphs+2
   print*, 'Reading... ', n
   read(10, end=10, err=20)dum, step
   if (test.eq.'!') then
      read(10, end=10, err=20)psi
APPENDIX A. PROGRAM CODE

```
else if (test.eq.'E') then
    read(10, end=10, err=20) eta
else
    stop
endif

C For each graph, output the converted slices to unique files;

do conv=11, conv+10

    print*, 'Writing...', n, 'slice at y=', yslice(conv)
    write(conv, '(a2)')(a, ' (a2, a5, i5)')#', dum, stop
    if (test.eq.'!') then
        do i=—xp, xp
            write(conv, '(i5, x, a11.4)')i, cabs(psi(i, yslice(conv))**2)
        enddo
    else if (test.eq.'E') then
        do i=—xp, xp
            write(conv, '(i5, x, a11.4)')i, eta(i, yslice(conv))
        enddo
    else
        stop
    endif
endo
dendo

10 print*, 'End of file encountered...'
close(unit=10)
stop

20 print*, 'An error has occurred while reading the input file'
stop

30 print*, 'Problem with file type - inconsistent ID.'
stop

end

C

subroutine input(filetype, n, yslice)

C

integer i, filetype, n, yslice(11:61), yp
parameter(yp=yp_val)
character*50 fin, fout
character*15 type(3)
character*4 cyslice
```
APPENDIX A. PROGRAM CODE

```fortran
data type /'Psi.data', 'eta.data', 'FileAllPsi.data'/

print*, 'Which filetype 1:''Psi'' 2:''eta'' 3:''FileAllPsi'' ?'
read(*, *)filetype

fin='B'/out_file/type(filetype)
open(10, err=10, file=fin, status='old', form='unformatted')

print*, 'How many conversions (up to 50) ?'
read(*, *)n

print*, 'Please type in the x-slice y integers ', -yp, ' -> ', yp

do i=1, n+10
   read(*, *)yslice(i)
   write(cyslice, '(i4.3)')y
gaive
   if (cyslice(1:1).eq. '') cyslice(1:1)= '+'
   fout='Id'/cyslice//out_file//type(filetype)
   open(i, file=fout, status='unknown', form='formatted')
enddo

print*, 'Thankyou'
return

print*, 'File ', fin, ' does not seem to exist! :-(
stop
end

A.1.7 Math library routines

2D quadrature using Simpson's rule (maths/simpson2.f)

```
integer xp, yp, j
integer*2 choice
real h, k, ixs, ixy

ixy=ixs(choice, h, xp, -yp)+ixs(choice, h, xp, yp)

      do j=1-yp, yp-1, 2
        ixy=ixy+4.0*ixs(choice, h, xp, j)
      enddo

      do j=2-yp, yp-2, 2
        ixy=ixy+2.0*ixs(choice, h, xp, j)
      enddo

      ixy=ixy*k/3.0
  end

C          real function ixs(choice, h, xp, j)

C

integer i, j, xp
integer*2 choice
real h, itgral

ixs=itgral(choice, -xp, j)+itgral(choice, xp, j)

      do i=1-xp, xp-1, 2
        ixs=ixs+4.0*itgral(choice, i, j)
      enddo

      do i=2-xp, xp-2, 2
        ixs=ixs+2.0*itgral(choice, i, j)
      enddo

      ixs=ixs+h/3.0
  end

C

1D quadrature using Simpson's rule (maths/simpson2-1D.f)

C       Author: Wilf (wp2gw)
C       creation date: 9-3-93 for simp1D! (3/9/93)
C       last revision date: 5/25/93, 15-4-94
C       revision from: simp1D.f, from simpson.f
C       revision: @(#)simp1D.f No.: 1.2
C
C       Notes: 1D variation
### APPENDIX A. PROGRAM CODE

**C**

```fortran
subroutine simpson2_1D(choice, h, xp, ixs)
    implicit none
    integer xp, i
    integer*2 choice
    real h, ixs, itgral1

    ixs=itgral1(choice, -xp)+itgral1(choice, xp)

    do i=1-xp, xp-1, 2
        ixs=ixs+4.0*itgral1(choice, i)
    enddo

    do i=2-xp, xp-2, 2
        ixs=ixs+2.0*itgral1(choice, i)
    enddo

    ixs=ixs*h/3.0

end
```

**Complex Thomas method (maths/tridiag3.f)**

**C**

```fortran
subroutine tridiag3(lpts, upts, l, b, d, u, x)
    implicit none
    integer lpts, upts, i
    complex l(lpts+1:upts), b(lpts:upts), d(lpts:upts)
    complex u(lpts:upts-1), x(lpts:upts)

    Gauss elimination and LU decomposition:

    do i=lpts+1, upts
        l(i)=l(i)/d(i-1)
        d(i)=d(i)-l(i)*u(i-1)
        b(i)=b(i)-l(i)*b(i-1)
    enddo

    back substitution:

    x(upts)=b(upts)/d(upts)

    do i=upts-1, lpts, -1
```

*Author: Wilf (sep2gw)*

*creation date: 2/4/93 (5/24/93)*

*last revision date: 6/1/93*

*revision from: tridiag2.f, from tridiag1.f*

*revision: @(^)tridiag3.f No.: 1.3*

*Notes: Tridiagonal equations solver*
A.1.8 Signal handler library routines

Signal handler for sig HUP (signals/sig1.f)

```c
C creation date: 10-6-93 (6/10/93)
C last revision date: 6/11/93
C revision from:
C revision: @(ff)sig1.f No.: 1.1
C Notes: some signal handlers
C objects:
C libraries:

integer function sig1()
```

```
C Chill out, I'll finish in due course;
    sig1=1
    call intprg(sig1)
end
```

Signal handler for sig TRAP (signals/sig2.f)

```c
C creation date: 10-6-93 (6/10/93)
C last revision date: 6/11/93
C revision from:
C revision: @(ff)sig2.f No.: 1.1
C Notes: some signal handlers
C objects:
C libraries:

integer function sig2()
```

```
C Chill out, I'll finish in due course;
    sig2=2
    call intprg(sig2)
```
APPENDIX A. PROGRAM CODE

Signal handler for sig TERM (signals/sig15.f)

```c
C creation date: 10-6-93 (6/10/93)
C last revision date: 6/11/93
C revision from:
C revision: @(#)sig15.f No.: 1.1
C Notes: some signal handlers
C objects:
C libraries:

integer function sig15()

C Ooops, lets get out of here (gracefully)

sig15=15
call killprg()

end
```

Handle mailing to user (signals/mailme.f)

```c
C Author: Wilf (eeepgw)
C creation date:
C last revision date: %G%
C revision from:
C revision: %2%%M% No.: %I%
C Notes: general system mailer
C objects:
C libraries:

subroutine mailme(subject, fname)

characters(*) fname, subject
characters=15 prgname
characters=200 command

C Send mail to user, with subject and fname defined;

call getarg(0, prgname)
```
command = 'elm -s ' ' ' //prgname/'subject/'' 'whoami' < ' //fname
call system(command)
end

——

Signal handler (signals/intprg.f)

Author: Wilf (eepBgw)
creation date: 11-6-93 (6/11/93)
last revision date: %G%
revision from:
revision: %Z%M% No.: %I%
Notes: called by interrupt handler

objects:
libraries: 10

subroutine intprg(sig)
logical int
integer sig
character signal(2)*3

common /int/ int
data signal /'HUP', 'INT'/
int=.true.

print*, ' '
print*, 'Received signal ', signal(sig), '(', sig, ')
print*, 'Shutting down gracefully, please wait...'
call mailme('has been shutdown', '/dev/null')
end

——

Inform user of completed simulation (signals/finish.f)

Author: Wilf (eepBgw)
creation date: 11-6-93 (6/11/93)
last revision date: %G%
revision from:
revision: %Z%M% No.: %I%
APPENDIX A. PROGRAM CODE

Notes:

Libraries:

Subroutine finish()

Logical int
Common /int/ int

If (.not.int) call mailme('Completion ok.', '/dev/null')
Close(unit=8, status='delete')
Close(unit=11)
Close(unit=10)

Print*, 'Done.'
Stop

end

A.1.9 Miscellaneous library routines

Abort the simulation (etc/abort.f)

Author: Wilf (eep8gw)
Creation date: (unknown) pre 25-5-93 (5/25/93)
Last revision date: 5/25/93
Revision from: an early ns51d.f
Revision: G(#)abort.f No.: 1.4

Notes:

Subroutine abort

Print*, 'Aborting...'
Stop

end

Abort due to bad energy level (etc/bailout2.f)

Author: Wilf (eep8gw)
Creation date: 28-5-93 (5/28/93)
Last revision date: %G%
Revision from: bailout.f
Revision: %Z% No.: %I%
Notes: Die gracefully if energy conditions not met

objects:

libraries:

subroutine bailout2(step, tau, energy, badnrg)

real energy, badnrg, tau
integer step

call writfil(step)

open(9, file='message', form='formatted')
write(9, *) 'Terminating program due to erroneous ',
+ 'energy condition'
write(9, *) 'Initial energy level: ', energy
write(9, *) 'Energy level at step ', step,
+ ' time ', step*tau, ': ', badnrg

call mailme('Energy problem', 'message')

close(9, status='delete')

call finish()
APPENDIX A. PROGRAM CODE

read(*, '(18)', iostat=problem) doclean
endif

if (problem.gt.0) call getout()

if (doclean) then
    command=' /bin/rm -f ' //epofl
    call system(command)
    command=' /bin/rm -f ' //psiofl 30
    call system(command)
endif

open(unit=10, file=epofl, status='new', iostat=problem)
if (binary) then
    open(unit=11, file=psiofl, status='new', iostat=problem, +
        form='unformatted')
else
    open(unit=11, file=psiofl, status='new', iostat=problem)
endif

if (problem.gt.0) call getout()
end

A.1.10 UNIRAS visualisation routines

Driver (uniras/n37disp2d.F)

Author: Wilf (esp2gw)
creation date: 26-5-93 (5/26/93)
last revision date: %G%
revision from: n2disp30.df
revision: %E%%M% No.: %1%

Notes:

#include "n37disp2d.h"

program display(fname) display

integer n, xp, yp, unitno, rxp, ryp, step, gx, gy, i, j, pos
parameter(xp=xp_val, yp=yp_val, gx=gx_val, gy=gy_val, unitno=11)
complex psi(-xp:xp, -yp:yp)
real grid(-gx:gx, -gy:gy), t, h, k, a, cmax
character*5 dum
character*1 test
character*55 bigtext
character*55 stepstr
character*60 device
character*55 post, fname
data cmax /0.0/

device=groute_val

call psname(fname, post, pos)

open(unitno, file=fname, status='old', form='unformatted')

C Check to see if the parameters match;
read(unitno, end=10, err=20)test, rxp, ryp, t, h, k, a
if (rxp.ne.xp.or.ryp.ne.yp) then
print*, ' array dimensions of input & param. files ',
+ 'do not match'
print*, ' change parameter file and re-compile.'
print*, 'xp= ', xp, 'rxp= ', rxp, 'yp= ', yp, 'ryp= ', ryp
stop
endif

C Convert parameter data to display friendly $;
write(bigtext, '(a2, ell.4, a3, ell.4, a3, f6.2,
+ a6, ell.3, a1, l3))' )'h=', h, ' k=', k, ' a=', a,
+ ' grid=', 2*rxp, 'x', 2*ryp
print*, bigtext

C open UNIRAS;
call groute(device)

C Start reading in the data;
do n=1, no_of_graphs
read(unitno, end=10, err=20)dum, step
print*, 'graph: ', n, ' reading step: ', step
print*, '-----------------------------------------------'
read(unitno, end=10, err=20)psi
print*, 'converting raw data to |psi|^2 . . .'
do i=-gx, gx
do j=-gy, gy
grid(i, j)=cabs(psi(i, j))**2
if (n.eq.1) cmax = max(cmax, grid(i, j))
end do
end do

C set up default contour levels based on first psi dump;
   if (n.eq.1) cmax = cmax*1.05
   print*, 'value of psi(0, 0, t) = ', psi(0, 0)
   print*, 'psi(gx, gy, t) = ', psi(gx, gy)
   print*, 'value of centre grid point= ', grid(0, 0),
     ' at edge = ', grid(gx, gy)

C Do some more number to string conversion;
   write (stepstr, ' (a4, x , e10.3, 3x, a10, i5, x , a5, x, f6.2 )')
   + 'tau= ', t, ' step no. : ', step, 'time= ', testep
   print*, stepstr

C Let Uniras take over(!);
   call unidisp(grid, gx, gy, h, k, cmax,
     + stepstr, bigtext, fname)

C if outputting to POST then rename it;
   if (device(5:9).eq. 'mpost') then
      write (post(pos-1:pos), '  (i2 .2 )')n
      print*, 'moving ' 'post2' ' to ' '//post
      call system('mv post2 '//post)
      print*, '  '
   end if

10 print*, 'end of file encountered . . '
stop

20 print*, 'an error has occurred while reading ', fname
   print*, 'aborting . . '
stop

end

C

C subroutine psname(fname, post, i)
C
character(*) fname, post
integer i

do i=5, 50
   if (fname(i-3:i).eq.'file') then
Graph drawing code (uniras/unidisp.f)

**Author:** Wilf (eep2gw)
**Creation date:** 26-5-93 (5/26/93)
**Last revision date:** %G%
**Revision from:** n2disp30.df
**Revision:** %Z% %M% No.: %I%

**Notes:**
**Objects:**
**Libraries:**

```
subroutine unidisp(grid, gx, gy, h, k, cmax, stepstr, bigtext, fname)
```

**NCL : Number of contouring levels.**

- integer gx, gy, iundef, ncl
- real rundef, cmin
- parameter (ncl=20, iundef=9999)
- parameter (rundef=999.999)
- real zcl(2), wi(ncl), step
- integer lenar1(4), lenar2(3), kol(ncl)
- character txtar1(4)*l, txtar2(3)*5
- character*55 stepstr
- character*55 bigtext
- character*55 fname
- real grid(-gx:gx, -gy:gy), cmax, h, k, xoff, yoff, height, tpx, tpy
- real xmin, xmax, ymin, ymax, zmin, zmax, dbl, xsi, ysi, xsise, ysize, xm
- integer ntick
- integer npx, npy

**Contour line widths and colors.**

- data wi / ncl*0.2 /
- data kol / ncl*1 /
- data cmin / 1e-3 /

**Character string lengths of axis texts and axis text strings.** No axis texts are to be plotted.
APPENDIX A. PROGRAM CODE

```
data lenarl /4=0/
data txtarl /4=' '/

C Spacing between axis labels, ticks between labels

data ntick /9/

C Character string lengths and character strings
C for color scale.

data lenar2 /5, 5, 0/
data txtax2 /'below', 'above', ' '/

print*, stepstr, ' ', bigtext, ' ', fname

print*, 'no. of x coords: ', gx*2+1
print*, 'no. of y coords: ', gy*2+1
print*, 'size of grid: ', (gx*2+1)*(gy*2+1), ' points.'

C User coordinate limits.

xmin=-gx*h
xmax=gx*h
ymin=-gy*k
ymax=gy*k
zmin=grid(gx, gy)
zmax=cmax
dbl=(xmax-xmin)/10.0

* print*, 'cmax=', cmax
* print*, 'cmin=', cmin
* print*, 'xmin=', xmin
* print*, 'ymax=', ymax

C Smallest contouring level and distance between levels.

* data cmin, step / 0.02, 0.02 /

C N.B. cmin can be found in the parameter list;
C if cmin is set to 0.0, a floating exception occurs in GCNRES();

* cmax = 0.0
* cmin = -0.5
step=(cmax-cmin)/real(ncl)

print*, 'contour step=', step

C Open output device;

call ropen

C Set limits and viewport.
```
APPENDIX A. PROGRAM CODE

* call grpsiz(xsi, ysi)
call rqarea(xsi, ysi, npx, npx)
xm = min(xsi, ysi)
xsiz = 0.80*xm
yvize = (ymax— ym in)/(xmax— xmin) *xsize 100
xoff = 0.5*(xsi— xsiz)
yoff = 0.5*(ysi— ysize)
call glimit(xmin, xmax, ymin, ymax, zm in, zm ax)
call gvport(xoff, yoff, xsize, ysize)

C Set contour levels.
   zcl(1) = cm in
   zcl(2) = step
* call rela ss(zcl, ncl, -5)
* call rela ss(zcl, ncl, -5)
call rclsts(zcl(1), zcl(2), ncl)

C Select a grey scale suitable for the plots in this manual.
* call rshade(-4, 0)
call rshacs(-4)

C Do the shaded contour plot.
print*, 'displaying data...'
call gcn2s(grid, gx+2+l, gy+2+1)
print*, 'adding some isolines...'

C Contour lines are plotted on top of the shaded map
C by GCNR2V. Set contour line widths and colors.
call gcswi(wi, ncl)
call gcconco(kol, ncl)
height = 0.01*min(xsiz, ysize)

C Set contour line annotation attributes and text font.

C Character height
C number of decimals
C Distance between labels
C Move and overlay
C options active
C
   call gcona(height, 2, -0.8*xsize, 3)

C Have a box plotted around each contour line label
C
C Color of box
C Turn plotting of box edge on
C
   call gconab(0, 1)
APPENDIX A. PROGRAM CODE

call gcnr2v(grid, gx*2+1, gy*2+1)
pprint*, 'contour sorted'

C Plot four axes. To make room for a
C color scale, the Y-axis to the right is
C plotted without numeric labels.

        call gscale
        call raxtef(4, 'swim', 1)
        call raxifo(2, 0, iundef, iundef)
        call raxbu(iundef, rundef, rundef, dbl)
        call raxsti(ntick)
        call raxdis(3, 1, iundef)

C Plot bottom X-axis and left Y-axis.

        call raxis2(ymin, xmin, height, lenarl, txtarl)

C Plot top X-axis.

        call raxis(1, ymax, height, 2)

C Turn numeric axis labels off, and plot right Y-axis.

        call raxdis(4, 0, iundef)
        call raxis(2, xmax, height, 2)

        print*, 'done plotting axes'

C Plot a color scale.

        call rtxfon('swim', 1)
        call gclopt(lenar2, txtar2, 1.75*height, 2, 0.0, 1)
        call gcoscl(xmax+0.05*(xmax—xmin), ymin)
        print*, 'done plotting scale'

C Give the plot a title.

        tpy = ymin—0.05*(ymax—ymin)
        tpx = 0.5*(xmin+xmax)
        call rtixus(1, 3)
        call rtixhe(0.0)
        call rtx(-1, bigtext, tpx, tpy)
        call rtx(-1, 'hello, world.', 0.0, 0.0)
        call rtx(-1, stepstr, tpx, ymax-h0.075* (ymax-ymin))
        call rtxbol(0.125)
        call rtxang(90.0)
        call rtx(-2, fname, -0.6*(xmax-xmin), 0.0)

        print*, ' ' 200
        call rclose
A.2 Code implementing problem posed in Part III

A.2.1 2D explicit solver

Header (sv/sv.h)

/* header file for Saint-Venant equations */

/* to test sv-restore as a standalone */
/* #define TEST */
/* #define DEBUG */

#define XP 128
#define YP XP
#define PI 3.14159265358979
/* #define PI 3.141592654 */

#define TAU 0.0001
#define TIME 1.
#define PICS 100
#define VARIANCE 0.1 /* allowed invariant stray (10%) */

#define STEPS int(TIME/TAU)

 /* for the grid to be 2*PI*2, these must be fixed */
#define H (PI/real(XP))
#define K (PI/real(YP))

/* init func */
/* DONT USE C, it's a comment character */

#define GRISHA

#elif defined WILF

#define A 1.0
#define B 1.0
#define D 10.0

#define INIT_VX (A * sin(x + 2.0*y) * exp((-x**2 - y**2)/2.0))
#define INIT_VY (B * sin(3.0*x + 4.0*y) * exp((-x**2 - y**2)/2.0))
#define INIT_PHI (D + 0.1 * cos((x**2 + y**2)/4))

#elif defined(GRISHA)

#define INIT_VX 0.0
#define INIT_VY 0.0
#define INIT_PHI (20 + 0.1 * cos(10.0*x + 10.0*y) * exp(-x**2 - y**2))
APPENDIX A. PROGRAM CODE

#else
#define A  1.0
#define B  1.0
#define D  10.0

#define INIT_VX (A * sin(x + 2.0*y) + exp((-x**2 - y**2)/2.0))
#define INIT_VY (B * sin(3.0*x + 4.0*y) + exp((-x**2 - y**2)/2.0))
#define INIT_PHI (D + 0.1*cos((x**2 + y**2)/4))

#endif

#define F (2.0*2.0*F/(24*3600*365.256/366.25))

/* fixed */
#define LEVS 0:1
#define DIMS -X:XP-1, -Y:YP-1, LEVS
#define ITERATIONS 20

Time evolution code (sv/sv-main2.F)

C Author: Wilf (eeplgw)
C creation date: 12-5-95
C last revision date: %G%
C revision from:
C revision: %2%%M% No.: %I%
C Notes:

C objects:
C libraries:

#include "sv.h"

C Set up some file descriptor ids

#define VXFILE 10
#define VYFILE 11
#define DELTAFILE 12
#define OMEGAFILE 13
#define PHIFILE 14
#define INVARFILE 15
#define PARAMFILE 16
#define RESTARTFILE 17

C We have either V(x, y) or omega and delta. We also have phi and C f is a constant.

C Two black boxes are required: (1) obtain omega/delta from V
C (2) obtain V form omega/delta

C (1) is simple, taking care with periodic boundaries
APPENDIX A. PROGRAM CODE

C (2) requires a poisson solver

C A two step, O(time^2), O(space^2) method is employed which needs
C a single step method to get started. We are using an O(time)
C method for this purpose. This should be replaced by a suitable
C O(time^2) method at a later date.

C Storage requirements are for omega, delta, V(x, y) at three time levels.
C It may be possible to replace t-tau level with t+tau level, thus
C requiring only two time levels of storage for each variable.

C All functions are periodic and real.

C In the first instant, no iteration is done on the non-linearity
C in equation 3, but a pseudo C-N method is used whereby delta*phi
C is replaced with delta(t)*phi(t)/2 + delta(t+tau)*phi(t)/2

C A requirement of the above is that eq 2 is solved before eq 3.

C Once all three eqs have been solved, the results are tested for
C energy and enstrophy conservation. It is expected that enstrophy
C will _not_ be conserved due to the nature of the truncated system.

C

program sv_main

---

C The last element of these arrays is the relative time level

character*50 fname_in, fname_out
real vx(DIMS)
real vy(DIMS)
real omega(DIMS)
real delta(DIMS)
real phi(DIMS)
real energy0, energy, enstrophy0, enstrophy
integer n
logical restart, process_args
integer begin

C FFT scratch space

real rs(2*(2*XP)+15)
real cs(4*(2*XP)+15)

common /n/ n
common /v/ vx, vy
common /delta/ delta, /omega/ omega, /phi/ phi

C Process command line and decide if we are restarting

restart = process_args(fname_in, fname_out)

C Init FFT scratch space, only need to do this once
call show_params()
call init_write('FFT scratch space')
call rffti(2*XP, rs)
call cffti(2*XP, cs)

C Initialise at time=0

if (restart) then
    call restarting(fname_in)
call open_out_files(fname_out)
call produce_invars(0, energy0, enstrophy0)
    begin = 1
else
    call open_out_files(fname_out)
call init()
call produce_invars(0, energy0, enstrophy0)
call init_leapfrog(rs, cs)
    begin = 2
endif

call save_params(fname_out, PARAM_FILE)

C Get going proper

call set_constants(TAU)

do n = begin, STEPS, 2

C even leapfrog steps, eg. 0, 1 -> 2

call set_method(0, 1, 0)
call step(n, rs, cs)

call produce_invars(n, energy, enstrophy)
call check_invars(energy0, energy, enstrophy0)

C odd leapfrog steps, eg. 1, 2 -> 3

call set_method(1, 0, 1)
call step(n+1, rs, cs)

call produce_invars(n+1, energy, enstrophy)
call check_invars(energy0, energy, enstrophy0)
enddo

call close_bfiles()
close(INVAR_FILE)
call save_data(fname_out, RESTART_FILE)
end

subroutine produce_invars(step, energy, enstrophy)
real energy, enstrophy
real energyf, enstf, trapez
external energyf, enstf
integer step

energy = trapez(energyf, H, K, XP, YP)
enstrophy = trapez(enstf, H, K, XP, YP)
call write_invars(step, energy, enstrophy, INVAR_FILE)

end

subroutine restarting(fname_in)
character(*) fname_in
integer next, delay

common /iterate/ next, delay

call load_params(fname_in, PARAM_FILE)
call load_data(fname_in, RESTART_FILE)
call compute_phi_mean()
delay = STEPS/PICS
next = delay

end

subroutine open_out_files(fname_out)
character*50 fname_out

call init_write('Output files')
call write_tinit('time_invars.' // fname_out, INVAR_FILE)
call write_binit('data_vx.' // fname_out, VX_FILE)
call write_binit('data_vy.' // fname_out, VY_FILE)
call write_binit('data_delta.' // fname_out, DELTA_FILE)
call write_binit('data_omega.' // fname_out, OMEGA_FILE)
call write_binit('data_phi.' // fname_out, PHI_FILE)

end

subroutine close_bfiles()

close(VX_FILE)
close(VY_FILE)
close(DELTA_FILE)
close(OMEGA_FILE)
close(PHI_FILE)
end

subroutine init()

C
------

real offset
integer next, delay

common /iterate/ next, delay
common /offset/ offset

data offset /0.0/

call init_write('Initial conditions')

delay = STEPS/PICS
next = 0

call set_method(0, 0, 0)
call init_write('v')
call init_v()
call init_write('omega')
call init_omega()
call init_write('delta')
call init_delta()
call init_write('phi')
call init_phi()
call compute_phi_mean()
call do_output(0)

end

subroutine init_leapfrog(rs, cs)

C
-------

real energy, enstrophy
real rs(1), cs(1)

C Kick start the leap frog method

C a single $O(t)$ time step, $\theta > 1$

call init_write('Leap frog method')
call set_method(0, 0, 1)
call set_constants(TAU/2.0)
call step(1, rs, cs)

call produce_invars(1, energy, enstrophy)

end

subroutine show_params()

C
------
APPENDIX A. PROGRAM CODE

print*, 'Info: x, y', XP, YP
print*, 'Info: data dumps:', PICS+1
print*, 'Info: h, k:', H, K
print*, 'Info: Constant f:', F
end

subroutine init_v()
C

C
C
C C

do j = -YP, YP-1
    do i = -XP, XP-1
        phi(i, j) = init_phifunc(i, j)
    enddo
enddo

real function init_phifunc(i, j)
C

end subroutine init_v()
init_v

end subroutine init_delta()
init_delta

end subroutine init_omega()
init_omega

end subroutine init_phi()
init_phi

real phi(DIMS)
integer i, j
real init_phifunc
integer old, now, new

common /phi/ phi
common /method/ old, now, new

do j = -YP, YP-1
    do i = -XP, XP-1
        phi(i, j, old) = init_phifunc(i, j)
    enddo
enddo

end subroutine init_phi()
init_phifunc

integer i, j
real x, y
\begin{verbatim}
x = real(i) * H
y = real(j) * K

init_phifunc = INIT_PHI

end

subroutine step(n, rs, cs)

real omega(DIMS)
real delta(DIMS)
real rs(1), cs(1)
integer old, now, new
integer n

common /method/ old, now, new
common /omega/ omega, /delta/, delta

C the procedure to complete one time step

call dealias(omega(-XP, -YP, now), rs, cs)
call step_omega()
call dealias(delta(-XP, -YP, now), rs, cs)
call step_delta()
call step_phi()
call compute_phi-mean()
call restore_v(rs, cs)
call do_output(n)

end

subroutine dealias(f, rs, cs)

C Remove possible aliasing frequencies from the surface f

real f(DIMS)
complex g(-XP:XP-1, -YP:YP-1)
real rs(1), cs(1)

call fftf(f, g, rs, cs, XP, YP)
call filter_highf(g)
call fftb(f, g, rs, cs, XP, YP)

end

subroutine do_output(n)

C Check if its time to output a graph, if so, increment next

real vx(DIMS)
\end{verbatim}
APPENDIX A. PROGRAM CODE

```
real vy(DIMS)
real omega(DIMS)
real delta(DIMS)
real phi(DIMS)
integer n
integer next, delay
integer old, now, new

common /v/ vx, vy
common /delta/ delta, /omega/, omega, /phi/, phi
common /iterate/ next, delay
common /method/ old, now, new

C could be .eq. but for sanity, use .ge.

if (n.ge.next) then
  print*, 'Committing step ', n, ' to disk...'  
call write_array('vx*', vx(-XP, -YP, new), VX_FILE, n)
call write_array('vy', vy(-XP, -YP, new), VY_FILE, n)
call write_array('delta', delta(-XP, -YP, new), DELTA_FILE, n)
call write_array('omega', omega(-XP, -YP, new), OMEGA_FILE, n)
call write_array('phi', phi(-XP, -YP, new), PHI_FILE, n)
  next = next + delay
endif
end

real function energyf(i, j)
energyf
C energy invariant function

real phi(DIMS)
real vx(DIMS)
real vy(DIMS)
real phi_mean
integer i, j
integer old, now, new

common /phi/ phi
common /v/ vx, vy
common /method/ old, now, new
common /phi_mean/ phi_mean

energyf = phi(i, j, new) + phi_mean * 
  (vx(i, j, new)**2 + vy(i, j, new)**2)
end

real function enstf(i, j)
C enstrophy invariant function
```

```
real omega(DIMS)
real phi(DIMS)
integer i, j
integer old, now, new

common /omega/ omega, /phi/ phi
common /method/ old, now, new

enstf = omega(i, j, new)**2 / phi(i, j, new)

end

subroutine compute_phi_mean()

C compute the mean of phi and store it in the common area phi_mean

real phi(DIMS)
real phi_mean
integer i, j
integer old, now, new

common /phi/ phi
common /phi_mean/ phi_mean
common /method/ old, now, new

phi_mean = 0.0

C To avoid overflowing phi_mean

do i = -XP, XP-1
  do j = -YP, YP-1
    phi_mean = phi_mean + phi(i, j, new)/real(2*XP*2*YP)
  enddo
endo
endo

end

subroutine step_omega()

C

real omega(DIMS)
real omegafunc
integer old, now, new
external omegafunc

common /omega/ omega
common /method/ old, now, new

call periodic(omega(-XP, -YP, new), omegafunc)

end

subroutine step_delta()
APPENDIX A. PROGRAM CODE

C

real delta(DIMS)
real deltafunc
integer old, now, new
external deltafunc

common /delta/ delta
common /method/ old, now, new

call periodic(delta(-XP, -YP, new), deltafunc)
end

subroutine step_phi()

real phi(DIMS)
real phifunc
integer old, now, new
external phifunc

common /phi/ phi
common /method/ old, now, new

call periodic(phi(-XP, -YP, new), phifunc)
end

real function omegafunc(i0, i, i2, j0, j, j2)

integer old, now, new
real omega.vx, omega.vy, ddx, ddy
real dt, dtdx, dtdy, dtdx2, dtdy2
real omega(DIMS)
integer i0, i, i2, j0, j, j2
real ddx, ddy
external omega.vx, omega.vy

common /omega/ omega
common /method/ old, now, new
common /constants/ dt, dtdx, dtdy, dtdx2, dtdy2

omegafunc = omega(i, j, old) -
+ ddx(omega.vx, i0, i2, j, now) -
+ ddy(omega.vy, i, j0, j2, now)
end

real function deltafunc(i0, i, i2, j0, j, j2)

integer old, now, new
real omega_vx, omega_vy, phi_v2
real dt, dtdx, dtdy, dtdx2, dtdy2
real ddx, ddy, ddx2, ddy2
real delta(DIMS)
integer i0, i2, j0, j
external omega_vx, omega_vy, phi_v2
common /delta/ delta
common /method/ old, now, new
common /constants/ dt, dtdx, dtdy, dtdx2, dtdy2

deltafunc = delta(i, j, old) +
+ ddx(omega_vy, i0, i2, j, now) -
+ ddy(omega_vx, i, j0, j2, now) -
+ ddx2(phi_v2, i0, i, i2, j, now) -
+ ddy2(phi_v2, i, j0, j, j2, now)
end

real function phifunc(i0, i, i2, j0, j, j2)

C The following functions are helper (sub)functions which make
C the function calculations more intuitive.

real function omega_vx(i, j, l)

C

real omega(DIMS)
real vx(DIMS)
APPENDIX A. PROGRAM CODE

```c
real vy(DIMS)
integer i, j

common /v/ vx, vy
common /omega/ omega

omega.vx = (omega(i, j, l) + F) * vx(i, j, l)

end

C

real function omegaVy(i, j, l)

real omega(DIMS)
real vx(DIMS)
real vy(DIMS)
integer i, j, l

common /v/ vx, vy
common /omega/ omega

omegaVy = (omega(i, j, l) + F) * vy(i, j, l)

end

C

real function phiVy2(i, j, l)

real phi(DIMS)
real vx(DIMS)
real vy(DIMS)
integer i, j

common /v/ vx, vy
common /phi/ phi

phiVy2 = phi(i, j, l) + (vx(i, j, l)**2 + vy(i, j, l)**2)/2.0

end

C

real function phif(i, j, l)

real phi(DIMS)
integer i, j, l

common /phi/ phi

phif = phi(i, j, l)

end
```
APPENDIX A. PROGRAM CODE

A.2.2 2D FFT Poisson solver

Velocity restoration (sv/sv-restore1.F)

C  Author: Wilf (cep3gw)
C  creation date: 18-1-95
C  last revision date: %G%
C  revision from: poisson1.F
C  revision: %Z%%M% No.: %I%
C  Notes: New, not using cyclic_r()

C Code contains (1) restore_v() from omega, delta
C (2) produce_omega() from vz, vy
C (3) produce_delta() from vz, vy

C objects:
C libraries: -lfft

#include "sv.h"

#define FILE_Space 11
#define FILE_SPECT 12
#define FILE_DATAR 13
#define FILE_DATAC 14

define init_vxfunc(i, j)
in i, j
real x, y

x = real(i) * H
y = real(j) * K

*init_vxfunc = A * real(cexp(cmplx(0.0, x + 2.0*y)))
init_vxfunc = INIT_VX
end

real function init_vyfunc(i, j)
in i, j
real x, y

x = real(i) * H
y = real(j) * K

*init_vyfunc = B * real(cexp(cmplx(0.0, 3.0*x + 4.0*y)))
init_vyfunc = INIT_VY
end
subroutine produce_v()

C     Produce the original space function.

C real vx(DIMS)
real vy(DIMS)
integer i, j
real init_vxfunc, init_vyfunc
integer old, now, new
common /method/ old, now, new
common /v/ vx, vy

do j = -YP, YP-1
  do i = -XP, XP-1
    vx(i, j, old) = init_vxfunc(i, j)
    vy(i, j, old) = init_vyfunc(i, j)
  enddo
enddo

end

subroutine restore_v(rs, cs)

real vx(DIMS)
real vy(DIMS)
real omega(DIMS)
real delta(DIMS)
real g(-XP:XP-1, -YP:YP-1)
complex f(-XP:XP-1, -YP:YP-1)
complex psi(-XP:XP-1, -YP:YP-1)
integer old, now, new
real r(1), cs(1)

C Do we really need both f and psi at the same time?

common /delta/ delta, /omega/ omega
common /v/ vx, vy
common /method/ old, now, new

equivalence (f, psi)

C First of all get v(x) back.

call produce_gx(g)
call fft(f, g, rs, cs, XP, YP)
call write_complex_file(f, FILE_SPECT, 0)
call solve_for_psi(f)
call write_complex_file(f, FILE_SPECT, 1)
call filter_high(f)
call circular_trunc(f)
call write_complex_file(f, FILE.SPECT, 2)

C Pass the 3d array as 2d array at level 'new'

call fftb(vx(-XP, -YP, new), psi, rs, cs, XP, YP)

C Now, get v(y) back, reusing some of the array space.

call produce_gy(g)
call fftf(g, f, rs, cs, XP, YP)
call write_complex_file(f, FILE.SPECT, 3)
call solve_for_psi(f)
call write_complex_file(f, FILE.SPECT, 4)
  * call filter_high(f)
call circular_trunc(f)
call write_complex_file(f, FILE.SPECT, 5)

C Pass the 3d array as 2d array at level 'new'

call fftb(vy(-XP, -YP, new), psi, rs, cs, XP, YP)

g

subroutine solve_for_psi(f)

integer i, j
complex f(-XP:XP-1, -YP:YP-1)

C don't play with the central point;
C Standard levels of optimization should bring the constant if() out
C of the loop.

C As we are filtering out high f components, don't bother with
C those grid points.

if 0
  do j = -YP/2, YP/2-1
    do i = 0, XP/2-1
      if (i.ne.0 .or. j.ne.0) then
        f(i, j) = f(i, j) / -(real(i)**2 + real(j)**2)
      endif
    enddo
  enddo
endif

do j = -YP+1, YP-2
  do i = 0, XP-2
    if (i.ne.0 .or. j.ne.0) then
      f(i, j) = f(i, j) / -(real(i)**2 + real(j)**2)
    endif
  enddo

C do the full (almost) translation: leave the edges for truncation.

  do j = -YP+1, YP-2
    do i = 0, XP-2
      if (i.ne.0 .or. j.ne.0) then
        f(i, j) = f(i, j) / -(real(i)**2 + real(j)**2)
      endif
    enddo
  enddo
APPENDIX A. PROGRAM CODE

end do 
end do 
end 

subroutine circular_trunc(f)
    
    C Strip out the top most frequency which has severely restricted circular
    C movement and is nonsense.

    integer i, j
    complex f(-XP:XP-1, -YP:YP-1)

    C top and bottom half edges
    do i = 0, XP-2
       f(i, -YP) = 0.0
       f(i, YP-1) = 0.0
    enddo

    C The right hand edge
    do j = -YP, YP-1
       f(XP-1, j) = 0.0
    enddo

end subroutine filter_highf
    
    integer i, j
    complex f(-XP:XP-1, -YP:YP-1)

    do j = -YP/2, YP/2-1
       do i = XP/2, XP-1
          f(i, j) = 0.0
       enddo
    do i = YP/2, YP-1
       do j = 0, XP-1
          f(i, j) = 0.0
          f(i, -j-1) = 0.0
       enddo
    enddo

    C produce_gx() and produce_gy() are essentially the same, except
    C one calls gfunc(f), the other, gfunc(f). Also these functions are
    C virtually identical, except one takes a difference, the other a sum.
subroutine produce_gx(g)

    real g(1), gxfunc
    external gxfunc

    call periodic(g, gxfunc)

end

subroutine produce_gy(g)

    real g(1), gyfunc
    external gyfunc

    call periodic(g, gyfunc)

end

subroutine produce_omega()

    real omega(DIMS)
    real init_omegafunc
    integer old, now, new
    external init_omegafunc

    common /method/ old, now, new
    common /omega/ omega

    call periodic(omega(-XP, -YP, old), init_omegafunc)

end

subroutine produce_delta()

    real delta(DIMS)
    real init_deltafunc
    external init_deltafunc
    integer old, now, new

    common /method/ old, now, new
    common /delta/ delta

    call periodic(delta(-XP, -YP, old), init_deltafunc)

end
APPENDIX A. PROGRAM CODE

real function gxfunc(i0, i1, i2, j0, j1, j2)  

real omega(DIMS)  
real delta(DIMS)  
integer i0, i1, i2, j0, j1, j2  
integer old, now, new  
real dx, dy  

C $i[012]$ is traditionally $i-1$, $i$, $i+1$ and similarly for $j$

common /method/ old, now, new  
common /omega/ omega  
common /delta/ delta

* $gxfunc = -dx(delta, i0, i2, j1, new) - dy(omega, i1, j0, j2, new))$  
$gxfunc = dx(delta, i0, i2, j1, new) - dy(omega, i1, j0, j2, new)$

end

real function gyfunc(i0, i1, i2, j0, j1, j2)  

real omega(DIMS)  
real delta(DIMS)  
integer i0, i1, i2, j0, j1, j2  
integer old, now, new  
real dx, dy  

C $i[012]$ is traditionally $i-1$, $i$, $i+1$ and similarly for $j$

common /method/ old, now, new  
common /omega/ omega  
common /delta/ delta

* $gyfunc = -dy(delta, i1, j0, j2, new) + dx(omega, i0, i2, j1, new))$  
$gyfunc = dy(delta, i1, j0, j2, new) + dx(omega, i0, i2, j1, new)$

end

real function init_omegafunc(i0, i1, i2, j0, j1, j2)  

real vx(DIMS)  
real vy(DIMS)  
integer i0, i1, i2, j0, j1, j2  
integer old, now, new  
real dx, dy  

C $i[012]$ is traditionally $i-1$, $i$, $i+1$ and similarly for $j$

common /method/ old, now, new
APPENDIX A. PROGRAM CODE

common /v/ vx, vy

init_omegafunc = dx(vy, i0, i2, j1, old) - dy(vx, i1, j0, j2, old)
end

real function init_deltafunc(i0, i1, i2, j0, j1, j2)

real vx(DIMS)
real vy(DIMS)
ingeger i0, i1, i2, j0, j1, j2
integer old, now, new
real dx, dy

if[012] is traditionally i-1, i, i+1 and similarly for j

common /method/ old, now, new
common /v/ vx, vy

init_deltafunc = dx(vx, i0, i2, j1, old) + dy(vy, i1, j0, j2, old)
end

C subroutine write_complex_file(matrix, unit, n)

integer unit, n
complex matrix(-XP:XP-1, -YP:YP-1)
end

real function dx(array, i0, i2, j, l)

integer i0, i2, j, l
real array(DIMS)

dx = (array(i2, j, l) - array(i0, j, l)) / (2.0*H)
end

real function dy(array, i, j0, j2, l)

integer i, j0, j2, l
real array(DIMS)

dy = (array(i, j2, l) - array(i, j0, l)) / (2.0*K)
end
APPENDIX A. PROGRAM CODE

2D to 1D FFT translator and post-processors (sv científico/SF1lib1.F)

Author: Wilf (eep2gw)
creation date: 18-1-95
last revision date: %G%
revision from: fft1.F
revision: %Z%%M% No.: %I%

Notes: 2D transform of real data using [rcjfftfifb] routines.
objects: debugged at 5-4-95
libraries: -lfft

/* #define DEBUG */
#define MAX 255

subroutine fftf(array, carry, wsave, cwsave, xp, yp)
----------------------------------------------------------
integer xp, yp
real array(-xp:xp -1, -yp:yp -1), wsave(1), cwsave(1)
complex carry(-xp:xp -1, -yp:yp -1)

C Do 1D FFT in x;
call message('FFT in x')
call fftf_in_x(array, carry, wsave, xp, yp)

C Do 1D FFT in y to complete 2D FFT;
call message('FFT in y')
call fftf_in_y(carry, cwsave, xp, yp)

C Sort the spectral image;
call message('Shifting data')
call fft_shift_in_y(carry, xp, yp)
end

subroutine fftb(array, carry, wsave, cwsave, xp, yp)
----------------------------------------------------------
integer xp, yp
real array(-xp:xp -1, -yp:yp -1), wsave(1), cwsave(1)
complex carry(-xp:xp -1, -yp:yp -1)

C unsort the spectral image;
call message('Shifting data')
call fft_shift_in_y(carry, xp, yp)

C Undo the FFT in y;
call message('FFTb in y')
call fftb_in_y(carray, csave, xp, yp)

C Undo the FFT in y - reproduce original;
call message('FFTb in x')
call fftb_in_x(rarray, carray, csave, xp, yp)
end

subroutine fftf_in_x(rarray, carray, csave, xp, yp)
C -----------------------------------------------
C Take real array and do 1D transform in x
integer i, j, xp, yp
real rarray(-xp:xp-1, -yp:yp-1), rslice(-MAX:MAX-1), csave(1)
complex carray(-xp:xp-1, -yp:yp-1)
do j = -yp, yp-1
  do i = -xp, xp-1
    rslice(i) = rarray(i, j)
  enddo
C FORTRAN KLUDGE to get around MAX:MAX-1 problem:
C send [rslice as an array with offset beginning!
call rfft(2*xp, rslice(-xp), csave)
call rfft2cfft(rslice(-xp), carray(-xp, j), xp, yp)
enddo
end

subroutine fftb_in_x(rarray, carray, csave, xp, yp)
C -----------------------------------------------
C Undo FFT in x
integer i, j, xp, yp
real rarray(-xp:xp-1, -yp:yp-1), rslice(-MAX:MAX-1), csave(1)
complex carray(-xp:xp-1, -yp:yp-1)
do j = -yp, yp-1
  C see above for details of kludge;
call cfft2rfft(rslice(-xp), carray(-xp, j), xp, yp)
call rfftb(2*xp, rslice(-xp), csave)
APPENDIX A. PROGRAM CODE

do i = -xp, xp-1
   rarray(i, j) = rslice(i) / real(2*xp)
enddo

enddo
end

subroutine fftf_in_y(carray, cwsave, xp, yp)
C Take complex 1D transformed array and produce 2D transform

integer i, j, xp, yp
complex carray(-xp:xp-1, -yp:yp-1), cslice(-MAX:MAX-1)
real cwsave(1)

do i = -xp, xp-1
   do j = -yp, yp-1
      cslice(j) = carray(i, j)
   enddo
   call cfftf(2*yp, cslice(-yp), cwsave)
   do j = -yp, yp-1
      carray(i, j) = cslice(j)
   enddo
enddo
end

subroutine fftb_in_y(carray, cwsave, xp, yp)
C Take 2D transform and do an inverse FFT in y

integer i, j, xp, yp
complex carray(-xp:xp-1, -yp:yp-1), cslice(-MAX:MAX-1)
real cwsave(1)

do i = -xp, xp-1
   do j = -yp, yp-1
      cslice(j) = carray(i, j)
   enddo
   call cfftb(2*yp, cslice(-yp), cwsave)
   do j = -yp, yp-1
      carray(i, j) = cslice(j) / real(2*yp)
   enddo
enddo
end
APPENDIX A. PROGRAM CODE

enddo
enddo
end

subroutine ft_shift_in_y(carray, xp, yp)
integer i, j, xp, yp
complex carray(-xp:xp-1, -yp:yp-1), temp

do j = -yp, -1
  do i = -xp, xp-1
    temp = carray(i, j)
    carray(i, j) = carray(i, j+yp)
    carray(i, j+yp) = temp
  enddo
enddo
end

subroutine fft2cfft(rslice, cslice, xp, yp)
integer i, xp, yp
real rslice(-xp:xp-1)
complex cslice(-xp:xp-1)

cslice(0) = cmplx(rslice(xp-1), rslice(-xp))

do i = -xp+1, xp-2, 2
  cslice((i+xp+1)/2) = cmplx(rslice(i), rslice(i+1))
enddo
end

subroutine cfft2rfft(rslice, cslice, xp, yp)
integer i, xp, yp
real rslice(-xp:xp-1)
complex cslice(-xp:xp-1)

cslice(0) = cmplx(rslice(xp-1), rslice(-xp))

do i = -xp+1, xp-2, 2
  cslice((i+xp+1)/2) = cmplx(rslice(i), rslice(i+1))
enddo
end

subroutine rfft2cfft(rslice, cslice, xp, yp)
integer i, xp, yp
real rslice(-xp:xp-1)
complex cslice(-xp:xp-1)

cslice(0) = cmplx(rslice(xp-1), rslice(-xp))

do i = -xp+1, xp-2, 2
  cslice((i+xp+1)/2) = cmplx(rslice(i), rslice(i+1))
enddo
end
APPENDIX A. PROGRAM CODE

C and create a real array from the complex array.

integer i, xp, yp
real rslice(-xp:xp-1)
complex cslice(-xp:xp-1)

rslice(-xp) = imag(cslice(0))
rslice(xp-1) = real(cslice(0))

do i = -xp+1, xp-2, 2
rslice(i) = real(cslice((i+xp+1)/2))
rslice(i+1) = imag(cslice((i+xp+1)/2))
enddo

end subroutine message(string)

C -----------------------------------------
character string(*)

#ifdef DEBUG
print*, string
#endif

A.2.3 Miscellaneous solver routines

Finite difference operator functions (sv/sv-fde.F)

C Author: Wilf (sep8gw)
C creation date: 23-May-95
C revision from:
C Notes: FDE functions for sv
C objects:
C libraries:

C FDE functions:
C
C          ddx = d(func)/dx
C          ddy = d(func)/dy
C          ddx2 = d2(func)/dx2
C          ddy2 = d2(func)/dy2

C real function ddx(func, i0, i2, j, l)
C-----------------------------------------

integer i0, i2, j, l
real dt, dtdx, dtdy, dtdx2, dtdy2
real func
externa func

common /constants/ dt, dtdx, dtddy, dtdx2, dtdy2

\( ddx = dtdx \times (func(i2, j, 1) - func(i0, j, 1)) \)

end

real function ddy(func, i, j0, j2, 1) ddy

C

integer i, j0, j2, 1
real dt, dtdx, dtddy, dtdx2, dtddy2
real func
externa func

common /constants/ dt, dtdx, dtddy, dtdx2, dtddy2

\( ddy = dtddy \times (func(i, j2, 1) - func(i, j0, 1)) \)

end

real function ddx2(func, i0, i, i2, j, 1) ddx2

C

integer i0, i, i2, j, 1
real dt, dtdx, dtddy, dtdx2, dtddy2
real func
externa func

common /constants/ dt, dtdx, dtddy, dtdx2, dtddy2

\( ddx2 = dtdx2 \times (func(i2, j, 1) - 2.0 \times func(i, j, 1) + func(i0, j, 1)) \)

end

real function ddy2(func, i, j0, j, j2, 1) ddy2

C

integer i, j0, j, j2, 1
real dt, dtdx, dtddy, dtdx2, dtddy2
real func
externa func

common /constants/ dt, dtdx, dtddy, dtdx2, dtddy2

\( ddy2 = dtddy2 \times (func(i, j2, 1) - 2.0 \times func(i, j, 1) + func(i, j0, 1)) \)

end
Miscellaneous file I/O and helper functions (sv/sv-misc.F)

C Author: Wilf (eepbgw)
C creation date: 18-5-95
C last revision date: %G%
C revision from:
C revision: %Z%%M% No.: %I%
C Notes:
C objects:
C libraries:

#include "sv.h"
#define SIGS 3

subroutine write_binit(fname, unit)
character fname(*)
integer unit
open(unit, file=fname, status='new', form='unformatted')
write(unit) int(XP), int(YP)
C Open up the output files.
end

subroutine write_tinit(fname, unit)
character fname(*)
integer unit
open(unit, file=fname, status='new', form='formatted')
end

subroutine write.array(array, unit, step)
real array(-XP:XP-1, -YP:YP-1)
integer unit, step
write(unit) 'Step=', step
write(unit) array
end

subroutine read.array(array, unit, step, xp, yp)
end

write_binit
write_tinit
write_array
read_array
integer xp, yp
real array(-xp:xp-1, -yp:yp-1)
integer unit, step
character dummy*5

read(unit) dummy, step
read(unit) array

end

subroutine write_invars(n, energy, enstrophy, unit)
  real energy, enstrophy, phi_mean
  real energy0, enstrophy0
  integer n, unit
  real offset, time

  common /phi_mean/ phi_mean
  common /offset/ offset
  common /invariants/ energy0, enstrophy0

  time = real(n)*real(TAU) + offset

  write(unit, *) time, energy/energy0, enstrophy/enstrophy0, phi_mean
  print*, n, time, energy/energy0, enstrophy/enstrophy0, phi_mean
  call flush(unit)
end

subroutine init_write(string)
  character string*(*)
  print*, 'Initializing: ', string
end

subroutine error(string)
  character string*(*)
  print*, 'Warning: ', string
end

subroutine fatal(string)
  character string*(*)
  print*, 'Fatal: ', string
  call abort()
end
subroutine set_method(s_old, s_now, s_new)
C
Integer s_old, old, s_now, now, s_new, new
Common /method/ old, now, new
old = s_old
now = s_now
new = s_new
end

subroutine set-constants(tau)
C
Real tau, dt, dtdx, dtdy, dtdx2, dtdy2
Common /constants/ dt, dtdx, dtdy, dtdx2, dtdy2
dt = tau
dttx = tau / H
dttx = tau / K
dttx2 = (2.0*tau) / H**2
dttx2 = (2.0*tau) / K**2
end

subroutine check_invars(n, energy, enstrophy)
C
Real energy0, energy, enstrophy0, enstrophy
Real egydiff, esydiff, egylimit, esylimit
Integer next, delay
Integer n
Common /iterate/ next, delay
Common /invariants/ energy0, enstrophy0
C Ensure that the invarients are still invariant.
egydif = abs(energy0 - energy)
esydif = abs(enstrophy0 - enstrophy)
egylimit = energy0 * VARIANCE
esylimit = enstrophy0 * VARIANCE
if (egydiff .gt. egylimit) or.
    esydif .gt. esylimit) then
C Force a data dump for later inspection
next = n
call do_output(n)

call fatal('Invariant problem')
endif
end

subroutine periodic(f, func)
C-----------------------------
C This function is as general as possible. You pass a 2D array and
C the address of the function that will satisfy the array. periodic
C arranges to call the function in a way conducent with periodic
C boundaries.

real f(-XP:XP-1, -YP:YP-1, 1)
real func
external func
integer i, j

doi = -XP+1, XP-2
C The main central box excluding x-y boundaries;
do j = -YP+1, YP-2
f(i, j, 1) = func(i-1, i, i+1, j-1, j, j+1)
enddo
C The y boundaries, excluding the four edge points;
f(i, -YP, 1) = func(i-1, i, i+1, YP-1, -YP, -YP+1)
f(i, YP-1, 1) = func(i-1, i, i+1, YP-2, YP-1, -YP)
enddo
C The x boundaries excluding the four edge points;
do j = -YP+1, YP-2
f(-XP, j, 1) = func(XP-1, -XP, -XP+1, j-1, j, j+1)
f(XP-1, j, 1) = func(XP-2, XP-1, -XP, j-1, j, j+1)
enddo
C The four edge points;
f(-XP, -YP, 1) = func(XP-1, -XP, -XP+1, YP-1, -YP, -YP+1)
f(XP-1, -YP, 1) = func(XP-2, XP-1, -XP, YP-1, -YP, -YP+1)
f(-XP, YP-1, 1) = func(XP-1, -XP, -XP+1, YP-2, YP-1, -YP)
f(XP-1, YP-1, 1) = func(XP-2, XP-1, -XP, YP-2, YP-1, -YP)
end

logical function process_args(fname_in, fname_out)
APPENDIX A. PROGRAM CODE

```
C

character(*) fname_in, fname_out
integer argc, large

argc = argc()

if (argc .gt. 2 .or. argc .lt. 1) then
  call error('bad command line arguments')
  stop
else if (argc .eq. 2) then
  call getarg(1, fname_in)
  call getarg(2, fname_out)
  process_args = .true.
else
  call getarg(1, fname_out)
  process_args = .false.
endif

end

subroutine load_params(fname, unit)

character*50 fname
integer unit
integer xp, yp, steps, pics
real h, k, tau

open(unit, file='params.' // fname,
     status='old', form='formatted')
read(unit, *) xp, yp, steps, pics
read(unit, *) h, k, tau

close(unit)

if (XP .ne. xp .or.
    YP .ne. yp) then
  print*, 'in: ', xp, yp, h, k
  print*, 'out: ', XP, YP, H, K
  call fatal('mismatching parameters')
endif

end

subroutine save_params(fname, unit)

character*50 fname
integer unit

open(unit, file='params.' // fname,
     status='new', form='formatted')
```
write(unit, *) XP, YP, STEPS, PICS
write(unit, *) H, K, TAU
write(unit, *) 'INIT-VX'
write(unit, *) 'INIT-VY'
write(unit, *) 'INIT-PHI'
close(unit)
end

subroutine load_data(fname, unit)
character*50 fname
integer unit
real vx(DIMS)
real vy(DIMS)
real omega(DIMS)
real delta(DIMS)
real phi(DIMS)
real offset
integer old, now, new
common /method/ old, now, new
common /omega/ omega, /delta/ delta, /phi/ phi, /v/ vx, vy
common /offset/ offset
call init_write('Input files ')
call set_method(l, 1, 0)
open(unit, file='restart.'// fname, status='old',
    + form='unformatted')
read(unit) offset
call read_in(vx(-XP, -YP, old), unit)
call read_in(vx(-XP, -YP, new), unit)
call read_in(vy(-XP, -YP, old), unit)
call read_in(vy(-XP, -YP, new), unit)
call read_in(omega(-XP, -YP, old), unit)
call read_in(omega(-XP, -YP, new), unit)
call read_in(delta(-XP, -YP, old), unit)
call read_in(delta(-XP, -YP, new), unit)
call read_in(phi(-XP, -YP, old), unit)
call read_in(phi(-XP, -YP, new), unit)
close(unit)
end

subroutine read_in(f, unit)
C
C read in a 2D array from the unit
real f(-XP:XP-1, -YP:YP-1)
integer unit

read(unit) f
end

subroutine write_out(f, unit)
------------------
C wrote out a 2D array to the unit
real f(-XP:XP-1, -YP:YP-1)
integer unit

write(unit) f
end

subroutine save_data(fname, unit)
------------------
C
character*50 fname
integer unit
real vx(DIMS)
real vy(DIMS)
real omega(DIMS)
real delta(DIMS)
real phi(DIMS)
integer n
integer old, now, new
real offset

common /n/ n
common /method/ old, now, new
common /omega/ omega, /delta/ delta, /phi/ phi, /v/ vx, vy
common /offset/ offset

C Adjustment if doing even leapfrog
if (now .eq. 1) n = n - 1

call system('/bin/rm -f restart.' // fname)

open(unit, file='restart.' // fname, status='new',
+ form='unformatted')

write(unit) real(n)*TAU + offset
call write_out(vx(-XP, -YP, old), unit)
call write_out(vy(-XP, -YP, now), unit)
call write_out(vy(-XP, -YP, old), unit)
call write_out(vy(-XP, -YP, now), unit)
call write_out(omega(-XP, -YP, old), unit)
call write_out(omega(-XP, -YP, now), unit)
call write_out(delta(-XP, -YP, old), unit)
call write_out(delta(-XP, -YP, now), unit)
call write_out(phi(-XP, -YP, old), unit)
call write_out(phi(-XP, -YP, now), unit)

close(unit)
end

subroutine catch_signals(handler)

integer handler
external handler

C How many signals are we catching?

integer i, signal, sig(SIGS), old_handler(SIGS)

common /signals/ old_handler

C What signals are we catching?

data sig /1, 2, 15/

do i = 1, SIGS
   old_handler(i) = signal(sig(i), handler, -1)
enddo
end

subroutine reset_signals()

C

C How many signals are we catching?

integer i, signal, dummy, sig(SIGS), old_handler(SIGS)

common /signals/ old_handler

C What signals are we catching?

data sig /1, 2, 15/

do i = 1, SIGS
   dummy = signal(sig(i), 0, old_handler(i))
enddo
end
2D quadrature using trapezoid rule (sv/trapez.f)

Author: Wilf (eep2gw)
creation date: 9-3-93 (9/9/93)
last revision date: %G%
revision from: simpson.f
revision: %Z% %M% No.: %1%

Notes: The ever popular:

objects:
libraries:

can only be used on meshes with odd No. of intervals: 2*xp, 2*yp

func is a function that expects i and j and returns a value dependent on these integers.

real function trapez(func, h, k, xp, yp)
     trapez
     integer xp, yp, j
     real h, k, ix, trapez
     real func
     external func

xp is simply passed through to ix unmodified.
All references to yp must be checked for out-of-bounds
trapez = ix(func, h, xp, -yp) + ix(func, h, xp, yp-1)
do j = -yp+1, yp-2
   trapez = trapez + 2.0*ix(func, h, xp, j)
enddo
trapez = trapez*k/2.0
end

real function ix(func, h, xp, j)
     ix
     integer i, j, xp
     real h, func
     external func

ix = func(-xp, j) + func(xp-1, j)
do i = -xp+1, xp-2
   ix = ix + 2.0*func(i, j)
enddo
\texttt{ix = ix*\texttt{h}/2.0}

\texttt{end}

---

\textbf{A.2.4 2D binary to 2D ASCII data converter}

译码器 for gnuplot (\textit{misc/readf77.c})

```c
/* code to read a FORTRAN77 binary file and produce an ascii file suitable for gnuplot. Output to stdout, messages to stderr */

#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

#define SPACING 4 /* Output 1 in every <SPACING> points */

/* read a byte from the binary file, discard */
static int read_size(FILE *fp)
{
    int size;
    return fread(&size, sizeof(int), (size_t) 1, fp);
}

/* Read the correct sized array, account for \( f[i, j] = C[j, i] \) and output ASCII suitable for gnuplot */
static void read_array(int xp, int yp, FILE *fp)
{
    int step_no, i, j;
    char step[6];
    float *array;

    fread(&step, sizeof(char), (size_t) 5, fp);
    step[5] = '\0';
    fread(&step_no, sizeof(int), (size_t) 1, fp);
    fprintf(stdout, "#"/\%s", step, step_no);
    fprintf(stderr, "#\%d\n", step_no);
    read_size(fp);

    array = (float*) malloc(2*xp * 2*yp * sizeof(float));
    fread(array, sizeof(float), (size_t) 2*xp * 2*yp, fp);

    for (i=0; i<2*xp; i+=SPACING) {
        for (j=0; j<2*yp; j+=SPACING)
            printf("%f \%d.0 \%d.0 \%d\n", i-xp, j-yp, (array+2*xp*j+i));
        putchar('\n');
    }

    putchar('\n');
```
read_size(fp);
}

int main(int argc, char *argv[])
{
    int xp, yp;
    char type;
    FILE *fp;

    if (argc != 2) {
        printf("\%s: supply one data filename as argument.\n", argv[0]);
        exit(1);
    }

    if ( !(fp = fopen(argv[1], "rb")) ) {
        fprintf(stderr, "\%s: cannot access ", argv[0]);
        perror(argv[1]);
        exit(1);
    }

    read_size(fp);
    fread(&type, sizeof (char), (size_t) 1, fp);
    fread(&xp, sizeof (int), (size_t) 1, fp);
    fread(&yp, sizeof (int), (size_t) 1, fp);

    if (type != '!') {
        fprintf(stderr, "foreign input file type\n");
        exit(1);
    }

    fprintf(stderr, "explicit array, type \%c: \%d:%d, \%d:%d\n", type,
            xp, xp-1, yp, yp-1);
    fprintf(stderr, "explicit array, type \%c: \%d:%d, \%d:%d\n", type,
            xp, xp-1, yp, yp-1);

    read_size(fp);

    while (read_size(fp) == 1)
        read_array(xp, yp, fp);

    if (feof(fp))
        fprintf(stderr, "error on file stream\n");

    fclose(fp);
    exit(0);
}