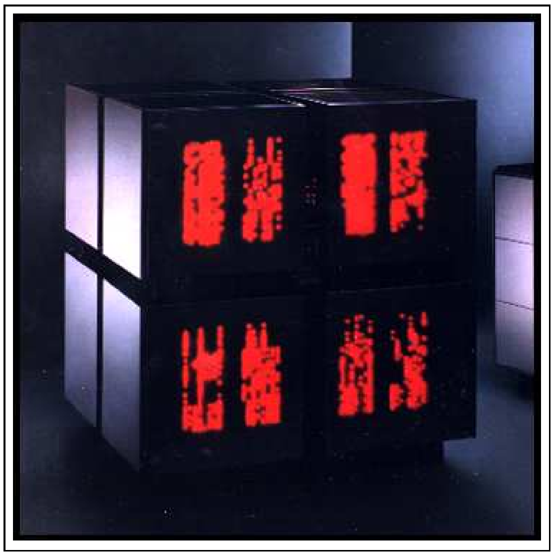

Spectral Methods

on

a Massively Parallel Computer.

M. Sc. Dissertation

by

Claus Bendtsen & Lars Kr. Lundin.
nmibtr@titan.uni-c.dk

July 30, 1992

Abstract

This document describes the spectral method for solving initial value problems of nonlinear partial differential equations. It presents an implementaion of this method on the Connection Machine, CM-200 and this is used to solve the one dimensional Korteweg de Vries equation, an extension of this equation, and the two dimensional Hasegawa-Mima equation, for a number of initial values. In the one-dimensional cases the conclusions are on the existance of solitons and the use of semiimplicit Runge Kutta methods. In the two-dimensional case conclusions regard an explicit Runge Kutta method.

Contents

1 Introduction. **1**

2 The Spectral Method. **2**
- 2.1 General Conventions . . . 2
- 2.2 Basic Properties of the Spectral Method used in one Dimension. . . 3
  - 2.2.1 The Discrete Fourier Transform. . . . 3
  - 2.2.2 The Continuous Fourier Transform. . . . 5
- 2.3 Basic Properties of the Spectral Method used in two Dimensions. . . 6
  - 2.3.1 The Discrete Fourier Transform. . . . 6
  - 2.3.2 The Continuous Fourier Transform in 2D. . . . 8

3 The Connection Machine. **9**
- 3.1 The Processor Elements of the CM. . . . 9
- 3.2 The layout of elements. . . . 10
- 3.3 The FFT on the CM. . . . 11

4 Fast Fourier Transforms on the Connection Machine. **13**
- 4.1 Conventions. . . . 13
- 4.2 DFT of two N-point real Sequences by means of one N-point Complex DFT. . . . 14
- 4.3 IDFT of two N-point Conjugate even Sequences by means of one N-point Complex IDFT. . . . 15
- 4.4 DFT of one 2N-point real Sequence by means of one N-point Complex DFT. . . . 16
- 4.5 IDFT of one 2N-point Conjugate even Sequence by means of one N-point Complex IDFT. . . . 17
- 4.6 Computation of a 2D-DFT on real Elements by use of Complex 1D-DFTs. . . . 19
- 4.7 Computation of a 2D-IDFT by use of Complex 1D-IDFTs. . . . 21
- 4.8 Implementation of the 1D Real-to-Complex FFT on the Connection Machine. . . . 23
- 4.9 Implementation of the 2D Real-to-Complex FFT on the Connection Machine. . . . 25
- 4.10 Implementation of the 2D Complex-to-Real IFFT on the Connection Machine. . . . 27

5 The Linear PDE with one Spatial Dimension. **29**
- 5.1 Transformation of a Spatially Discretized Linear PDE into an ODE. 29
- 5.2 Solving the Linear PDE using the Continuous Transform. . . . 31
- 5.3 Implementation of a Solver of a PDE with 1 Spatial Dimension on the Connection Machine. . . . 32
- 5.4 Description of the Program Linpde1d. . . . 32

5.5 A Test of the Program Linpde1d. 35

6 The Linear PDE with two Spatial Dimensions. 36
6.1 Solving the Linear PDE by use of the Continuous Transform. 36
6.2 A Discretized Solution to a Linear PDE with 2 Spatial Dimensions. . 38
6.3 Implementation of a Solver of a Linear PDE with 2 Spatial Dimensions on the Connection Machine. 39
6.4 Description of the Program Linpde2d. 39
6.5 A Test of the Program Linpde2d. 43

7 General one step methods for solving ODEs. 44
7.1 Euler's Method. 45
7.2 Second Order Explicit Runge Kutta. 45
7.3 Fourth Order Explicit Runge Kutta. 47
7.4 Second Order Semiimplicit Runge Kutta. 47
7.4.1 The Choice of Initial Guess for the SIRK2. 49
7.5 A note on the use of Runge-Kutta methods. 50

8 Properties Regarding ODE's Derived from PDE's with one Spatial Dimension, and their Jacobians. 51
8.1 The Eigenvalues of the Jacobian. 51
8.2 The Problem of Aliasing. 52
8.3 Some Properties Concerning the Jacobian. 54

9 The Korteweg de Vries Equation. 56
9.1 Transforming the KdV Equation into a System of First-Order ODEs. 56
9.2 Solving the ODE's of the Transformed KdV Equation. 58
9.2.1 The Matter of Stability. 58
9.3 Implementation of the Solver. 61
9.4 Solving the Initial Value Problem for the KdV Equation. 62
9.4.1 The Reaction to Different Initial Values. 62
9.4.2 The Difference in using the Operators $u\partial u/\partial x$ and $\partial u^2/(2\partial x)$. 66
9.4.3 The Error Rate of the Different Runge Kutta Methods. . . . 66
9.4.4 The Effect on Accuracy when Changing the Spatial Resolution. 68
9.4.5 The Stability of the Runge Kutta Methods. 68
9.4.6 The Effect of using Antialiasing. 69
9.4.7 The Parameters of the Semiimplicit Runge Kutta. 69
9.4.8 The Efficiency of the Programs. 71
9.4.9 Concluding Remarks Regarding the Solution of the Initial Value Problem for the KdV Equation. 72

10 Extending the KdV Equation. 73
10.1 The Existence of Solitons for the EKdV Equation. 73
10.2 Implementation of the Initial Value Solver. 74
10.3 Computational Results for the Extended KdV Equation. 74
10.3.1 The case $\gamma\beta > 0$, $V\beta > 0$. 74
10.3.2 The case $\gamma\beta > 0$, $V\beta < 0$. 77
10.3.3 The case $\gamma\beta < 0$, $V\beta > 0$. 77
10.3.4 The case $\gamma\beta < 0$, $V\beta < 0$. 77
10.4 Concluding Remarks on the Extended KdV equation. 77

11 Properties Regarding ODE's Derived from PDE's with two Spatial Dimensions, and their Jacobians. 79
11.1 The Eigenvalues of the Jacobian. 79
11.2 Some Properties Concerning the Jacobian. 81
11.3 The Matter of Antialiasing. 85

12 The Hasegawa-Mima Equation. 86
12.1 Fourier Transformation of the PDE. 86
12.2 Discretization of the ODE. 87
12.3 Solution of the Discretized ODE. 88
12.3.1 The Jacobi-Matrix of the Discretized ODE. 88
12.3.2 The Eigenvalues of the Jacobi-Matrix of the Discretized ODE. 90
12.4 Implementation of a PDEsolver, *pde2d*, using ERK4. 92
12.5 Improving the Solver with Antialiasing. 93
12.5.1 The Eigenvalues of the Antialiased ODE. 93
12.6 Testing the Improved Solver. 97
12.6.1 Relating the Solutions to Physics. 98
12.6.2 The Stability and Floprate of the Solver. 99

13 Concluding Remarks. 102

14 Acknowledgments. 104

A FFT Performance. 106
A.1 One-dimensional Transforms. 106
A.2 Two-dimensional Transforms. 111

B Some visualization utilities. 115
B.1 The Program, 'tiff2dv'. 115
B.2 The Program, 'dv2movie'. 116
B.3 Code Listing of 'tiff2dv.cs'. 117
B.4 Code Listing of 'dv2movie.cs'. 117

C The Computation of the Eigenvalues of the Hasegawa-Mima Equation. 118

D Real Fourier Transform code for CM. 120
D.1 Code for One-dimensional Real-to-Complex FFT. 121
D.2 Code for Two-dimensional Real-to-Complex FFT and Complex-to-Real IFFT. 121

E Linear PDE Solver code for CM. 122
E.1 The 1-dimensional Solver. 123
E.2 The 2-dimensional Solver. 123

F Results of the KdV programs. 124

G Code KdV eq. Solver. 128

H Flops Required to Solve the HM-Equation. 130

I Code for Hasegawa Mima Solver. 131

Chapter 1

Introduction.

The past years have shown a growing interest in the solution of nonlinear equations. This is due to the development of still more powerful computers and especially the development of more efficient numerical schemes that compensate for the inability to solve nonlinear equations analytically. The subject of this document is primarily to implement a numerical method for solving the initial value problem for nonlinear partial differential equations on a massively parallel computer, the Connection Machine CM-200. To benefit from our description of the numerical method, the reader should be familiar with the basic properties of numerical analysis, linear algebra and Fourier transforms - while reading the description of the implementation requires a basic knowledge of the Connection Machine.

The basic concept of the method, known as the spectral method for solving partial differential equations, is to transform a partial differential equation into a system of ordinary differential equations. This transformation is defined along with some of its properties in chapter 2.

Due to the lack of Fourier transforms for real valued sequences on the Connection Machine, chapter 4 describes an efficient implementation of a real to complex FFT on the basis of detailed knowledge of the hardware model described in chapter 3.

Chapter 5 and 6 present an analytical solution of the one and two dimensional linear partial differential equations using Fourier transforms. This is done to acquaint ourselves with the use of the spectral method.

In order to solve initial value problems for nonlinear one and two dimensional partial differential equations we deduce some general properties concerning the ordinary differential equations derived from these systems in the chapters 8 and 11.

Before commencing the analysis of the specific equations we introduce the one-step methods for solving ordinary differential equations in chapter 7. The description covers the definition of explicit and implicit Runge-Kutta methods, and investigates a handful of these methods more thoroughly, primarily in order to clarify their regions of absolute stability.

The main chapters 9, 10 and 12 describe the solution of the initial value problems for the Korteweg de Vries equation, the Korteweg de Vries equation expanded with a fifth order derivative and the Hasegawa-Mima equation. In all cases we implement a solver, and analyze the stability, and to some extent also the precision and efficiency of the CM implementation.

For the handling of the vast amounts of data produced when solving the equations described above, we have developed a set of tools for visualization, described in appendix B.

Chapter 13 contains some concluding remarks as well as some proposals for future work in this area.

Chapter 2

The Spectral Method.

The initial value problem for PDE's is in general

$$\begin{aligned} \frac{\partial u(t)}{\partial t} &= f(t, u(t)),\ u(a) = u_0 \\ f:\ \mathcal{R}^2 &\rightarrow \mathcal{R} \\ u(t): \mathcal{R}^n \rightarrow \mathcal{R},\ u_0: \mathcal{R}^n &\rightarrow \mathcal{R},\ t \in [a; b] \end{aligned} \tag{2.1}$$

where we will restrict ourselves to handle the case $n = 1, 2$. The spectral method can be used in two different ways. One way is to discretize u_0, and apply it to the PDE, turning it into a set of coupled PDE's (we will later refer to this as a discretized PDE). This set of PDE's can, as we will soon see, be transformed into a set of ordinary differential equations (ODE's), and retransforming their solution will then yield, as we will also see, a discretized solution of the PDE. In this case the transform is the discrete Fourier transform. Another approach, which is convenient when dealing with f's that are more complex, is to transform the PDE $\partial u(t)/\partial t = f(t, u(t))$ into an ODE, transform u_0, discretize the transformed u_0, and apply this to the ODE, thereby obtaining the same set of ODE's as when using the first approach. In this second case the transform is the continuous Fourier transform. We will, to put it short, turn the problem into solving a set of coupled ODE's by the use of two methods, either a discretization succeeded by a transform, or a transform succeeded by a discretization.

2.1 General Conventions

In order to obtain a concise notation, we use a concept that arises when working with massively parallel computers. If $u(x, t)$ is a real function of two variables, then we define

$$u(\vec{x}, t),\ \vec{x} = (x_0, x_1, \ldots, x_{N-1})$$

as the vector

$$(u(x_0, t), u(x_1, t), \ldots, u(x_{N-1}, t))$$

This is especially used with functions such as e^x, and x^p, but also with for example $\partial u(x, t)/\partial t$. Furthermore, the product $\vec{x} \odot \vec{y}$ defines elementwise multiplication.

For simplicity we index vectors of length N in the Fourier domain (defined in the next section) from $-N/2$ to $N/2 - 1$, knowing that the indices will have to be shifted into $1, 2, \ldots, N$ when using these vectors in a computer program.

The same convention is used in 2D; if $f(x, y, t)$ is a real function of three variables, then we define

$$f(\vec{x}, \vec{y}, t),\ \vec{x} = (x_0, x_1, \ldots, x_{N-1}),\ \vec{y} = (y_0, y_1, \ldots, y_{M-1}),$$

as a matrix where element (i,j) is

$$f(x_i, y_j, t),\ i = 0, 1, \ldots, N-1,\ j = 0, 1, \ldots, M-1.$$

This notation is used in things like $\underline{\underline{\hat{u}}} = \mathcal{DFT}(\underline{\underline{u}})$, where $\underline{\underline{u}}$ is a matrix consisting of the evaluation of $u(x,y)$ in the grid-points given by $\vec{x}$, $\vec{y}$. Again, the product $\underline{\underline{x}} \odot \underline{\underline{y}}$ defines elementwise multiplication, which should not be confused with the ordinary matrix product. As with vectors, we index N by M arrays in the Fourier domain (defined in section 2.2.2) from $-N/2$ to $N/2-1$ and from $-M/2$ to $M/2-1$, knowing that the indexes will have to be shifted into $1, 2, \ldots, N$ and $1, 2, \ldots, M$, when using these arrays in a computer program.

2.2 Basic Properties of the Spectral Method used in one Dimension.

2.2.1 The Discrete Fourier Transform.

When given a function, $u(x,t)$, $x \in [a; a+L]$, $L > 0$, and an N-point discretization, $x_j = a + Lj/N$, $j = 0, 1, \ldots, N-1$, the discrete Fourier transform, $\mathcal{DFT}(\vec{u}(t))$, is defined by

$$\hat{u}_k(t) = \sum_{j=0}^{N-1} u_j(t) e^{\frac{-2\pi ijk}{N}},\ k = -\frac{N}{2}, \ldots, \frac{N}{2} - 1$$

and the inverse transform, $\mathcal{IDFT}(\hat{\vec{u}}(t))$, by

$$u_j(t) = \frac{1}{N} \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \hat{u}_k(t) e^{\frac{2\pi ijk}{N}},\ j = 0, \ldots, N-1$$

where $u_j(t)$ denotes $u(x_j, t)$. In the following $\hat{\vec{u}}(t)$ denotes a Fourier transform of the spatial dimension of $u(\vec{x}, t)$.

Since $2\pi ijk/N = 2\pi ik(x_j - a)/L$, it follows that the derivatives can be found by

$$\begin{aligned}
\left.\frac{\partial^p u(x,t)}{\partial x^p}\right|_{x=x_j} &= \left.\frac{\partial^p}{\partial x^p} \frac{1}{N} \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \hat{u}_k(t) e^{\frac{2\pi ik(x-a)}{L}}\right|_{x=x_j} \\
&= \frac{1}{N} \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} (2\pi ik/L)^p \hat{u}_k(t) e^{\frac{2\pi ik(x_j-a)}{L}} \\
&= \mathcal{IDFT}(\vec{e}^p \odot \hat{\vec{u}}(t))_j,\ j = 0, 1, \ldots, N-1, \\
& \quad e^p_k = (2\pi ik/L)^p,\ p = 1, 2, \ldots \\
& \quad k = -\frac{N}{2}, \ldots, \frac{N}{2} - 1.
\end{aligned}$$

We will consider the p in $\vec{e}^p$ as an index, rather than an elementwise power, since this notion becomes handy later on. The above shows roughly how differentiation is done using the discrete Fourier transformation, however, this formula has to be modified, which can be seen from the following.

Consider a real-valued $\vec{u}$ of length N, $\vec{e}^p$ of the same length, and $\vec{v}$, defined by $\hat{\vec{v}} = \vec{e}^p \odot \hat{\vec{u}}$. Since $\vec{u}$ is real-valued, it follows from the definition of the DFT, that $u_{-N/2}$ is real, and $\hat{u}_k = \hat{u}^*_{-k}$, $k = 1 - N/2, \ldots, N/2 - 1$, which means that

$\hat{u}_0$ is also real. In the above differentiation such a vector of Fourier coefficients is elementwise multiplied by a vector, yielding a vector which is also a vector of Fourier coefficients derived from a real-valued vector, i.e. $\vec{v}$ must be real-valued. Since $\hat{v}_{-N/2} = e_{-N/2}\hat{u}_{-N/2}$, and both $\hat{u}_{-N/2}$ and $\hat{v}_{-N/2}$ are real, it follows that $e^p_{-N/2}$ must be real. It follows likewise that e^p_0 must be real. Since

$$e^p_k \hat{u}_k = \hat{v}_k = \hat{v}^*_{-k} = (e^p_{-k}\hat{u}_{-k})^* = e^{p*}_{-k}\hat{u}^*_{-k} = e^p_{-k}{}^*\hat{u}_k,\ k = 1-\frac{N}{2},\ldots,\frac{N}{2}-1,$$

then

$$e^p_k = e^p_{-k}{}^*,\ k = 1-\frac{N}{2},\ldots,\frac{N}{2}-1,$$

which means that $\vec{e}^p$ must satisfy the same properties as the vectors of Fourier coefficients to which it is multiplied, independent of which PDE that is to be solved.

Since $e^p_k = (2\pi ik/L)^p$ then the property $e^p_k = e^p_{-k}{}^*$, $k = 1-N/2,\ldots,N/2-1$, holds, but $e^p_{-N/2}$ defined as $(-2\pi iN/(2L))^p$ is not automatically real, which it, according to the above, must be. In the case where p is odd, $e^p_{-N/2}$ will be purely imaginary, and this means, that in the case of an odd number of differentiations, the highest Fourier coefficient must be set to zero, in order to maintain a real-valued solution. From this it follows that differentiation in the Fourier-domain transforms into multiplication as follows:

$$\frac{\partial^p u(x,t)}{\partial x^p}|_{x=x_j} = \mathcal{IDFT}(\vec{e}^p \odot \hat{\vec{u}}(t))_j,\ j = 0,1,\ldots,N-1,$$
$$e^p_k = \begin{cases} (2\pi ik/L)^p, & k = 0,\pm 1,\ldots,\pm(\frac{N}{2}-1) \\ \mathrm{Real}((2\pi ik/L)^p), & k = -\frac{N}{2} \end{cases} \tag{2.2}$$

Differentiation in the Fourier domain, with respect to t, is very much simpler, since the exponential-factor in the transformation is timeindependent, which means that the identity

$$\frac{\partial u(\vec{x},t)}{\partial t} = \mathcal{IDFT}(\mathcal{DFT}(u(\vec{x},t))),$$

with the elements

$$\frac{\partial u(x_j,t)}{\partial t} = \frac{1}{N}\sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \frac{\partial \hat{u}_k(t)}{\partial t} e^{\frac{2\pi ikj}{N}},\ j = 0,1,\ldots,N-1,$$

can be, by transforming both sides, written as

$$\frac{\partial \hat{\vec{u}}(t)}{\partial t} = \mathcal{DFT}(\frac{\partial u(\vec{x},t)}{\partial t}),$$

with the elements

$$\frac{\partial \hat{u}_k(t)}{\partial t} = \sum_{j=0}^{N-1} \frac{\partial u(x_j,t)}{\partial t} e^{\frac{-2\pi ikj}{N}},\ k = -\frac{N}{2},\ldots,\frac{N}{2}-1,$$

which shows that time differentiation and spatial Fourier transformation are commutative. Since the spatial Fourier transform yields a set of equations, where the only differentiation is with respect to t, the Fourier transform represents a method of transforming a discretized PDE into a set of ODE's, which is the basic concept of the spectral method.

2.2.2 The Continuous Fourier Transform.

Discretized PDE's that are complicated can become quite unhandy, when transformed into ODE's expressed in terms of Fourier-series. This motivates the use of the continuous Fourier transform, i.e. a transformation that is done prior to the discretization of the equation. When given a function, $u(x,t),\ x \in \mathcal{R}$, the continuous Fourier transform is defined as

$$\mathcal{F}(u(x,t)) = \hat{u}(k,t) = \int_{-\infty}^{\infty} u(x,t)e^{-ikx}dx \tag{2.3}$$

and, provided $\hat{u}(k,t)$ exists, the inverse transform is defined as

$$\mathcal{F}^{-1}(\hat{u}(k,t)) = u(x,t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{u}(k,t)e^{ikx}dk \tag{2.4}$$

where $\hat{u}(k,t)$ denotes a Fourier transform of the spatial dimension of $u(x,t)$.

Since

$$\begin{aligned}\frac{\partial u(x,t)}{\partial x} &= \frac{\partial}{\partial x}\mathcal{F}^{-1}(\hat{u}(k,t)) = \frac{\partial}{\partial x}\frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{u}(k,t)e^{ikx}dk \\ &= \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{u}(k,t)(ik)e^{ikx}dk = \mathcal{F}^{-1}((ik)\hat{u}(k,t))\end{aligned}$$

differentiation turns into multiplication using the continuous Fourier-transformation.

Since the exponential factor in the integrand is time-independent, it is clear that

$$\frac{\partial u(x,t)}{\partial t} = \frac{1}{2\pi}\int_{-\infty}^{\infty} \frac{\partial \hat{u}(k,t)}{\partial t}e^{ikx}dk$$

Forward transforming both sides yields

$$\frac{\partial \hat{u}(k,t)}{\partial t} = \int_{-\infty}^{\infty} \frac{\partial u(x,t)}{\partial t}e^{-ikx}dx$$

which shows that time differentiation and spatial Fourier transform are commutative, as seen in the discrete case.

This means that a Fourier transform of the spatial dimension of a PDE, will turn it into an equation, where the only differentiation is with respect to t, i.e. an ODE.

Spatial differentiation are both in the discrete and in the continuous case transformed into multiplication. The difference between the differentiation factor shows however, that the order of discretization and Fourier transformation is not immediately interchangeable, i.e that using $(ik)^p$ as differentiation factor would not result in the same (meaningful) set of equations as the ones derived when the discretization is done before the Fourier transform. This means that to benefit from the convenience of the continuous transform, a succeeding N-point discretization of the differentiation factor $(ik)^p$ can not be done directly. It should instead be done in a way that results in the set of ODE's, that would be derived by performing the discretization before the transform.

When discretizing a PDE in the interval $[a; a+L]$, the spatial differentiation factor to a certain mode k, is $(2\pi ik/L)^p$ (or 0), while it simply is $(ik)^p$ in the continuous case (where the interval is $\mathcal{R}$). This difference shows that the discretization succeeding the continuous transform, can be done, using the vector found in the case of the discrete transform, wherever spatial differentiation is done.

2.3 Basic Properties of the Spectral Method used in two Dimensions.

2.3.1 The Discrete Fourier Transform.

As in the 1-dimensional case, we will analyze the effect of Fourier transforming differentiation-terms. Consider therefore a function,

$$u(x,y,t),\ (x,y) \in [a;a+L_x] \times [b;b+L_y],\ L_x > 0,\ L_y > 0,$$

and an $N \times M$-point discretization,

$$x_n = a + nL_x/N,\ n = 0,1,\ldots,N-1,\ y_m = a + mL_y/M,\ m = 0,1,\ldots,M-1,$$

and $\underline{\underline{u}}(t) = u(\vec{x},\vec{y},t)$. The discrete Fourier transform, $\mathcal{DFT}(\underline{\underline{u}}(t))$, is defined by

$$\hat{u}_{k,l}(t) = \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} u_{n,m}(t)e^{-2\pi i(\frac{nk}{N}+\frac{ml}{M})},$$
$$k = -\frac{N}{2},\ldots,\frac{N}{2}-1,\ l = -\frac{M}{2},\ldots,\frac{M}{2}-1 \tag{2.5}$$

and the inverse transform, $\mathcal{IDFT}(\hat{\vec{u}}(t))$, by

$$u_{n,m}(t) = \frac{1}{NM}\sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1}\sum_{l=-\frac{M}{2}}^{\frac{M}{2}-1} \hat{u}_{k,l}(t)e^{2\pi i(\frac{nk}{N}+\frac{ml}{M})},$$
$$n = 0,\ldots,N-1,\ m = 0,\ldots,M-1, \tag{2.6}$$

where $u_{n,m}(t)$ denotes $u(x_n,y_m,t)$. In the following $\underline{\underline{\hat{u}}}(t)$ denotes a Fourier transform of the spatial dimensions of $u(\vec{x},\vec{y},t)$, and element (n,m) of $\mathcal{IDFT}(\underline{\underline{\hat{u}}})$ is denoted $\mathcal{IDFT}(\underline{\underline{\hat{u}}})_{n,m}$. Expressing differentiation with respect to x with Fourier series we find

$$\begin{aligned}
\frac{\partial^p u(x,y_m,t)}{\partial x^p}\Big|_{x=x_n} &= \frac{\partial^p}{\partial x^p}\mathcal{IDFT}(\underline{\underline{\hat{u}}}(t))_{n,m}\Big|_{x=x_n} \\
&= \frac{\partial^p}{\partial x^p}\frac{1}{NM}\sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1}\sum_{l=-\frac{M}{2}}^{\frac{M}{2}-1}\hat{u}_{k,l}(t)e^{2\pi i(\frac{nk}{N}+\frac{ml}{M})}\Bigg|_{x=x_n} \\
&= \frac{\partial^p}{\partial x^p}\frac{1}{NM}\sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1}\sum_{l=-\frac{M}{2}}^{\frac{M}{2}-1}\hat{u}_{k,l}(t)e^{2\pi i(k\frac{x-a}{L_x}+\frac{lm}{M})}\Bigg|_{x=x_n}
\end{aligned}$$

From this it can be seen that differentiation with respect to x is done exactly as in the 1D-case, i.e. by elementwise multiplication of $u_{k,l}$ by the k' element of the vector found in 2.1. Since the differentiation is done by multiplying all the columns of $\underline{\underline{\hat{u}}}$ with the same vector, then

$$\frac{\partial^p u(x,y_m,t)}{\partial x^p}\Big|_{x=x_n} = \mathcal{IDFT}(\underline{\underline{A}}^p\,\underline{\underline{\hat{u}}}(t))_{n,m}$$

where

$$A^p_{k,l} = \begin{cases} (2\pi ik/L_x)^p,\ k = 0,\pm1,\ldots,\pm(\frac{N}{2}-1),\ l = -\frac{M}{2},\ldots,\frac{M}{2}-1 \\ \mathrm{Real}((2\pi ik/L_x)^p),\ k = -\frac{N}{2},\ l = -\frac{M}{2},\ldots,\frac{M}{2}-1 \end{cases}$$

where p in $\underline{\underline{A}}^p$ is an index referring to the power used in each element. It is clear that differentiation with respect to y is done in the same way, the only difference is that $\hat{u}_{k,l}$ now should be multiplied by the l' element of the vector found in section 2.1, and therefore

$$\frac{\partial^q u(x_n, y, t)}{\partial y^q}|_{y=y_m} = \mathcal{IDFT}(\underline{\underline{B}}^q \underline{\underline{\hat{u}}}(t))_{n,m}$$

where

$$B^q_{k,l} = \begin{cases} (2\pi i l/L_y)^q, & k = -\frac{N}{2}, \ldots, \frac{N}{2}-1, \ l = 0, \pm 1, \ldots, \pm(\frac{M}{2}-1) \\ \mathrm{Real}((2\pi i l/L_y)^q), & k = -\frac{N}{2}, \ldots, \frac{N}{2}-1, \ l = -\frac{M}{2} \end{cases}$$

From this it follows that

$$\begin{aligned} \frac{\partial^{p+q} u(x,y,t)}{\partial x^p \partial y^q}|_{x=x_n, y=y_m} &= \frac{\partial^p}{\partial x^p}|_{x=x_n} \mathcal{IDFT}(\underline{\underline{B}}^q \underline{\underline{\hat{u}}}(t))_{n,m} = \\ \mathcal{IDFT}(\underline{\underline{A}}^p \underline{\underline{B}}^q \underline{\underline{\hat{u}}}(t))_{n,m} &= \mathcal{IDFT}(\underline{\underline{C}}^{p,q} \underline{\underline{\hat{u}}}(t))_{n,m} \end{aligned} \tag{2.7}$$

where $\underline{\underline{C}}^{p,q} = \underline{\underline{A}}^p \underline{\underline{B}}^q$, i.e.

$$C^{p,q}_{k,l} = \begin{cases} (2\pi i k/L_x)^p (2\pi i l/L_y)^q, \ k = 0, \pm 1, \ldots, \pm(\frac{N}{2}-1), \\ l = 0, \pm 1, \ldots, \pm(\frac{M}{2}-1) \\ (2\pi i k/L_x)^p \mathrm{Real}((2\pi i l/L_y)^q), \ k = 0, \pm 1, \ldots, \pm(\frac{N}{2}-1), \ l = -\frac{M}{2} \\ \mathrm{Real}((2\pi i k/L_x)^p)(2\pi i l/L_y)^q, \ k = -\frac{N}{2}, \ l = 0, \pm 1, \ldots, \pm(\frac{M}{2}-1) \\ \mathrm{Real}((2\pi i k/L_x)^p (2\pi i l/L_y)^q), \ k = -\frac{N}{2}, \ l = -\frac{M}{2} \end{cases}$$

i.e. differentiation transforms into multiplication in the Fourier-domain, analogous to the 1D-case.

Differentiation in the Fourier domain, with respect to t, is, like in the 1D-case, very much simpler, since the exponential-factor in the transformation is timeindependent, which means that the elements of

$$\frac{\partial u(\vec{x}, \vec{y}, t)}{\partial t} = \mathcal{IDFT}(\mathcal{DFT}(u(\vec{x}, \vec{y}, t))),$$

namely

$$\frac{\partial u(x_n, y_m, t)}{\partial t} = \frac{1}{NM} \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \sum_{l=-\frac{M}{2}}^{\frac{M}{2}-1} \frac{\partial \hat{u}_{k,l}(t)}{\partial t} e^{2\pi i (\frac{kn}{N} + \frac{lm}{M})},$$
$$n = 0, 1, \ldots, N-1, m = 0, 1, \ldots, M-1$$

can be, by transforming both sides, written as

$$\frac{\partial \hat{u}_{k,l}(t)}{\partial t} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} \frac{\partial u(x_n, y_m, t)}{\partial t} e^{-2\pi i (\frac{kn}{N} + \frac{lm}{M})},$$
$$k = -\frac{N}{2}, \ldots, \frac{N}{2}-1, \ l = -\frac{M}{2}, \ldots, \frac{M}{2}-1.$$

Written as matrices this is

$$\frac{\partial \underline{\underline{\hat{u}}}(t)}{\partial t} = \mathcal{DFT}(\frac{\partial u(\vec{x}, \vec{y}, t)}{\partial t}),$$

which again shows that time differentiation and spatial Fourier transformation are commutative. As in the 1D-case the spatial Fourier transform therefore yields a set of equations, where the only differentiation is with respect to t, i.e. a set of ODE's.

2.3.2 The Continuous Fourier Transform in 2D.

For the very same reasons as in the 1D-case, we also consider the Fourier transform of continuous terms. When given a function $u(x,y,t)$, $(x,y) \in \mathcal{R}^2$, the 2-dimensional Fourier transformation is defined as

$$\mathcal{F}(u(x,y,t)) = \hat{u}(k,l,t) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} u(x,y,t) e^{-i(kx+ly)} dx dy$$

and, if $\mathcal{F}(u(x,y,t))$ exists, the inverse transform by

$$\mathcal{F}^{-1}(\hat{u}(k,l,t)) = u(x,y,t) = \frac{1}{4\pi^2} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \hat{u}(k,l,t) e^{i(kx+ly)} dk dl$$

where $\hat{u}(k,l,t)$ again denotes a Fourier transform of the spatial dimensions of $u(x,y,t)$. Differentiation expressed in terms of this transformation is very much simpler than using the discrete transform (provided $\hat{u}(k,l,t) = \mathcal{F}(u(x,y,t))$ exists):

$$\begin{aligned}
\frac{\partial^{p+q} u(x,y,t)}{\partial x^p \partial y^q} &= \frac{\partial^{p+q} \mathcal{F}^{-1}(\hat{u}(x,y,t))}{\partial x^p \partial y^q} \\
&= \frac{\partial^{p+q}}{\partial x^p \partial y^q} \frac{1}{4\pi^2} \int_{\mathcal{R}^2} \hat{u}(k,l,t) e^{i(kx+ly)} dk dl \\
&= \frac{1}{4\pi^2} \int_{\mathcal{R}^2} \hat{u}(k,l,t) \frac{\partial^{p+q} e^{i(kx+ly)}}{\partial x^p \partial y^q} dk dl \\
&= \frac{1}{4\pi^2} \int_{\mathcal{R}^2} \hat{u}(k,l,t) (ik)^p (il)^q e^{i(kx+ly)} dk dl \\
&= \mathcal{F}^{-1}(\hat{u}(k,l,t)(ik)^p (il)^q) \qquad (2.8)
\end{aligned}$$

i.e. differentiation transforms into multiplication as in the discretized case. Using the definition yields

$$\frac{\partial u(x,y,t)}{\partial t} = \frac{1}{4\pi^2} \int_{\mathcal{R}^2} \frac{\partial \hat{u}(k,l,t)}{\partial t} e^{i(kx+ly)} dk dl$$

Forward transforming both sides yields

$$\frac{\partial \hat{u}(k,t)}{\partial t} = \mathcal{F}(\frac{\partial u(x,y,t)}{\partial t})$$

which shows that time differentiation and spatial Fourier transform are commutative, as seen in the three other cases.

In subsection 2.3.1 we saw that, when discretizing a 2D PDE in the domain $[a; a + L_x]$, $[b; b + L_y]$, the spatial differentiation factor to a certain mode k, is $(2\pi ik/L)^p$ (or 0), while it simply is $(ik)^p$ in the continuous case (where the domain is $\mathcal{R}^2$). This difference shows that the discretization succeeding the continuous transform, can be done, using the array found in the case of the discrete transform, wherever spatial differentiation is done, analogous to the one-dimensional case.

Chapter 3

The Connection Machine.

In order to be able to write efficient programs for the Connection Machine one has to have quite a detailed knowledge of the machine structure. In this chapter we will describe the CM and the relevant tools for creating efficient programs. The information is primarily based on [3, 4] and our experiences, together with a lot of explanation from the TMC application engineer at UNI•C.

3.1 The Processor Elements of the CM.

The CM is a machine of the SIMD type, meaning that all its processors have to execute the same command at a given time but on different data. The parallel instructions are sent to the CM from a front end computer, which also performs all sequential operations. The CM processors are placed in a Boolean n-cube, where $n \in \{13, 14, 15, 16\}$ depending upon whether we have 8K, 16K, 32K or 64K processors. For every 32 processors, which are one bit each, there is a Weitek coprocessor, and all these coprocessors are then placed in a Boolean $(n-5)$-cube. There are now two basic ways to use the Connection Machine. Either we can place data on the machine in a *fieldwise* manner meaning that a given data element is placed in the memory of a single one bit processor. Whenever we have to perform floating point operations on our dataset, the elements, of each of the 32 one bit processors allocated to a given Weitek processor, one by one sends the data to the coprocessor and wait for it to return. This way of using the machine is supported by the C*, the *Lisp and the CM-FORTRAN paris compilers.

Alternatively we can place our data in a *slicewise* manner, meaning that a given element is spread among the 32 one bit processors of one Weitek coprocessor. When we perform a floating point operation, each processor sends the element to the coprocessor bit by bit in a pipeline. This means that the processor element on which each data element is now placed is no longer equal to a one bit processor, but instead equal to a Weitek coprocessor. As a result of this, we now have to regard the machine as having 2^{n-5} instead of 2^n processor elements (PEs). The only compiler at the present time supporting this machine structure is the CM-FORTRAN slicewise compiler.

It turns out that the slicewise model makes more efficient use of the Weitek processor, which imposes the use of the slicewise model when doing many floating point computations.

Every PE has a unique address which is Gray coded. The Gray code has, by definition, the property that adjacent elements differ in precisely one bit. This means that the distance between two PE's in terms of how many wires one has to go by to reach one PE from another is equal to the Hamming distance between the two PEs address fields (the Hamming distance between to numbers is the number of bits differing in the two numbers, $\sum p \oplus q$).

3.2 The layout of elements.

In this section we will describe some of the needs for effectively utilizing the CM, as well as how data is mapped onto the hardware model and how to choose a correct mapping.

In order to make efficient use of the CM, one has to keep as many processor elements as possible busy simultaneously. Under the fieldwise model, this is generally done by operating on datastructures with at least as many elements as the number of one bit processors. Under the slicewise model the number of elements in the data structures has to be four times the number of Weitek coprocessors. The reason why it is not only the number of PEs, is that the slicewise model uses a vector length of four, meaning that it always has to make computations in series of four. On the Connection Machine at UNI•C, the number of elements to operate on therefore has to be a least 8K under fieldwise and 1K under slicewise if one wishes to get the full benefit of the machine.

It is not enough to have a sufficiently large data set though. It is evenly important to map the data onto the hardware model in an efficient way such that elements that interact often are located close to one another. In order to be able to do this, one needs a fair knowledge of where data is located on the CM.

Mapping a datastructure onto the hardware model is a two step process. First there is created a structure of virtual processors that are mapped onto the hardware model, and then our dataset is mapped onto the virtual processors.

The ideal perspective of massive parallelism is that each element of a data structure should have its own processor. This is not realistic, however, when operating on large data sets. Instead we create a structure of virtual processors, that is capable of containing our data structure in a straightforward way such that each element of our data structure is assigned to its own virtual processor. This structure of virtual processors is then mapped down onto the physical processor elements. The set of virtual processors is created by extending each axis of our p-dimensional array to a power of 2 and then calculating the total amount of elements in this new structure. If this number is equal to or larger than the number of processing elements, then the number of virtual processors to be created is equal to this number. If instead, the number is smaller than the number of processing elements the number of virtual processors to be created is equal to the number of PEs. Each of these virtual processors gets a specific address (called a send address) which in binary notation is composed by a PE address followed by an index number,

$$\langle p_n, p_{n-1}, ..., p_1, p_0 | i_m, i_{m-1}, ..., i_1, i_0 \rangle$$

The PE address specifies which PE the virtual processor is allocated on and the index identifies the different virtual processors on a given PE.

The mapping of data structures onto the virtual processor model can be done in three ways: SEND, NEWS and SERIAL. If we have a one-dimensional array laid out as SEND it means that the array index of a given element is equal to the send address. This means that elements that are close to each other on the hardware model are the ones with small Hamming distances between their index numbers. If we instead lay out the array as NEWS, the elements which are adjacent in the sequence will also be adjacent on the hyper cube [1]. The SERIAL layout means

[1] This is mostly true, but sometimes adjacent elements will have a physical distance of two.

that we want to assign the sequence to a single processor. When using it on a one-dimensional array, the array will live on the front end computer.

When having a multidimensional array, we can assign different layout methods to different axes. Thus we can e.g. choose to make one axis SEND and all the rest NEWS. An array of dimension p is mapped onto the VP set by 'chopping' up the address field's PE address part and its index part into p chunks each. Each axis of the array is then linked to an address field composed by one PE address chunk and one index part chunk,

$$\langle p_p, p_{p-1}, ..., p_{q+1}, p_q | i_m, i_{m-1}, ..., i_{n+1}, i_n \rangle$$

and the elements along the axis is then mapped onto the address field as described for the one-dimensional case above[2].

When we have a multidimensional array, we can furthermore choose how to distribute the index part of the address field between the different axes. This is done by assigning 'weights'. A large weight along a given axis means that the axis should obtain a relatively large chunk from the index part. This would imply that communication along that axis would be faster than along the other axes since the elements along that axis would live close to a single PE (if not entirely on).

It should now be obvious that layouts and weights are powerful tools when optimizing programs.

3.3 The FFT on the CM.

In the mathematic library for the CM there exists an implementation of the Cooley-Tukey FFT. The subroutines for performing the FFTs can be found in [3]. They consist of a setup routine that calculates the twiddle factors etc., a routine that performs the transform, and a deallocation routine that frees the memory allocated for twiddle factors etc.. When using these subroutines it is vital to use the SEND layout along the axis that is transformed. The reason for this lies in the nature of the butterfly network describing the FFT. As described in [1, 2, 4] the FFT can be viewed as being composed by a number of stages in which series of two elements are added to one another and multiplied by a constant. A typical network for a sequence of 16 elements is shown in figure 3.1.

Based on this is should be fairly easy to convince oneself that when having a sequence of $P = 2^p$ elements, a given 'row' y in the butterfly network has connection to the 'rows' $y \oplus 2^z, \quad \forall z \in \{0, 1, ..., p-1\}$ ($\oplus$ denotes the bit-wise exclusive-or operation). This is exactly the definition of a Boolean p-cube, which means that if we map the sequence we want to transform onto the machine as SEND, then the elements that have to interact at a given step will be neighbors on the hypercube, which implies that the communication cost will be low. If we instead used a NEWS layout the communication would be a tremendous bottleneck, which is why the subroutine, when called with an array of layout NEWS, transforms it to a SEND layout prior to making the computation (afterwards it likewise makes a transformation back to the NEWS layout). Due to this overhead in communication, the execution rate when performing FFTs on NEWS structure arrays is 2-3 times as slow as the execution rate on SEND structured arrays.

In appendix A we present some time measurements of the CMSSL v. 2.2 FFTs in 1D and 2D. We also calculate a MFlops/s rate, which is done by assuming that the number of floating point operations is $5N/2 \log_2 N$ for the one-dimensional transforms on a vector of length N. This should be fairly accurate since this is what

[2] The SERIAL layout is a little different. If an axis is serial all its elements have to reside on a single PE, which means that the PE address chunk is of size zero. In order for this to be possible, the above described creation of the virtual processor set sometimes has to be modified.

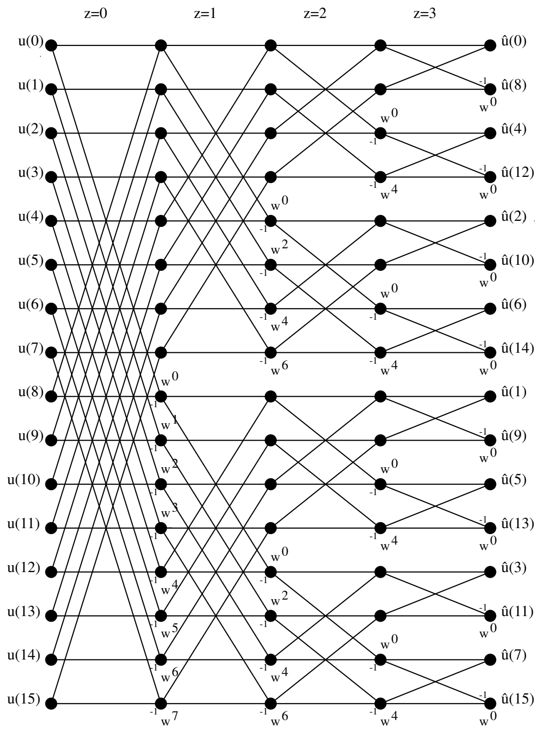

Figure 3.1: Butterfly network of a 16 point sequence.

is specified for the Cooley-Tukey radix-2 FFT, and it should not differ that much even though the subroutines also use radix 4 and 8 algorithms.

Chapter 4

Fast Fourier Transforms on the Connection Machine.

In this chapter we will present algorithms for computing FFT on a real sequence. Since [1, 2] are incomplete on the subject we have been forced to derive the algorithms for the one-dimensional problem before we could address the two-dimensional matter. After having derived the algorithms we discuss and present an implemention in CM-FORTRAN and compare the performance with the FFT routines from the CMSSL-library.

4.1 Conventions.

In order to understand our deduction of the algorithms we need the reader to be familiar with the following definitions and sentences.

In $f_{a,b}$ we define a as the first and b as the second axis.

If F is of length N then F_N refers to F_0.

Definition 4.1.1 *We define the domain $\mathcal{K}$ as a domain in which all members are equal to their inversely permuted conjugate.*

If the Fourier coefficients of F is of length M and $\forall k$: $F_k = F^*_{M-k}$ then we say that F belongs to $\mathcal{K}^M$.

If the Fourier coefficients of F is M by N and $\forall k,l$: $F_{k,l} = F^*_{M-k,N-l}$ then we say that F belongs to $\mathcal{K}^{M,N}$.

□

The following lemmas are presented in order to be able to conduct some proofs later on. They are all presented without proofs since these can be found in many books concerning Fourier transforms e.g. [1]. All the following lemmas describe a transformation, but even though we are transforming from lower to uppercase letters we do not say whether we perform a normal DFT or an inverse DFT.

Lemma 4.1.1 *If $f_n \rightsquigarrow F_k$ and $g_n \rightsquigarrow G_k$ then $cf_n + dg_n \rightsquigarrow cF_k + dG_k$ where c and d are arbitrary complex constants.*

□

Lemma 4.1.2 *If $f_n \rightsquigarrow F_k$ then $f^*_n \rightsquigarrow F^*_{-k}$ and $f^*_{-n} \rightsquigarrow F^*_k$.*

$\square$

Lemma 4.1.3 *If f is real and $f \rightsquigarrow F$ then F is conjugate even: $F_k = F^*_{-k}$*

$\square$

Lemma 4.1.4 *If f is purely imaginary and $f \rightsquigarrow F$ then F is conjugate odd: $F_k = -F^*_{-k}$*

$\square$

Lemma 4.1.5 *If $f \rightsquigarrow F$ then f is real and even only if F is real and even.*

$\square$

Lemma 4.1.6 *If $f \rightsquigarrow F$ then f is real and odd only if F is purely imaginary and odd.*

$\square$

Lemma 4.1.7 *If $f \rightsquigarrow F$ then f is purely imaginary and even only if F is purely imaginary and even.*

$\square$

Lemma 4.1.8 *If $f \rightsquigarrow F$ then f is purely imaginary and odd only if F is real and odd.*

$\square$

4.2 DFT of two N-point real Sequences by means of one N-point Complex DFT.

Algorithm 4.2.1 If we wish to perform a DFT on the two N-point real sequences x and y, then we first pack these into one complex sequence:

$$a_n = x_n + iy_n\,, \quad n = 0, 1, \ldots, N-1 \tag{4.1}$$

Now performing the Fourier transform leads to:

$$a \rightsquigarrow A$$

From which we can derive the transforms X and Y:

$$X_k = \frac{1}{2}(A_k + A^*_{N-k}), \quad k = 0, 1, \ldots, N-1 \tag{4.2}$$

$$Y_k = \frac{-i}{2}(A_k - A^*_{N-k}), \quad k = 0, 1, \ldots, N-1 \tag{4.3}$$

$\square$

Proof 4.2.1 *Let x and y be two N-point sequences. We define:*

$$a_n = x_n + iy_n\,, \quad n = 0, 1, \ldots, N-1$$

By splitting into even and odd parts we obtain:

$$\left.\begin{array}{rcl} x^e_n &=& \frac{1}{2}(x_n + x_{N-n}) \\ x^o_n &=& \frac{1}{2}(x_n - x_{N-n}) \\ y^e_n &=& \frac{1}{2}(y_n + y_{N-n}) \\ y^o_n &=& \frac{1}{2}(y_n - y_{N-n}) \end{array}\right\} \Rightarrow$$

$$a_n = x_n^e + x_n^o + iy_n^e + iy_n^o, \quad n = 0, 1, \ldots, N-1$$

Now performing the Fourier transforms yields (due to lemma 4.1.1):

$$A_k = X_k^e + X_k^o + iY_k^e + iY_k^o, \quad k = 0, 1, \ldots, N-1 \tag{4.4}$$

where we because of lemma 4.1.5, 4.1.6, 4.1.7 and 4.1.8 have:
X^e *is real and even,*
X^o *is odd and purely imaginary,*
iY^e *is even and purely imaginary and*
iY^o *is real and odd.*

Using this, we obtain:

$$A_{N-k} = X_k^e - X_k^o + iY_k^e - iY_k^o, \quad k = 0, 1, \ldots, N-1 \tag{4.5}$$

$$\Updownarrow$$

$$A_{N-k}^* = X_k^e + X_k^o - iY_k^e - iY_k^o, \quad k = 0, 1, \ldots, N-1 \tag{4.6}$$

From equation 4.4 and 4.6 it is obvious that,

$$\begin{aligned} X_k &= \tfrac{1}{2}(A_k + A_{N-k}^*), \quad k = 0, 1, \ldots, N-1 \\ Y_k &= \tfrac{-i}{2}(A_k - A_{N-k}^*), \quad k = 0, 1, \ldots, N-1 \end{aligned}$$

□

4.3 IDFT of two N-point Conjugate even Sequences by means of one N-point Complex IDFT.

Algorithm 4.3.1 If we wish to perform an inverse discrete Fourier transform of the two N-point sequences X and Y, which belongs to $\mathcal{K}^N$, then we first pack the sequences into one:

$$A_k = X_k + iY_k, \quad k = 0, 1, \ldots, N-1 \tag{4.7}$$

And then perform the inverse transform on A followed by unpacking:

$$\begin{aligned} &A \rightsquigarrow a \\ &x_n = \textsc{Real}(a_n), \\ &y_n = \textsc{Imag}(a_n), \quad n = 0, 1, \ldots, N-1 \end{aligned}$$

□

Proof 4.3.1 *From equation 4.2 we obtain:*

$$iY_k = \frac{1}{2}(A_k - A_{N-k}^*), \quad k = 0, 1, \ldots, N-1$$

By addition with equation 4.3 we obtain:

$$A_k = X_k + iY_k, \quad k = 0, 1, \ldots, N-1$$

And by performing an inverse discrete Fourier transform on this sequence we return to equation 4.1 from with we can easily derive the sequences x and y. One must remember that X and Y are sequences belonging to $\mathcal{K}^N$, that is $X_k = X_{N-k}^*$ and similarly for Y.

□

4.4 DFT of one 2N-point real Sequence by means of one N-point Complex DFT.

Algorithm 4.4.1 Let f be a real sequence of length 2N. We wish to perform a DFT on f in order to obtain F. First we pack f into g:

$$g_n = f_{2n} + i f_{2n+1}, \quad n = 0, 1, \ldots, N-1 \tag{4.8}$$

And perform a DFT on g in order to obtain:

$$g \rightsquigarrow G$$

$$\begin{aligned} F_k &= \frac{1}{2}(1 - ie^{-2\pi i \frac{k}{2N}})G_k + \frac{1}{2}(1 + ie^{-2\pi i \frac{k}{2N}})G^*_{N-k}, \\ &\quad k = 0, 1, \ldots, N \\ F_k &= F^*_{2N-k}, \quad k = N+1, N+2, \ldots, 2N-1 \end{aligned} \tag{4.9}$$

□

Proof 4.4.1 *From the definition of the discrete Fourier transform, we have:*
Definition 4.4.1

$$F_k = \sum_{n=0}^{2N-1} f_n e^{-2\pi i n \frac{k}{2N}}, \quad k = 0, 1, \ldots, 2N-1$$

□

We now split the sum into:

$$\begin{aligned} F_k &= \sum_{n=0}^{N-1} f_{2n} e^{-2\pi i n \frac{k}{N}} + \\ &\quad \sum_{n=0}^{N-1} f_{2n+1} e^{-2\pi i n \frac{k}{N}} e^{-2\pi i \frac{k}{2N}}, \quad k = 0, 1, \ldots, 2N-1 \end{aligned} \tag{4.10}$$

And since f is a real sequence, lemma 4.1.3 yields:

$$F_{2N-k} = F^*_k, \quad k = 1, 2, \ldots, N-1$$

This leaves us with:

$$\begin{aligned} F_N &= \sum_{n=0}^{N-1} f_{2n} e^0 + \sum_{n=0}^{N-1} f_{2n+1} e^0 e^{-\pi i} \\ &= \sum_{n=0}^{N-1} f_{2n} e^0 - \sum_{n=0}^{N-1} f_{2n+1} e^0 \end{aligned} \tag{4.11}$$

From equation 4.10 and 4.11 we see that all we have to compute is:

$$\begin{aligned} F^e_k &= \sum_{n=0}^{N-1} f_{2n} e^{-2\pi i n \frac{k}{2N}}, \quad k = 0, 1, \ldots, N-1 \\ &= \mathcal{DFT}(f_{0:2:2N-1}) \end{aligned} \tag{4.12}$$

$$\begin{aligned} F^o_k &= \sum_{n=0}^{N-1} f_{2n+1} e^{-2\pi i n \frac{k}{2N}}, \quad k = 0, 1, \ldots, N-1 \\ &= \mathcal{DFT}(f_{1:2:2N-1}) \end{aligned} \tag{4.13}$$

From the equations 4.10, 4.11, 4.12 and 4.13 we conclude, that:

$$\begin{aligned} F_k &= F_k^e + e^{-2\pi i \frac{k}{2N}} F_k^o, \quad k = 0, 1, \ldots, N-1 && (4.14) \\ F_N &= F_0^e - F_0^o && (4.15) \\ F_k &= F_{2N-k}^*, \quad k = N+1, N+2, \ldots, 2N-1 && (4.16) \end{aligned}$$

As shown in section 4.2, the discrete Fourier transforms from equation 4.12 and 4.13 can be computed simultaneously which, when using the notation $G_N = G_0$, all together results in:

$$\begin{aligned} g_n &= f_{2n} + i f_{2n+1}, \quad n = 0, 1, \ldots, N-1 && (4.17) \\ G &= \mathcal{DFT}(g) && (4.18) \\ F_k &= \frac{1}{2}(G_k + G_{N-k}^*) + \frac{-i}{2} e^{-2\pi i \frac{k}{2N}} (G_k - G_{N-k}^*) && \\ &= \frac{1}{2}(1 - i e^{-2\pi i \frac{k}{2N}}) G_k + \frac{1}{2}(1 + i e^{-2\pi i \frac{k}{2N}}) G_{N-k}^*, && (4.19) \\ & k = 0, 1, \ldots, N-1 && \\ F_N &= (\frac{1}{2} + \frac{i}{2}) G_0 + (\frac{1}{2} - \frac{i}{2}) G_0^* && (4.20) \\ F_k &= F_{2N-k}^*, \quad k = N+1, N+2, \ldots, 2N-1 && (4.21) \end{aligned}$$

Which can be reduced to:

$$\begin{aligned} F_k &= \frac{1}{2}(1 - i e^{-2\pi i \frac{k}{2N}}) G_k + \frac{1}{2}(1 + i e^{-2\pi i \frac{k}{2N}}) G_{N-k}^*, && (4.22) \\ & k = 0, 1, \ldots, N && \\ F_k &= F_{2N-k}^*, \quad k = N+1, N+2, \ldots, 2N-1 && (4.23) \end{aligned}$$

□

4.5 IDFT of one 2N-point Conjugate even Sequence by means of one N-point Complex IDFT.

Algorithm 4.5.1 If we want to perform a inverse discrete Fourier transform of F into f then we first reorder F:

$$A_k = \frac{1}{2}(F_k + F_{N+k}) + (F_k - F_{N+k}) \frac{i}{2} e^{2\pi i k \frac{1}{2N}}, N = 0, 1, \ldots, N-1$$

Then we perform and inverse discrete Fourier transform on A in order to obtain:

$$\begin{aligned} & A \rightsquigarrow a \\ & f_{2n} = \text{Real}(a_n), \\ & f_{2n+1} = \text{Imag}(a_n), \quad n = 0, 1, \ldots, N-1 \end{aligned}$$

□

Proof 4.5.1 *From the definition of the discrete inverse Fourier transform, we have:*

Definition 4.5.1

$$f_n = \frac{1}{2N} \sum_{k=0}^{2N-1} F_k e^{2\pi i k \frac{n}{2N}}, \quad n = 0, 1, \ldots, 2N-1$$

$\square$

We now split this sum into:

$$\begin{aligned}
f_{2n} &= \frac{1}{2}\frac{1}{N}\sum_{k=0}^{N-1} F_k e^{2\pi i k \frac{n}{N}} + \frac{1}{2}\frac{1}{N}\sum_{k=0}^{N-1} F_{N+k} e^{2\pi i k \frac{n}{N}} e^{2\pi i n} \\
&= \frac{1}{N}\sum_{k=0}^{N-1} \frac{1}{2}(F_k + F_{N+k}) e^{2\pi i k \frac{n}{N}}, \quad n = 0, 1, \ldots, N-1 \\
&= \mathcal{IDFT}(\frac{1}{2}(F_{0:N-1} + F_{N:2N-1})) \qquad (4.24) \\
f_{2n+1} &= \frac{1}{2}\frac{1}{N}\sum_{k=0}^{N-1} F_k e^{2\pi i k \frac{n}{N}} + \frac{1}{2}\frac{1}{N}\sum_{k=0}^{N-1} F_{N+k} e^{2\pi i k \frac{n}{N}} e^{2\pi i n} e^{2\pi i k \frac{1}{2N}} e^{2\pi i \frac{1}{2}} \\
&= \frac{1}{N}\sum_{k=0}^{N-1} \frac{1}{2}(F_k - F_{N+k}) e^{2\pi i k \frac{1}{2N}} e^{2\pi i k \frac{n}{N}}, \quad n = 0, 1, \ldots, N-1 \\
&= \mathcal{IDFT}(\frac{1}{2}(F_{0:N-1} - F_{N:2N-1}) \odot e^{2\pi i (0:N-1)\frac{1}{2N}}) \qquad (4.25)
\end{aligned}$$

Theorem 4.5.1 *Since F is conjugate even - the two sequences* $\frac{1}{2}(F_k + F_{N+k})$ *and* $\frac{1}{2}(F_k - F_{N+k})e^{2\pi i k \frac{1}{2N}}$ *are also conjugate even.*

$\square$

Proof 4.5.2

$$\begin{array}{l}
(F_k = F^*_{2N-k} \;\Leftrightarrow\; F_{N+k} = F^*_{N-k}) \wedge \\
(F_k = F^*_{2N-k} \;\Leftrightarrow\; F_{-k} = F^*_k) \;\Rightarrow
\end{array}$$

$$\left.\begin{array}{l}
\frac{1}{2}(F_k + F_{N+k}) \\
= \frac{1}{2}(F_k + F^*_{N-k}) \\
\hline
\frac{1}{2}(F_{N-k} + F_{-k}) \\
= \frac{1}{2}(F_k + F^*_{N-k})
\end{array}\right\} \Rightarrow \frac{1}{2}(F_k + F_{N+k}) \textit{ conjugate even}$$

$$\left.\begin{array}{l}
\frac{1}{2}(F_k - F_{N+k})e^{2\pi i k \frac{1}{2N}} \\
= (F_k - F^*_{N-k})\frac{1}{2}e^{2\pi i k \frac{1}{2N}} \\
\hline
\frac{1}{2}(F_{N-k} - F_{-k})e^{2\pi i (N-k)\frac{1}{2N}} \\
= (F_k - F^*_{N-k})\frac{-1}{2}e^{-\pi i}e^{2\pi i k \frac{1}{2N}} \\
= (F_k - F^*_{N-k})\frac{1}{2}e^{2\pi i k \frac{1}{2N}}
\end{array}\right\} \Rightarrow \begin{array}{l} \frac{1}{2}(F_k + F_{N+k})e^{2\pi i k \frac{1}{2N}} \\ \textit{conjugate even} \end{array}$$

$\square$

Since the two sequences in the equations 4.24 and 4.25 are conjugate even, one can perform the two inverse discrete Fourier transforms simultaneously as shown in section 4.3 by computing:

$$A_k = \frac{1}{2}(F_k + F_{N+k}) + (F_k - F_{N+k})\frac{i}{2}e^{2\pi i k \frac{1}{2N}}, N = 0, 1, \ldots, N-1 \qquad (4.26)$$

And then performing an inverse discrete Fourier transform in order to obtain:

$$a_n = f_{2n} + i f_{2n+1}, \quad n = 0, 1, \ldots, N-1 \qquad (4.27)$$

$\square$

4.6 Computation of a 2D-DFT on real Elements by use of Complex 1D-DFTs.

We define the 2D-DFT on a N by M sequence as:
Definition 4.6.1

$$F_{k,l} = \sum_{m=0}^{M-1} [\sum_{n=0}^{N-1} f_{n,m} e^{-2\pi i k \frac{n}{N}}] e^{-2\pi i l \frac{m}{M}},$$
$$k = 0, 1, \ldots, N-1 \quad l = 0, 1, \ldots, M-1$$

□

From this it is obvious that two-dimensional discrete Fourier transforms can be performed by means of one-dimensional DFTs. One way to look at it, is that one first has to perform N one-dimensional DFTs on all the rows. After this, one has to compute M one-dimensional DFTs on all the columns respectively.

Now let us take a look at the algorithm when exposed to real data. In the first part of the algorithm, where we have to perform transforms along the first axis, we can choose to either perform $N/2$ DFTs each of length M, or we can perform N DFTs each of length $M/2$, depending upon whether we use the algorithm described in section 4.2 or 4.4 respectively. Since the FFT algorithm implemented on the Connection Machine is based on the Cooley-Tukey algorithm which requires approximately $5N/2log_2N$ floating point operations in order to perform a N-point complex one-dimensional DFT (see [4]), it should be most cost efficient to perform many FFTs of half length rather than half the amount of full length. Furthermore, one must remember that the Connection Machine has a tremendous bottleneck in communication which speaks in favor of performing many short FFTs. Based on this discussion we must assume that the most performance efficient implementation is to pack the real array into a half length complex array on the active axis, and then perform the N half length DFTs as described in section 4.4.

Now we must perform transforms on the other axis. Since the transforms on the first axis was on real data, the array is now (due to lemma 4.1.3) conjugate even on the first axis. Since the transform of conjugate data can be deduced from the transform on nonconjugate data using lemma 4.1.2, we only have to perform $M/2$ N-length transforms on the second axis.

All together this results in an amount of work generally a little more than 50% in comparison with the complex two-dimensional FFT, as seen in appendix H.

The complete derived algorithm is as follows:

Algorithm 4.6.1 First we pack the real array of N times M into a complex array of N times $M/2$:

$$g_{n,m} = f_{n,2m} + i f_{n,2m+1}, \quad n = 0, 1, \ldots, N-1, \quad m = 0, 1, \ldots, \frac{M}{2} - 1$$

Then we perform DFTs on all rows:

$$G_{k,l} = \mathcal{DFT}(g_{k,0:\frac{N}{2}-1})(l)$$
$$k = 0, 1, \ldots, N-1, \quad l = 0, 1, \ldots, \tfrac{M}{2} - 1$$

Before performing DFTs on all columns we rearrange the contents of the array:

$$I_{k,l} = \begin{cases} (1+i)G^*_{k,0} \\ k = 0,1,\ldots,M-1 \quad l = 0 \\ \\ \frac{1}{2}(1 - ie^{-2\pi i \frac{l}{M}})G_{k,l} + \frac{1}{2}(1 + ie^{-2\pi i \frac{l}{M}})G^*_{k,\frac{M}{2}-l} \\ k = 0,1,\ldots,N-1, \quad l = 1,2,\ldots,\frac{M}{2}-1 \end{cases}$$

Performing the transform on all columns yields:

$$J_{k,l} = \mathcal{DFT}(I_{0:N-1,l})(k)$$
$$k = 0,1,\ldots,N-1, \;\; l = 0,1,\ldots,\tfrac{M}{2}-1$$

Remaining is only the unpacking:

$$F_{k,l} = \begin{cases} \frac{1}{2}(J_{k,0} + J^*_{N-k,0}) & k = 0,1,\ldots,N-1, \\ & l = 0 \\ \frac{-i}{2}(J_{k,0} - J^*_{N-k,0}) & k = 0,1,\ldots,N-1, \\ & l = \frac{M}{2} \\ J_{k,l} & k = 0,1,\ldots,N-1, \\ & l = 1,2,\ldots,\frac{M}{2}-1 \\ J^*_{N-k,M-l} & k = 0,1,\ldots,N-1, \\ & l = \frac{M}{2}+1,\frac{M}{2}+2,\ldots,M-1 \end{cases}$$

□

Proof 4.6.1 *Using the algorithm 4.4.1 we pack the real array of N times M into a complex array of N times $M/2$:*

$$g_{n,m} = f_{n,2m} + if_{n,2m+1}, \;\; n = 0,1,\ldots,N-1, \;\; m = 0,1,\ldots,\frac{M}{2}-1$$

Then we issue N $M/2$ length DFTs on all rows:

$$G_{k,l} = \mathcal{DFT}(g_{k,0:\frac{M}{2}-1})(l)$$
$$k = 0,1,\ldots,N-1, \;\; l = 0,1,\ldots,\tfrac{M}{2}-1$$

After this we have to rearrange the array so that we can issue the DFTs on the other axis. As mentioned above we can deduce the transforms of the conjugate sequences from the nonconjugate sequences using lemma 4.1.2. This leaves us with $M/2+1$ DFTs - but since column 0 and $M/2$ are real (due to lemma 4.1.2) we can pack these two columns into one using the algorithm 4.2.1 (we still use the convention that $\forall k: \; G_{k,M} = G_{k,0}$):

$$\begin{aligned} H_{k,l} &= \tfrac{1}{2}(1 - ie^{-2\pi i \frac{l}{M}})G_{k,l} + \tfrac{1}{2}(1 + ie^{-2\pi i \frac{l}{M}})G^*_{k,\frac{M}{2}-l} \\ & \quad\; k = 0,1,\ldots,N, \quad l = 0,1,\ldots,\tfrac{M}{2} \\ I_{k,0} &= H_{k,0} + iH_{k,\frac{M}{2}} \\ &= \tfrac{1}{2}(1-i)G_{k,0} + \tfrac{1}{2}(1+i)G^*_{k,0} \\ & \quad\; + \tfrac{i}{2}(1-i)G_{k,0} + \tfrac{i}{2}(1+i)G^*_{k,0} \\ &= (1+i)G^*_{k,0} \;\; k = 0,1,\ldots,N-1 \\ I_{k,l} &= H_{k,l}, \;\; k = 0,1,\ldots,N-1, \quad l = 1,2,\ldots,\tfrac{M}{2}-1 \end{aligned}$$

Performing N-point discrete transforms on all columns yields:

$$J_{k,l} = \mathcal{DFT}(I_{0:N-1,l})(k)$$
$$k = 0,1,\ldots,N-1, \;\; l = 0,1,\ldots,\tfrac{M}{2}-1$$

All that is left to do now is the unpacking using equation 4.2 and 4.3 as well as lemma 4.1.2:

$$F_{k,l} = \begin{cases} \frac{1}{2}(J_{k,0} + J^*_{N-k,0}) & k = 0,1,\ldots,N-1, \\ & l = 0 \\ \frac{-i}{2}(J_{k,0} - J^*_{N-k,0}) & k = 0,1,\ldots,N-1, \\ & l = \frac{M}{2} \\ J_{k,l} & k = 0,1,\ldots,N-1, \\ & l = 1,2,\ldots,\frac{M}{2}-1 \\ J^*_{N-k,M-l} & k = 0,1,\ldots,N-1, \\ & l = \frac{M}{2}+1, \frac{M}{2}+2,\ldots,M-1 \end{cases}$$

□

4.7 Computation of a 2D-IDFT by use of Complex 1D-IDFTs.

Definition 4.7.1 *We define the two-dimensional inverse discrete Fourier transform on a N by M sequence as:*

$$F_{k,l} = \frac{1}{N}\sum_{n=0}^{N-1}[\frac{1}{M}\sum_{m=0}^{M-1} f_{n,m}e^{2\pi ik\frac{m}{M}}]e^{2\pi il\frac{n}{N}}, \qquad (4.28)$$
$$k = 0,1,\ldots,N-1 \quad l = 0,1,\ldots,M-1$$

□

Based on this definition it is undoubtedly that the two-dimensional inverse Fourier transform can be performed on basis of one-dimensional IDFTs. Initially one issues N inverse transforms along the first axis; then one executes M inverse transforms along the second axis.

In this section we will restrict ourselves to an array which is transformed into a real array - that is an array with the following properties.

Theorem 4.7.1

If F belongs to $\mathcal{K}^{N,M}$, meaning that $\forall k,l$: $F_{k,l} = F^*_{N-k,M-l}$, then the inverse Fourier transforms is real.

□

Proof 4.7.1 *Issuing N IDFTs along the first axis leads to:*

$$\forall k,l\colon\ F_{k,l} = F^*_{N-k,M-l} \leadsto \forall k,m\colon\ F_{k,m} = F^*_{N-k,m}$$

because of lemma 4.1.2.

Then performing M IDFTs on the second axis results in:

$$\forall k,m\colon\ F_{k,m} = F^*_{N-k,m} \leadsto \forall n,m\colon\ f_{n,m}\ real$$

as a consequence of lemma 4.1.3.

□

Corollary 4.7.1 *If the two-dimensional array F belongs to $\mathcal{K}$, then, when having performed inverse Fourier transforms along one axis, each vector along the other axis also belongs to $\mathcal{K}$.*

□

In order to minimize work we have to perform as short and as few Fourier transforms as possible. As discussed in section 4.6, the nature of the Connection Machine motivates the use of half length transforms rather than half the amount of full length transforms. Keeping this in mind we deduce the following algorithm.

Algorithm 4.7.1 FIRST WE PACK OUR INITIAL ARRAY, F INTO G:

$$G_{k,l} = \begin{cases} F_{0,l} + iF_{\frac{N}{2},l} & k = 0, \quad l = 0,1,\ldots,M-1 \\ F_{k,l} & k = 1,2,\ldots,\frac{N}{2}-1, \quad l = 0,1,\ldots,M-1 \end{cases} \tag{4.29}$$

THEN WE ISSUE $N/2$ IDFTS OF LENGTH M ALONG THE FIRST AXIS:

$$H_{k,m} = \mathcal{IDFT}((G_{k,0:M-1})(m) \quad k = 0,1,\ldots,\frac{N}{2}-1, \quad m = 0,1,\ldots,M-1 \tag{4.30}$$

AFTER THIS WE PERFORM A TRANSLOCATION OF THE ELEMENTS:

$$J_{k,m} = \begin{cases} H_{k,m} \quad k = 0, \quad m = 0,1,\ldots,M-1 \\ \\ (\frac{1}{2} + \frac{i}{2}e^{2\pi ik\frac{1}{N}})H_{k,m} + (\frac{1}{2} - \frac{i}{2}e^{2\pi ik\frac{1}{N}})H^*_{\frac{N}{2}-k,m} \\ k = 1,2,\ldots,\frac{N}{2}-1, \quad m = 0,1,\ldots,M-1 \end{cases} \tag{4.31}$$

THEN WE ISSUE M IDFTS EACH OF LENGTH $N/2$ ALONG THE SECOND AXIS:

$$g_{n,m} = \mathcal{IDFT}(J_{0:\frac{N}{2}-1,m})(n) \quad n = 0,1,\ldots,\frac{N}{2}-1, \quad m = 0,1,\ldots,M-1 \tag{4.32}$$

AND FINALLY WE PERFORM THE UNPACKING INTO THE TRANSFORMED ARRAY, f:

$$\begin{aligned} f_{2n,m} &= \text{REAL}(g_{n,m}) \\ f_{2n+1,m} &= \text{IMAG}(g_{n,m}) \quad n = 0,1,\ldots,\tfrac{N}{2}-1, \quad m = 0,1,\ldots,M-1 \end{aligned} \tag{4.33}$$

□

Proof 4.7.2 *First of all we have to compute the inverse Fourier transform along axis 1. Since the array belongs to $\mathcal{K}^{N,M}$ it is clearly perceptible that because of lemma 4.1.2 we do not have to issue IDFT's on the upper part of the array. Furthermore if the length of axis one is N then element 0 and $N/2$ are conjugate even, which means that we, using the algorithm of section 4.3, can compute the inverse transform of these two sequences by means of a single IDFT of length N. All together we compute:*

$$G_{k,l} = \begin{cases} F_{0,l} + iF_{\frac{N}{2},l} & k = 0, \quad l = 0,1,\ldots,M-1 \\ F_{k,l} & k = 1,2,\ldots,\frac{N}{2}-1, \quad l = 0,1,\ldots,M-1 \end{cases}$$

$$H_{k,m} = \mathcal{IDFT}(G_{k,0:M-1})(m) \quad k = 0,1,\ldots,\tfrac{N}{2}-1, \quad m = 0,1,\ldots,M-1$$

The connection between H and the inverse transform of F along axis one is:

$$F \in \mathcal{K}^{N,M}$$
$$F \overset{\text{DFT along axis 1}}{\rightsquigarrow} I$$
$$I_{k,m} = \begin{cases} Real(H_{k,m}) & k = 0, \\ & m = 0,1,\ldots,M-1 \\ H_{k,m} & k = 1,2,\ldots,\frac{N}{2}-1, \\ & n = 0,1,\ldots,M-1 \\ Imag(H_{k,m}) & k = \frac{N}{2}, \\ & m = 0,1,\ldots,M-1 \\ H^*_{N-k,m} & k = \frac{N}{2}+1,\frac{N}{2}+2,\ldots,N-1, \\ & m = 0,1,\ldots,M-1 \end{cases}$$

From corollary 4.7.1 it is obvious that we can use the algorithm 4.5.1 to compute the inverse transform along axis 2. Given the above packing we derive:

$$J_{k,m} = \begin{cases} (\frac{1}{2}+\frac{i}{2}e^{2\pi i k\frac{1}{N}})I_{k,m} + (\frac{1}{2}-\frac{i}{2}e^{2\pi i k\frac{1}{N}})I_{\frac{N}{2}+k,m} \\ k=0,1,\ldots,\frac{N}{2}-1, \quad m=0,1,\ldots,M-1 \end{cases}$$

$$= \begin{cases} (\frac{1}{2}+\frac{i}{2})Real(H_{k,m}) + (\frac{1}{2}-\frac{i}{2})Imag(H_{k,m}) = \\ (\frac{1}{2}+\frac{i}{2}-\frac{i}{2}+\frac{1}{2})H_{k,m} = H_{k,m} \\ k=0, \quad m=0,1,\ldots,M-1 \\ \\ (\frac{1}{2}+\frac{i}{2}e^{2\pi i k\frac{1}{N}})H_{k,m} + (\frac{1}{2}-\frac{i}{2}e^{2\pi i k\frac{1}{N}})H^{*}_{\frac{N}{2}-k,m} \\ k=1,2,\ldots,\frac{N}{2}-1, \quad m=0,1,\ldots,M-1 \end{cases}$$

Performing the IDFTs and then unpacking finally yields:

$$\begin{aligned} g_{n,m} &= \mathcal{IDFT}(J_{0:\frac{N}{2}-1,m})(n), \\ &\quad n=0,1,\ldots,\frac{N}{2}-1, \quad m=0,1,\ldots,M-1 \\ \\ f_{2n,m} &= Real(g_{n,m}), \\ f_{2n+1,m} &= Imag(g_{n,m}), \\ &\quad n=0,1,\ldots,\frac{N}{2}-1, \quad m=0,1,\ldots,M-1 \end{aligned}$$

□

4.8 Implementation of the 1D Real-to-Complex FFT on the Connection Machine.

When writing code for the Connection Machine the first thing to determine is how we want to map our datastructures onto the hardware model. In this case this is fairly easy to determine, since the essential part of our code uses a FFT and as described in section 3 the butterfly structure imposes the use of the SEND-layout. Since we in this case have only one dimension, we do not have to make any considerations concerning weights.

We now analyze the algorithm in order to find out how the most efficient code is implemented.

The packing of the real sequence into the complex sequence is bound to be inefficient using the SEND-layout, since the maximum Hamming distance between the addresses of two adjacent elements, in the sequence is $P-1$ where P is the dimension of the physical hypercube on the CM (this is true only when all the PEs have elements which is a fair enough assumption (see section 3)). Because of this we choose not to perform the packing into the complex sequence as well as the conjugate copying of the positive modes into the negative ones. Omitting this is reasonable since the operations we have to perform (se chapter 2) in the Fourier domain is elementwise multiplication by an odd sequence and in the time domain are elementwise operations with real numbers.

The only thing left is to perform a FFT followed by the following computation from equation 4.10 (F denotes the rearranged array and G denotes the FFT of the packed real array):

$$F_k = \tfrac{1}{2}(1 - ie^{-2\pi i\frac{k}{2N}})G_k + \tfrac{1}{2}(1 + ie^{-2\pi i\frac{k}{2N}})G^{*}_{N-k}, \quad k=0,1,\ldots,N$$

Naturally we will use an initialization subroutine to calculate two constant sequences, so the only thing left to do on the transformed sequence is:

$$\forall k: \quad F_k = k1_k G_k + k2_k G^{*}_{N-k}$$

If we map the constant vectors $k1$ and $k2$ down onto the machine in the same way as F, the above computation will be very efficient, except for the communication needed to perform the permutation of F. In order not to break up the pipeline, it is important to make the permutation prior to the multiplications. In order to ensure this, we first assign the permuted vector to a temporary vector, and then use this when performing the computation.

The communication amount involved in making the permutation is unfortunately quite big. The maximum Hamming distance between the addresses p and $N-p$, where $N=2^n$ and $p=0,1,\ldots,N-1$, is $n-1$ since p and $N-p$ always will share the least significant bit and might differ on all the other ones (e.g. look at 1 and $N-1$). There is however, as far as we know, no way to get around this, and this implicates that we will only have a higher performance than the CMSSL complex-to-complex FFT for long sequences (relative to the order of the binary cube on which it is laid out).

Basically there are two ways of performing the permutation. Either we can perform $N/2$ cyclic shifts using the CSHIFT command, or we can use a permutation vector to perform a command of the type:

$$C = C(perm)$$

The latter is the most preferable in our case, since the CSHIFT command is very slow under SEND layout, due to the long distance between adjacent elements. When using this, it is for the compiler to decide how to implement the permutation. This is in no way thought to be ideal, but it is the best we can do in CM-FORTRAN.

The final code is presented in appendix D.1. It should be noted that we have not performed any optimization on the setup routine that calculates the constant vectors.

We have performed time measurements on the code for sequences of length N for $N=2^n, n=7,8,\ldots,18$; the results are shown here along with the analogous timings for the CMSSL complex-to-complex FFT with normal output.

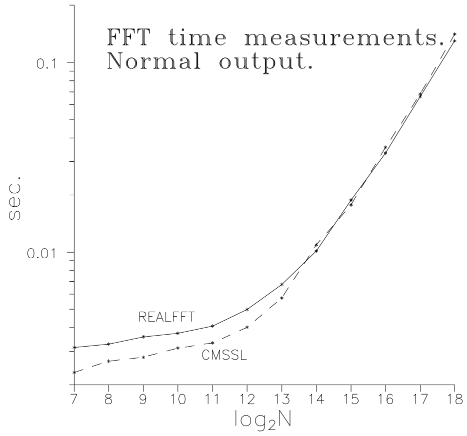

Figure 4.1

The figure is based on the data from table A.1 and A.3 and the maximum speed-up is 1.09. A poor performance is expected for vectors of length 1024 and smaller since these do not fully exploit the Connection Machine but it is remarkable that the communication is so intensive so that time gained by performing less floating point operations is spend on the translocation of the elements, even for long vectors.

If we want to make use of the bit reversed output from the FFT that we use in the algorithm, we would also get the final output in a bit reversed form based on the assumption that we want to minimize communication. The code,

$$Fperm = F(perm)$$
$$F = k1 * F + k2 * Fperm$$

will still work correctly if we bit reverse $k1$ and $k2$ as well as $Fperm$. The bit reversal of $Fperm$ is done the following way. First $perm$ contains the FORTRAN indexes of F in incremental order,

$$perm = [1, 2, \ldots, N]$$

where N is the length of F. This is now considered to be in the bit reversed domain. We then transform the vector to the normal domain and permute it:

$$perm(2 : N) = perm(N : 2 : -1),$$

in order to finally perform a transform on the vector back to the bit reversed domain. The vector $perm$ will now make a correct bit reversal of $Fperm$. The only changes to be made is in the initialization procedure where we calculate $k1$, $k2$ and $perm$. The bit reversed version of the program is integrated into the normal version shown in appendix D.1. Performing the same timings as for the normal output lead to (on basis of table A.2 and A.4):

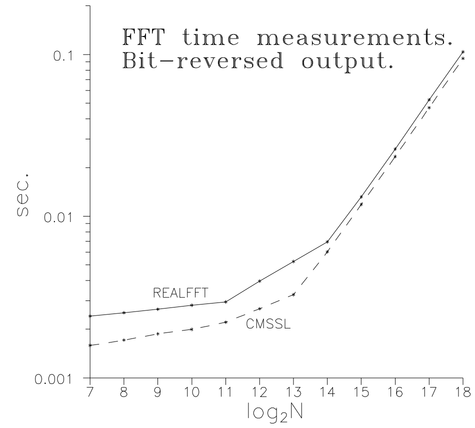

Figure 4.2

where we now compare with the CMSSL bit-reversed output complex-to-complex FFT. The maximum speed up is 0.94 (i.e. no a speed up), and since we do not have a gain in performance - not even for irrelevantly large vectors - we decided not to develop the corresponding code for the complex-to-real inverse FFT.

4.9 Implementation of the 2D Real-to-Complex FFT on the Connection Machine.

The considerations regarding the 2D transform are very similar to the ones concerning the 1D transform.

Due to the many FFTs there is no doubt that we have to use the SEND-layout, which implies that the initial packing of the real array into the complex one will be very communication intensive, as shown in the previous section.

Since the operations we want to perform on the array in the time domain are of the type $A \diamond B$ (where $\diamond$ is an arbitrary elementwise function performing a transform from $\mathcal{R}^{N,M} \times \mathcal{R}^{N,M} \rightarrow \mathcal{R}^{N,M}$), we can effectively perform the operation on the packed array by performing it separately on the real as well as on the imaginary part of the array. Because of this, there is no reason not to keep the arrays packed instead of performing the packing or unpacking every time we perform a transform from or to the time domain respectively.

In the Fourier domain things are a little bit more tricky. We know that we will maintain the "conjugate even" structure of the array, and because of this we only have to perform the operations in the Fourier domain on half the modes, since the other half can be deduced from the first. There is still a problem though, concerning the Fourier coefficients 0 and $N/2$ since these are packed together, which will result in a separate handling of these elements. Due to the nature of the Connection Machine this is bound to be very inefficient, since most of the PEs will be idle while handling this special case. Since we furthermore often do not need the mode $N/2$, as shown in section 2.2.2, we have chosen to eliminate the special case by setting the mode $N/2$ to zero, which we do by not packing it into the first element. This should be reasonable even though we for even derivatives need the upper most negative mode since we only lose a tiny bit of precision, and further more often would have zeroed the upper third of the Fourier coefficients anyway due to aliasing as shown in section 8.2. The total change in the algorithm will thus be in the rearrangement of the elements between the series of 1D FFTs and in the omitment of the initial and final packing/unpacking of the elements.

The implementation of the 1D FFTs along the first and second axes is trivial, and the rearrangement of the elements in between is analogous to the 1D-real FFT implementation except, for the fact that we now refer to vectors instead of single elements.

As a result of now having two axes, we have to make considerations concerning weights. There is no doubt that we need the highest weight along the second axis, since we have chosen to make the communication intensive rearrangement along this axis, which is a swapping of the axes in comparison to the way they are written in algorithm 4.6.1. The reason why we have chosen to swap the axes, is that we later, when performing FFTs on non quadratic arrays, often have the first axis as the longer one. We have tried different weights, and found that if the weight on the second axis is larger than 4 times the weight of the first axis, then we have a maximum performance. It seems to make no difference how many times larger than 4 we make the ratio between the two weights when performing transforms on quadratic arrays, but since we will not always do so, we have chosen to double the weight to 8 in order to ensure that the communication along the second axis will be sufficiently local.

The performance in comparison to the CMSSL 2D FFT with output in normal order on basis of table A.8 and A.12 is:

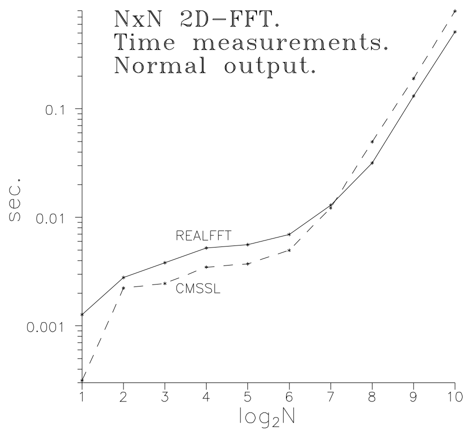

Figure 4.3

where the peak speed-up is 1.57 or 82% of the theoretically possible.

The considerations regarding bit reversal are similar to the one dimensional case, except for the fact that we refer to vectors instead of elements. Due to this we will not discuss the matter further. The code can be found in appendix D.1.

The performance in comparison to the CMSSL 2D FFT with output in bit reversed order is:

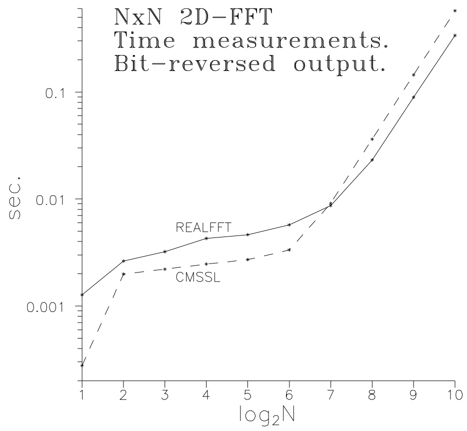

Figure 4.4

The figure is based on table A.9 and A.13 and the maximum speed-up is 1.69 or 88% of the theoretically possible.

4.10 Implementation of the 2D Complex-to-Real IFFT on the Connection Machine.

The algorithm for the inverse Fourier transform presented in section 4.7 and the algorithm for the forward transform are very much alike. The only differences are that one has to perform IFFTs instead of FFTs and multiply by different constant vectors (which turns out to be the conjugate constant vectors from the forward transform). Due to this we will not discuss the IFFT code in details, since it is based on the FFT code. In the previous section we chose to set the lowest (negative) mode to zero. Naturally, we now have to make the code compatible with this. The change in the algorithm is at the translocation of the elements, where we now generalize since the first element now only contains mode 0, and thereby eliminates the special case. The complete code can be found in appendix D.1.

The performance in comparison to the CMSSL 2D IFFT with output in normal order is (based on table A.10 and A.14):

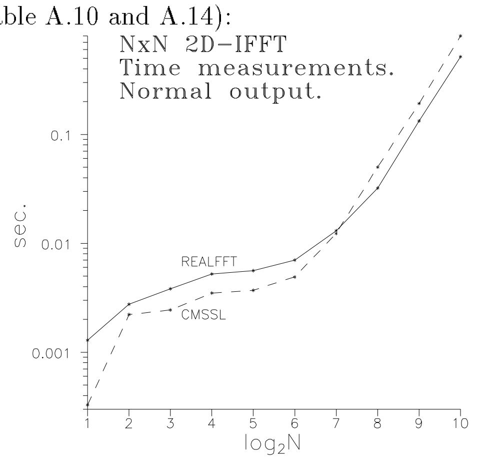

Figure 4.5

The peak speed-up is 1.55 or 81% of the theoretically possible. The performance of the bit reversed version in comparison to the CMSSL 2D IFFT with output in bit reversed order found on table A.11 and A.15 is:

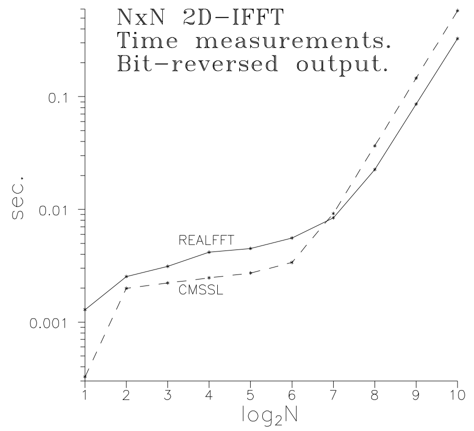

Figure 4.6

where the maximum speed-up is 1.77 or 93% of the theoretically possible.

All together the measurements described above show how a structured code development can lead to an efficient implementation of a complex problem.

Chapter 5

The Linear PDE with one Spatial Dimension.

We have decided to solve a linear PDE, because it is a simple way of getting acquainted with the spectral method, but we have chosen a simple one because it would leave us more time to work with the PDE's that cannot be solved analytically.

5.1 Transformation of a Spatially Discretized Linear PDE into an ODE.

The linear PDE of n' order with one spatial dimension is a differential equation of the form

$$\frac{\partial u(x,t)}{\partial t} = v(x,t) + \sum_{p=0}^{n} C_p(t) \frac{\partial^p u(x,t)}{\partial x^p} \tag{5.1}$$

where $v(x,t)$ and $C_p(t), p = 0, 1, \ldots, n$ are known.
We restrict u to be real, and therefore v and C_p must be real as well.

To make things easier we begin with the much simpler

$$\frac{\partial u(x,t)}{\partial t} = \sum_{p=0}^{n} C_p \frac{\partial^p u(x,t)}{\partial x^p} \tag{5.2}$$

where C_p, $p = 0, 1, \ldots, n$ are known, where we restrict $u(x,t)$ to be 2π-periodic, and where $u(x,0)$ is known in N points, i.e. $u(\vec{x},0)$, $\vec{x} = \frac{2\pi}{N}(0, 1, \ldots, N-1)$ is known.

Using the property from section 2.1 in the Fourier domain, the simple PDE turns into

$$\mathcal{DFT}(\frac{\partial u(\vec{x},t)}{\partial t}) = \sum_{p=0}^{n} c_p \vec{e}_p \hat{\vec{u}}(t), \;\; \vec{x} = \frac{2\pi}{N}(0, 1, \ldots, N-1),$$
$$\vec{e}_p = (real((-i\frac{N}{2})^p), (-i(\frac{N}{2}-1))^p, \ldots, (i(\frac{N}{2}-1))^p).$$

Using the definition of the DFT, it follows that

$$\frac{\partial \hat{\vec{u}}(t)}{\partial t} = \sum_{p=0}^{n} c_p \vec{e}_p \hat{\vec{u}}(t), \;\; \vec{x} = \frac{2\pi}{N}(0, 1, \ldots, N-1),$$
$$\vec{e}_p = (real((-i\frac{N}{2})^p), (-i(\frac{N}{2}-1))^p, \ldots, (i(\frac{N}{2}-1))^p)$$

When denoting $\sum_{p=0}^{n} c_p \vec{e}_p$ by $\vec{a}$, we get

$$\frac{\partial \hat{\vec{u}}(t)}{\partial t} = \vec{a}\hat{\vec{u}}(t),$$

i.e. a set of uncoupled simple ODE's.

This set of ODE's has the solution

$$\hat{\vec{u}}(t) = e^{t\vec{a}}\hat{\vec{u}}(0),$$

where $e^{\vec{v}}$ works elementwise, and where $\hat{\vec{u}}(0)$ holds the Fourier coefficients of the known $u(\vec{x}, 0)$, and therefore

$$u(\vec{x}, t) = \mathcal{IDFT}(e^{t\vec{a}}\mathcal{DFT}(u(\vec{x}, 0))), \tag{5.3}$$

where $u(\vec{x}, 0)$, $\vec{x} = 2\pi/N(0, 1, \ldots, N-1)$ is known. This solution has an interesting property. If $c_p = 0$, for p even, then $\vec{a}$ is purely imaginary, i.e. the Fourier coefficients of the solution can be written as

$$\hat{\vec{u}}(t) = e^{it\vec{b}}\hat{\vec{u}}(0),$$

where $\vec{b}$ is real. Since $\vec{b}_k$ is a polynomial in ik, with coefficients c_p, there will exist a P so that all of $P\vec{b}$'s elements are whole numbers, if no pairs of the coefficients c_p are incommensurable. Using $e^x = e^{x+2p\pi i}$, where p is a whole number, yields

$$\hat{\vec{u}}(t) = e^{i(t\vec{b}+P\vec{b}2\pi)}\hat{\vec{u}}(0) = e^{i\vec{b}(t+2\pi P)}\hat{\vec{u}}(0)$$

It follows from this, that if $c_p = 0$, for p even, then the Fourier coefficients of the solution are periodic in time, with $T = 2P\pi$, where P is the smallest number making $P\vec{b}$ a vector of whole numbers. It is clear that since $\hat{\vec{u}}(t) = \hat{\vec{u}}(t + 2P\pi)$, then

$$\vec{u}(t) = \vec{u}(t + 2P\pi),$$

i.e. that the solution in this case is periodic. If $\vec{b}$ itself consists of whole numbers, i.e. $P = 1$, then the solution will have 2π as period. If furthermore one of the elements of $\vec{b}$ is 1 (this will for instance happen if $c_p = 1, p$ odd, and $c_q = 0, q \neq p$) then 2π will be the shortest period. We will use this in our implementation of a simple PDE-solver.

5.2 Solving the Linear PDE using the Continuous Transform.

In this section a solution to the linear PDE, equation 5.1, is found. To solve this equation the same principle as the above is used, but it is more convenient to use the continuous Fourier transformation. Using the properties in section 2.1 and the linearity of the transform the PDE turns into

$$\begin{aligned}\frac{\partial u(x,t)}{\partial t} &= v(x,t) + \sum_{p=0}^{n} C_p(t)\mathcal{F}^{-1}((ik)^p\hat{u}(k,t)) \\ &= \mathcal{F}^{-1}(\hat{v}(k,t) + \sum_{p=0}^{n} C_p(t)(ik)^p\hat{u}(k,t)) \end{aligned} \tag{5.4}$$

From the definition of $\mathcal{F}$ transform it is clear that

$$\mathcal{F}(\frac{\partial u(x,t)}{\partial t}) = \frac{\partial \mathcal{F}(u(x,t))}{\partial t} = \frac{\partial \hat{u}(k,t))}{\partial t}$$

and therefore the PDE can be transformed to

$$\frac{\partial \hat{u}(k,t))}{\partial t} = \hat{v}(k,t) + A(k,t)\hat{u}(k,t))$$

where $A(k,t)$ denotes $\sum_{p=0}^{n} C_p(t)(ik)^p$. This is, as above, a linear first degree ODE, and it has the solution

$$\begin{aligned}\hat{u}(k,t) &= e^{\mathcal{A}(k,t)}(B(k) + \int e^{-\mathcal{A}(k,t)}\hat{v}(k,t)dt) \\ \mathcal{A}(k,t) &= \int A(k,t)dt = \int \sum_{p=0}^{n} C_p(t)(ik)^p dt = \sum_{p=0}^{n} \mathcal{C}_p(t)(ik)^p \\ \mathcal{C}_p(t) &= \int C_p(t)dt\end{aligned}$$

If $u(x,0)$ is known, then this will be satisfied by

$$\begin{aligned}\hat{u}(k,t) &= e^{\mathcal{A}(k,t)}(\hat{u}(k,0) + \int_0^t e^{-\mathcal{A}(k,\tau)}\hat{v}(k,\tau)d\tau) \\ \mathcal{A}(k,t) &= \int A(k,t)dt = \int \sum_{p=0}^{n} C_p(t)(ik)^p dt = \sum_{p=0}^{n} \mathcal{C}_p(t)(ik)^p \\ \mathcal{C}_p(t) &= \int C_p(t)dt\end{aligned} \tag{5.5}$$

And therefore

$$\begin{aligned}u(x,t) &= \mathcal{F}^{-1}(e^{\mathcal{A}(k,t)}(\hat{u}(k,0) + \int_0^t e^{-\mathcal{A}(k,\tau)}\hat{v}(k,\tau)d\tau)) \\ \mathcal{A}(k,t) &= \int A(k,t)dt = \int \sum_{p=0}^{n} C_p(t)(ik)^p dt = \sum_{p=0}^{n} \mathcal{C}_p(t)(ik)^p \\ \mathcal{C}_p(t) &= \int C_p(t)dt\end{aligned} \tag{5.6}$$

If $\forall(x,t):\ v(x,t) = 0$ and $\forall t:\ C_p(t) = c_p$, then

$$u(x,t) = \mathcal{F}^{-1}(e^{A(k,t)}\hat{u}(k,0))$$

which confirms equation 5.3.

5.3 Implementation of a Solver of a PDE with 1 Spatial Dimension on the Connection Machine.

This section discusses the considerations made in order to efficiently implement the above method to solve equation 5.2 on the Connection Machine 200.

As shown above, the solution of 5.2 is

$$\begin{aligned} u(\vec{x},t) &= \mathcal{IDFT}(e^{t\vec{a}}\mathcal{DFT}(u(\vec{x},0))), \\ \vec{a} &= \sum_{p=0}^{n} \vec{e}_p, \\ \vec{e}_p &= \mathrm{Real}((-i\tfrac{N}{2})^p), (-i(\frac{N}{2}-1))^p, \ldots, (i(\frac{N}{2}-1))^p). \end{aligned} \tag{5.7}$$

The CM-200 is powerful enough to make solutions with a high spatial resolution (that is to make FFT's of long vectors). However, a high spatial resolution will produce a lot of output, so instead of looking at all the numbers, we have chosen to visualize the output. Each point in the investigated interval is represented as a dot, with a color representing the value in that point. Since many graphic screens have a resolution of 1024 by 1280 (including those connected to CM-200), we have chosen $N = 1024$ as the spatial resolution, making it possible to make a picture showing solutions at 1280 steps in time. To actually compute the solution at the time t, the following must be done:

a) At first the Fourier coefficients of the known initial value of the function u must be computed with a FFT.

b) the Fourier coefficients must be elementwise multiplied by the factor $e^{t\vec{a}}$.

c) and finally $u(\vec{x},t)$ is computed with an IFFT.

Since step b) actually yields the Fourier coefficients of $u(\vec{x},t)$, this result can be considered as the result of step a) in a computation of the solution at the time $2t$. What this means, is that if the result of step b) is saved, then a repetition of steps b) and c) will compute the solution at $2t$. This can of course be repeated, yielding the solution at jt after j iterations of step b) and c). Since the factor $e^{t\vec{a}}$ only has to be calculated once, the calculations needed to compute j solutions are j elementwise vector-vector multiplications, and j IFFT's, in our case all of length 1024, with a maximum of 1279 iterations (since the first column in the picture is reserved for the initial value of u). The fact that some linear PDE's have periodic solutions leads to the use of a number of periods (of 2π), instead of an absolute time, which means that the factor $\vec{e}$ may be written as $e^{2\pi p\vec{a}}$, where p is a real number.

5.4 Description of the Program Linpde1d.

The spatial resolution is 1024, defined by the constant $N = 1024$, and the highest derivative that can be computed is the 11' defined by the constant $maxdegree = 12$. This maximum is chosen completely arbitrarily (but quite high), since a maximum of this size has no influence on the amount of calculations that have to be done. The constants c_i describing the PDE are held in the array pde, which is of length $maxdegree$. It is important to remember that the first element of this array is c_0, which means that the coefficient to the first derivative is stored in $maxdegree(1)$. $\vec{e}$ is stored in the array e, which, before $\vec{e}$ is calculated, contains the temporary $\vec{a}$. The array u contains at first the initial value of the solution, and then the Fourier coefficients of the solutions. ru contains the real-valued time domain solutions. The time domain solutions are stored in arrays of type DOUBLE COMPLEX because the CMSSL FFT-routine only does complex to complex transformations. Since the FFT's are the most costly, u and ru are laid out as send, to optimize the performance, and for this reason all arrays are laid out as send (with the exception of pde, because its elements are used only in serial operations).

The number of solutions that are to be computed is stored in $imax$, and the number of periods (of 2π) in time to be calculated is stored in per. As described above, only some PDE's have periodic solutions, but this doesn't prevent us from measuring the time in periods of 2π.The equivalent step in time per solution is stored in h. Since it is desirable to know if an e-array can be reused in the calculation of another picture, the LOGICAL new_pde and the variable h_old keeps track of the need for the computation of a new e-array, when input to a new set of solutions is given. The set of solutions is stored in the INTEGER-array pic where each row contains a solution from 1 step in time. There are 256 colors available, of which we use 255, since the first, black, does not look nice. The solutions are transformed into the interval from 1 to 255 using the variables $minu$ and $conu$.

The program includes 3 libraries, the CM Scientific Software Library with the FFT-routine, a library with a routine to display a CM-array as a picture, and a library with routines for time-measurement.

To use the FFT and the display routines provided by the libraries, an initialization has to be done. This and a reset of pde is done at first.

After the destination of the picture has been input, the program prompts for per, and if it is non-zero, the program enters the main loop of the program, where each iteration solves a PDE.

Each iteration of the loop starts with a call to $make\_pde$, which prints the non-zero coefficients in the PDE, and then prompts for (i, c_i), until an illegal value of i is input. The coefficients are returned in $make\_pde$ and if the contents of $make\_pde$ were changed, then new_pde is set. On return from $make\_pde$ the number of iterations is input until this number is less than 1280.

The next thing is to choose the initial value of the solution. This is done by choosing between 3 different functions. A number is input to $uflag$, and the value of this flag determines which of 3 procedures is to set u to its initial value. $make\_sin$ will set

$$u(i) = \sin(2\pi i/N),\ i = 0, \ldots, N-1,$$

$make\_exp$ will set

$$u(i) = \exp(-10|i - \frac{N}{2}|/N),\ i = 0, \ldots, N-1,$$

and $make\_sech2$ will set

$$u(i) = \frac{3v}{a} sech((i - \frac{N}{2})\pi\sqrt{v}/N)^{-2},\ i = 0, \ldots, N-1.$$

Since the sine is exactly represented in the Fourier domain, the intention with this initial value is to demonstrate how the program works when the solution should be exact. The intention with the exp is to show how the programs reacts on the Gibb's effect coming from the peak in the middle of the solution. Finally the intention with the sech is to show how the solver responds to a initial solution which is quite smooth, and that has all modes represented.

After the initial solution, u, has been calculated, $minu$ and $conu$ are calculated. They are used to transform u into the interval $[1; 255]$. If u lies between $min(u)$ and $max(u)$, and $conu = 254/(max(u) - min(u))$, then it can easily be seen that $(u - min(u) + conu^{-1}) * conu$ lies between 1 and 255. Therefore $conu$ is set to $254/(max(u) - min(u))$, and $minu$ set to $min(u) + conu^{-1}$. This transformation has the advantage of using all the colors available; there is however the drawback that if the minimum or maximum of the solution changes as the iterations proceed, the new extremum will be transformed into a color in the far end of the used palette, resulting in a picture with peaks in a deviant color. This should, however, with the PDE's and initial values we use, not be a problem.

After this, pic is reset, h is calculated, and if h or the PDE has changed then the e-array is calculated (h_old is set to zero before the main loop is entered to ensure that this will be done in the first iteration).

$make\_a$ uses pde to make $\vec{a}$. The array $kvec$ is initialized to contain the purely imaginary factors, ik, from the derivation. Since the FFT-routine swaps the blocks with negative and positive modes, the i' mode has index $i+1$, for $0 \leq i < N/2$, and index $N + i + 1$, for $-N/2 \leq i < 0$. Therefore $kvec(k) = i(k-1), k = 1, \ldots, N/2$, and $kvec(k) = i(k - N - 1), k = N/2 + 1, \ldots, N$. Since $\vec{a}$ is a polynomial in ik with coefficients in pde, the Horner algorithm is used to evaluate this polynomial. After this the imaginary part of mode $-N/2$ is reset.

The FFT works fastest if the order of the Fourier coefficients involved is bit-reversed; the reason for this lies in the FFT-algorithm. Since the Fourier coefficients are bit-reversed, and since the e-array is to be multiplied element-wise by a vector of Fourier coefficients, the elements of the e-array has to be bit-reversed as well. This can be obtained by making a dummy-call of the FFT-routine, where no transformation is performed, but where the output is bit-reversed.

In the case where $make\_a$ is called, h_old is updated, the e-array is computed, and the solving of the PDE commences. A timer is reset and started, the initial value of the solution is stored in pic and transformed into the Fourier domain, and a loop of $imax$ iterations doing step b) and c) of the algorithm described above is entered. The multiplication yields the Fourier coefficients of the solution at step i, they are copied to ru, which is transformed back to the time domain, into the interval [0;255], and stored in pic.

As the loop ends, the timer is stopped, its valued printed, and pic sent to the chosen graphics window using a procedure provided by the display library.

Then the program prompts for a new per, and the main loop will go on until per is zero. When this happens the space reserved for the FFT-routine is deallocated, and the program terminates.

5.5 A Test of the Program Linpde1d.

When solving

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x}, \; 0 \leq t \leq 2\pi,$$

in 1280 steps, the program produces a nice picture showing how the initial solution displaces through a period of 2π, in all three cases of different initial solutions. Since

$$\frac{\partial u}{\partial x} = -\frac{\partial^3 u}{\partial x^3}, \; u(x) = \sin x,$$

then the equation

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x} + \frac{\partial^3 u}{\partial x^3}, \; 0 \leq t \leq 2\pi, \; u(0,x) = \sin x,$$

should cause no changes in time at all, which can be seen using the solver. When solving the same equation, with the second (the exp) and third (the sech) initial solution, the solutions are periodic as expected. The flops used to calculate a screenful (1279) solutions is one FFT, 1279 IFFT's, 1279 vector-multiplications, and 1280 conversions (2N flops each) of the solutions, all of length N=1024. The Fourier transformations are the most costly, $5/2N \log_2 N$ flops each, but since the number of transformations never exceeds 1280, the total number of flops pr. job will be quite modest (about 20-30 Mflops), and therefore the solution of the 1-dimensional linear PDE does not exploit the full potential of the Connection-Machine.

Chapter 6

The Linear PDE with two Spatial Dimensions.

The motivation for solving a linear PDE in two dimensions is again, that we wish to familiarize ourself with the spectral method, this time having to handle two dimensions.

6.1 Solving the Linear PDE by use of the Continuous Transform.

The method is the same as in the 1-dimensional case, but now the equation is:

$$\frac{\partial u(x,y,t)}{\partial t} = v(x,y,t) + \sum_{p=0}^{n}\sum_{q=0}^{m} C_{p,q}(t)\frac{\partial^{p+q} u(x,y,t)}{\partial x^p \partial y^q} \tag{6.1}$$

where $v(x,y,t)$ and $C_{p,q}(t), p = 0,1,\ldots,n,\ q = 0,1,\ldots,m$ are known.

Using the property from section 2.2.2 and the linearity of the Fourier transform, the PDE transforms into

$$\begin{aligned}\frac{\partial u(x,y,t)}{\partial t} &= v(x,y,t) + \sum_{p=0}^{n}\sum_{q=0}^{m} C_{p,q}(t)\mathcal{F}^{-1}((ik)^p(il)^q\hat{u}(k,l,t)) \\ &= \mathcal{F}^{-1}(\hat{v}(k,l,t) + \sum_{p=0}^{n}\sum_{q=0}^{m} C_{p,q}(t)(ik)^p(il)^q\hat{u}(k,t))\end{aligned}$$

Transforming both sides of this equation yields

$$\frac{\partial \hat{u}(k,l,t))}{\partial t} = \hat{v}(k,l,t) + A(k,l,t)\hat{u}(k,l,t))$$

where $A(k,l,t)$ denotes $\sum_{p=0}^{n}\sum_{q=0}^{m} C_{p,q}(t)(ik)^p(il)^q$.

As above, this is a linear first degree ODE, with the solution

$$\begin{aligned}
\hat{u}(k,l,t) &= e^{\mathcal{A}(k,l,t)}(B(k,l) \\
&+ \int e^{-\mathcal{A}(k,l,t)}\hat{v}(k,l,t)dt) \\
\mathcal{A}(k,l,t) &= \int A(k,l,t)dt = \int \sum_{p=0}^{n}\sum_{q=0}^{m} C_{p,q}(t)(ik)^p(il)^q dt \\
&= \sum_{p=0}^{n}\sum_{q=0}^{m} \mathcal{C}_{p,q}(t)(ik)^p(il)^q \\
\mathcal{C}_{p,q}(t) &= \int C_{p,q}(t)dt
\end{aligned}$$

If $u(x,y,0)$ is known, then this will be satisfied by

$$\begin{aligned}
\hat{u}(k,l,t) &= e^{\mathcal{A}(k,l,t)}(\hat{u}(k,l,0) + \int_0^t e^{-\mathcal{A}(k,l,\tau)}\hat{v}(k,l,\tau)d\tau) \\
\mathcal{A}(k,l,t) &= \int A(k,l,t)dt = \int \sum_{p=0}^{n}\sum_{q=0}^{m} C_{p,q}(ik)^p(il)^q dt \\
&= \sum_{p=0}^{n}\sum_{q=0}^{m} \mathcal{C}_{p,q}(t)(ik)^p(il)^q \\
\mathcal{C}_{p,q}(t) &= \int C_{p,q}(t)dt
\end{aligned}$$

And therefore

$$\begin{aligned}
u(x,y,t) &= \mathcal{F}^{-1}(e^{\mathcal{A}(k,l,t)}(\hat{u}(k,l,0) + \int_0^t e^{-\mathcal{A}(k,l,\tau)}\hat{v}(k,l,\tau)d\tau)) \\
\mathcal{A}(k,l,t) &= \int A(k,l,t)dt = \int \sum_{p=0}^{n}\sum_{q=0}^{m} C_{p,q}(t)(ik)^p(il)^q dt \\
&= \sum_{p=0}^{n}\sum_{q=0}^{m} \mathcal{C}_{p,q}(t)(ik)^p(il)^q \\
\mathcal{C}_{p,q}(t) &= \int C_{p,q}(t)dt
\end{aligned}$$

In the simple case where $\forall(x,y,t):\ v(x,y,t) = 0$ and $\forall t:\ C_{p,q}(t) = c_{p,q}$, then

$$u(x,y,t) = \mathcal{F}^{-1}(e^{tA(k,l)}\hat{u}(k,l,0)), \tag{6.2}$$

$$A(k,l) = \sum_{p=0}^{n}\sum_{q=0}^{m} c_{p,q}(ik)^p(il)^q \tag{6.3}$$

6.2 A Discretized Solution to a Linear PDE with 2 Spatial Dimensions.

This section discretizes the result found in the previous section. As in the 1-dimensional case we have chosen to simplify the PDE, in order to get more time to work with nonlinear PDE's.

The simplified equation is:

$$\frac{\partial u(x,y,t)}{\partial t} = \sum_{p=0}^{n}\sum_{q=0}^{m} C_{p,q} \frac{\partial^{p+q} u(x,y,t)}{\partial x^p \partial y^q} \tag{6.4}$$

where $C_{p,q}, p = 0,1,\ldots,n,\ q = 0,1,\ldots,m$ are known. Solving in the domain $(x,y) \in [0;2\pi]^2$, an N by N point-discretization, will require $u(x,y,0)$ to be known for

$$(x,y) = \frac{2\pi}{N}(i,j),\ i = 0,1,\ldots,N-1,\ j = 0,1,\ldots,N-1.$$

Using the solution 6.2 found in the previous section yields:

$$u(\vec{x},\vec{y},t) = \mathcal{IDFT}(()e^{t\underline{\underline{A}}}\mathcal{DFT}(()u(\vec{x},\vec{y},0)), \underline{\underline{A}} = \sum_{p=0}^{n}\sum_{q=0}^{m} c_{p,q}\underline{\underline{C}}_{p,q}, \tag{6.5}$$

where $\underline{\underline{C}}_{p,q}$ is defined in equation 2.3.1.

Since the length of the domain is 2π by 2π, $\underline{\underline{C}}^{p,q}$ can be simplified to

$$C^{p,q}_{k,l} = \begin{cases} (ik)^p(il)^q,\ k = 0,\pm1,\ldots,\pm(\frac{N}{2}-1), \\ l = 0,\pm1,\ldots,\pm(\frac{M}{2}-1) \\ (ik)^p \mathrm{Real}((il)^q),\ k = 0,\pm1,\ldots,\pm(\frac{N}{2}-1),\ l = -\frac{M}{2} \\ \mathrm{Real}((ik)^p)(il)^q,\ k = -\frac{N}{2},\ l = 0,\pm1,\ldots,\pm(\frac{M}{2}-1) \\ \mathrm{Real}((ik)^p(il)^q),\ k = -\frac{N}{2},\ l = -\frac{M}{2} \end{cases}$$

The discrete solution to the simple PDE has, as in the 1-dimensional case, the interesting property that the solution in some cases is periodic. This be can seen in the same way as in the 1-dimensional case. Since the relevant factor now is $e^{t\underline{\underline{A}}}$, $\underline{\underline{A}}$ must purely imaginary, to ensure the periodicity. From the definition of $\underline{\underline{A}}$ it follows that this is obtained, if $c_{p,q} = 0$, for $p+q$ even, so $\underline{\underline{A}}$ can be written as $i\underline{\underline{B}}$, where $\underline{\underline{B}}$ consists of real numbers. Furthermore no pairs of coefficients, $c_{p,q}$, in the PDE may be incommensurable, i.e. there must exist a P, making $P\underline{\underline{B}}$ a matrix of whole numbers. Using again, that $e^x = e^{x+2p\pi i}$, where p is a whole number, yields

$$\hat{u}(t) = e^{i(t\underline{\underline{B}}+P\underline{\underline{B}}2\pi)}\hat{u}(0) = e^{i\underline{\underline{B}}(t+2\pi P)}\hat{u}(0).$$

If P is the smallest number making $P\underline{\underline{B}}$ consist of whole numbers, then $T = 2P\pi$ will be the length of the shortest period of the Fourier coefficients and the solution. Like the 1-dimensional case, a PDE with for instance one $c_{p,q} = 1, p+q$ odd, and all other coefficients zero, will have 2π as shortest period.

6.3 Implementation of a Solver of a Linear PDE with 2 Spatial Dimensions on the Connection Machine.

The considerations about the main algorithm are exactly the same as the ones in section 5.3, but the Fourier transforms and elementwise multiplications are now done in 2D. The concept of measuring the time in units of 2π is used again for the same reasons as in the 1-dimensional case. Since each step in time produces a picture, the entire solution will be a 'film', and therefore there is unlike the 1-dimensional case no hardware determined maximum of the number of possible iterations. It is, however, recommendable to consider if the solution will be periodic, since this will give the user the opportunity to compute only one period of the solution and thereby use a minimum of CPU-time. There is one thing which is not analogous to the 1-dimensional case; in the 2D-case it is feasible to exploit the fact that the solutions are real-valued, and thereby use our real tocomplex FFT described in section 4.9. To use this, the initial real-valued solution has to be packed to a half sized complex array, and the Fourier coefficients returned from $realfft2d$ will be half sized as well. As described in section 4.9 the array of Fourier coefficients will have $N/2$ rows, where the modes $0, 1, \ldots, N/2-1$ are kept in the rows $1, 2, \ldots, N/2$, and where the mode $N/2$ is ignored (the packed array has N columns, and therefore the packed does not effect the column-modes). We assume that the effect of ignoring mode $N/2$ is negligible, since the Fourier modes of the solutions normally are small.

Therefore, all operations in the Fourier-domain will only be half sized, and in this case the only operation that has to be done is an element-wise multiplication with the matrix $e^{h\underline{\underline{A}}}$ ($h = 2\pi per/imax$ is the step in time), and since each element in the packed array represents one mode, the element-wise multiplication can still be done with the packed array (but the factor is now a half sized array).

A slight difference from the 1-dimensional solver is that the initial values are assumed to lie in the interval $[0;1]$, in order to make the conversion to the interval required to get a nice color representation simple. Scaling the initial solution is no problem; consider equation 6.1, with the solution $u(t, x, y)$:

$$\frac{\partial u(x,y,t)}{\partial t} = \sum_{p=0}^{n} \sum_{q=0}^{m} c_{p,q} \frac{\partial^{p+q} u(x,y,t)}{\partial x^p \partial y^q}$$

where $c_{p,q}, p = 0, 1, \ldots, n,\ q = 0, 1, \ldots, m$ is known. Multiplying both sides with any b yields

$$\frac{\partial bu(x,y,t)}{\partial t} = \sum_{p=0}^{n} \sum_{q=0}^{m} c_{p,q} \frac{\partial^{p+q} bu(x,y,t)}{\partial x^p \partial y^q},$$

i.e. if u is a solution, then bu is a solution, for any b.

6.4 Description of the Program Linpde2d.

Unlike the program for the 1-dimensional case, this program has been given the flexibility to solve PDE's with different spatial resolutions within the same job. The reason for this is that jobs using the full screen resolution of 1024 by 1024 cannot by done interactively. Since the program uses variables whose size depends on the input resolution, the main program reads the necessary input, and then calls the procedure PDE_solver, which dynamically declares the resolution-dependent variables and does the computing.

The first parameter to PDE_solver, is the spatial resolution, N (we have for simplicity restricted the resolution to be N by N). N is read after the reset of the descriptor-array of the PDE, and as long as N lies between 1 and 1024 a loop is iterated (the program however, will only work with $N = 2^i, i = 1, 2, \ldots, 10$). The first thing done in this loop is to initialize the graphics screen. Since this only has to be done when N is changed, then N_old holds the previous resolution (N_old is initialized to -1 to ensure initialization at first iteration). Secondly the rest of the parameters are read the integer $tiff\_flag$ tells if the pictures should be stored as a TIFF-file on the front end, and in this case a filename is read. After this, the number of periods (of 2π) in time the calculation should produce is read into per; this can be any real number.

Next to be read is the number of steps in time; this number is read into $imax$. The step in time per solution, h, is therefore the total time divided by the number of solutions, i.e. $h = 2\pi per/imax$.

As in the 1-dimensional case, the user has to choose between a number of different initial values, in this case 4, of the initial solution of the equation; this is read into u_flag.

The program has been given an optional feature, that is to prompt the user between the display of each picture. This can be used if the generated pictures are to be recorded manually on for instance video. This choice is read into $single\_flag$, but since we rarely use this feature, this input has been changed to a comment, and $single\_flag$ is instead set to zero.

The last thing to be read before the call to pde_solver is the coefficients to the PDE. This is done with a call to $make\_pde$. The highest derivative that can be computed in any direction is the 11' defined by the constant $maxdegree = 12$. This maximum is chosen completely arbitrarily (but quite high), since a maximum of this size has no influence on the amount of calculations that has to be done. The constants $c_{i,j}$ describing the PDE is held in the matrix pde, which is $maxdegree$ by $maxdegree$. As in the 1-dimensional case, $pde(1,1)$, holds $c_{0,0}$. The program preserves pde between each solution of a PDE, and therefore $make\_pde$ at first displays the non-zero elements of pde (there will be none at the first iteration, since pde is set to zero in the beginning of the main program), and then lets the user input the desired corrections. This is to make it easier to input slight changes in the PDE-coefficients. The user inputs the coefficient, $c_{p,q}$, to $\partial^{p+q}u/(\partial x^p \partial y^q)$ as $p, q, c_{p,q}$ and this goes on until either p or q has illegal values. The layout of pde is serial, the reason for this can be seen in the way $\underline{\underline{A}}$ is computed. As seen in section 6.2 pde holds the coefficients of a polynomial, and since each element in this polynomial is to be added to a CM-array, a minimum of communication is obtained by making pde serial, causing it to be stored on the front-end (the absolute fastest solution to this, would be to force the CM to hold a copy of pde on each processing element, but since this takes up much more memory, and since the calculations using pde are not time-consuming, we have not done this).

Experience from the 1-dimensional solver shows that the overhead in programming to preserve the e-array in the case where the PDE is not changed (i.e. when the same equation is to be solved with different initial values), does not outweigh the small reduction in processing time used to calculate a new e-array, and therefore $make\_pde$ does not, unlike the 1-dimensional program, return a variable, telling if pde was changed in the call to $make\_pde$.

pde_solver has another parameter, $bit\_reversed$; this constant indicates whether the Fourier coefficients returned by the FFT-transforms are bit-reversed or not. In the developing phase of the program this constant was set to $CMSSL\_normal$, i.e. normal numbered Fourier coefficients, while the actual version uses $CMSSL\_bit\_reversed$, i.e. the Fourier coefficients are all bit-reversed.

The procedure *pde_solver* includes the CMSSL-library with the FFT-routine, a library with routines to save files in the TIFF-format, and a library with routines to use graphics screens. To save a CM-array as a TIFF-picture, information about the color map, i.e. how the numbers in the array corresponds to the colors, must be given along with the array, to the save-routine. Therefore, if $tiff\_flag$ is set, the color map is read into the three variables red, $green$, $blue$. Secondly the routine to perform the FFT-transform is initialized. Since the solutions are all real-valued, we use our own real to complex, 'RTOC'- transform to do this. The initialization is done calling $realfft2d\_setup$, described in 4.9. Apart from the $setup\_id$ passed on to the CMSSL-FFT, $realfft2d$ returns five other parameters, $perm$, $e1$, $e2$, $e3$ and $e4$. These arrays are only used inside $realfft2d$ and $realifft2d$, and they are described in section 4.9.

The matrix $e^{h\underline{\underline{A}}}$, where $h = 2\pi per/imax$ is the step in time, is stored in e, and the matrix is computed in $make\_e$ using pde, per, $imax$ to do this. Since we use our own RTOC-FFT, we have no use for the upper half of the rows of $e^{h\underline{\underline{A}}}$, i.e. we only need the rows corresponding to the modes $k = 0, 1, \ldots, N/2-1, -N/2$. As described in 4.9 the Fourier coefficients returned by $realfft2d$ has mode 0 and $-N/2$, in the reduced dimension, packed together. We find that the unpacking of these two modes is quite time consuming, and therefore we have decided simply to ignore the mode $-N/2$ of the reduced dimension. This means that each row of the array with the Fourier coefficients simply holds the modes $k = 0, 1, \ldots, N/2-1$.

Since $\underline{\underline{A}}$ basically is a polynomial of two (imaginary) variables, $\underline{\underline{A}}$ is evaluated using Horner's algorithm. The evaluation of the polynomial is done in $horner$, where the arguments are the size of the array, N, the size of the polynomial-array, $maxdegree$, the polynomial-array, pde, three arrays, x, y, v with the two arguments, and the result. $horner$ is called with x set to $kvec$, and y set to $lvec$. $kvec$ is of type complex of size $N/2$ by N; each column holds the modes $0, 1, \ldots, N/2$ in the imaginary part (the real part is zero). $lvec$ is of same type and size as $kvec$; each row of $lvec$ holds the column-wise modes $0, 1, \ldots, N/2-1, -N/2, 1-N/2, \ldots, -1$ in the imaginary part, and again the real part is zero. The Horner algorithm is changed however, in order to obtain the $\underline{\underline{A}}$ found in equation 6.5. Since the arguments to the polynomial are purely imaginary, the truncation of the imaginary part can be obtained by skipping the evaluation of the odd powers in the case of mode $-N/2$ (the evaluation of mode 0 must be real-valued as well, but since the argument to the polynomial is zero, this is achieved automatically). Since the packing of the Fourier coefficients forces the row-wise mode $-N/2$ to be ignored, there is only one special case in the Horner algorithm, the evaluation of the column-wise $-N/2$, stored in column $N/2+1$. To identify this case, the loop counting the column-wise powers is unrolled twice, and by restricting $maxdegree$ to be even (since $maxdegree$ is chosen arbitrarily, this is no problem), the second copy of the code in the loop will have to deal with the odd power and thereby the special case.

Upon return from $make\_e$, the real-valued initial solution, kept in ur is computed. The solver assumes that the initial solution is transformed into $[0;1]$ (or a part of this interval), to make the color mapping of the solutions more simple. Depending on the value of u_flag, ur will be set to one of four different vectors, each computed by the four routines: $make\_u\_sin$, $make\_u\_sin2$, $make\_u\_trk$ and $make\_sinr\_r$. $make\_u\_sin$ sets

$$ur_{i+1,j+1} = \frac{2 + \sin\frac{2\pi i}{N} + \sin\frac{2\pi j}{N}}{4}, i = 0, 1, \ldots, N-1, j = 0, 1, \ldots, N-1$$

i.e. one period of a sine that has the offset 1. $make\_u\_sin2$ sets

$$ur_{i+1,j+1} = \frac{4 + \sin\frac{2\pi i}{N} + \sin\frac{2\pi j}{N} + \sin\frac{6\pi i}{N} + \sin\frac{6\pi j}{N}}{8}, \\ i = 0, 1, \ldots, N-1, \; j = 0, 1, \ldots, N-1$$

that is in each direction the sum of one period of $\sin x + \sin 3x$, again offset to make the minimum zero. $make\_sinr\_r$ sets

$$ur_{i+1,j+1} = \frac{1 + \frac{\sin R_{i,j}}{R_{i,j}}}{2}, R_{i,j} = \sqrt{(\frac{4\pi(i-\frac{N}{2})}{N})^2 + (\frac{4\pi(j-\frac{N}{2})}{N})^2},$$
$$i = 0, 1, \ldots, N-1, \; j = 0, 1, \ldots, N-1, \; (i,j) \neq (\frac{N}{2}, \frac{N}{2}), \; ur_{\frac{N}{2},\frac{N}{2}} = 1 \qquad (6.6)$$

which is a 'sombrero', with maximum 1. The forth possibility is $make\_u\_trk$, which produces a 'pyramid', with maximum 1. The four sides of the pyramid are made with the FOR-ALL-construction. This is in this case quite inefficient (the compiler gives a warning about this), but is does not matter, since the computation of the initial solution is only done in order to test the solver. Next thing is to display the initial solution, and, depending on $tiff\_flag$, write it to the TIFF-file on the front end, and to pack it into the complex half sized u. Unless the constant $reverse\_flag$ is set to $CMSSL\_normal$, all Fourier coefficients are bitreversed, and therefore the e-array has to be this as well. For this purpose the arrays nop, $normal$, $bit\_reversed$, and $noscale$ are initialized to make a dummy call to the FFT-routine bit-reversing the e-array. After this, the iterations can be done, and therefore a timer is reset, and the solving loop is entered, the timer started, and the Fourier coefficients of the next solution are computed. A copy is made of these coefficients, and this copy is transformed back to the time-domain using our $realifft2d$. The complex array returned from $realifft2d$ is unpacked into the real-valued ur, and the timer is stopped. Unless $single\_flag$ is zero, the program will wait for a keypress, making it possible to 'grab' each solution with f.ex. a video-recorder (N.B. the current version does not have this feature, $single\_flag$ is set to zero, instead of letting the user input it). Depending on $tiff\_flag$, the solution is either displayed, or appended to the TIFF-file (in this case the displaying is skipped because it is time-consuming, if the net-work-connection to the graphics screen is slow). As the solving loop finishes, the value of the timer is output to the screen, the space allocated to the CMSSL-FFT is freed, and PDE_solver terminates. The user is asked to input another N, and if it has a legal value, another set of input can be given, to produce a new solution. This will go on until N is either non-positive or greater than 1024, which will cause the main loop and thereby the program to terminate.

6.5 A Test of the Program Linpde2d.

Given simple PDE's the program will produce nice sequences of pictures, that behave as expected. The only remark to make is that the solver is not good at solving PDE's that have the pyramid as initial solution. The sharp edges produce a lot of distortion, caused by Gibb's effect; this can especially be seen with differentiation of order three or more. By use of appendix H, we find that the flops used to make i pictures is

$$\frac{3iNM}{2} + iNM(\frac{5}{4}\log_2(NM) + \frac{9}{4})$$

or

$$\frac{5iNM}{4}(3 + \log_2(NM)).$$

Using $N = 1024$ and $i = 40$ takes 18.9 seconds of CPU-time, equal to 64 Mflops/s. Using $N = 128$ instead takes 0.43 CPU-seconds, yielding 32 Mflops/s. The efficiency of the program is however not only described by the floprate. The use of our real-to-complex FFT-routine requires more floprate reducing communication, but reduces the required flops to 50%, which means that we could get a higher floprate using full sized complex-to-complex FFT's, but at the price of an increase in used CPU-time.

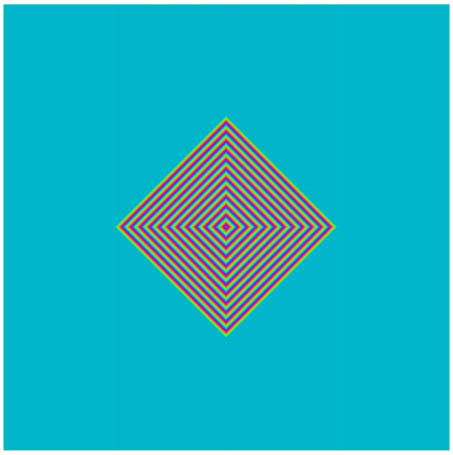

Figure 6.1 The pyramid, $t = 0$.

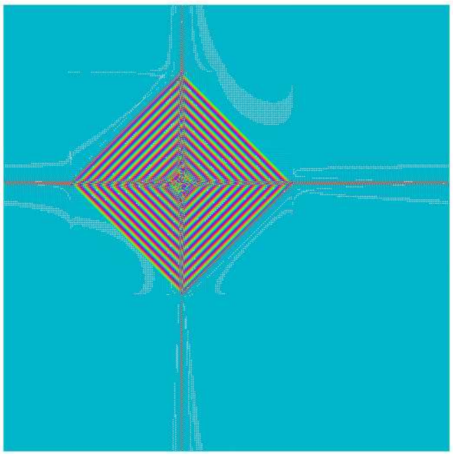

Figure 6.2 The pyramid on the move at $t = 2\pi/10$.

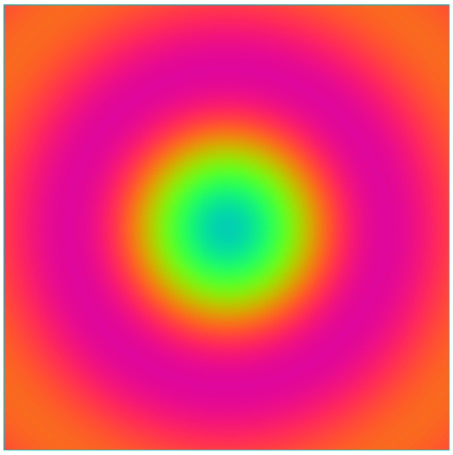

Figure 6.3 The sombrero at $t = 0$.

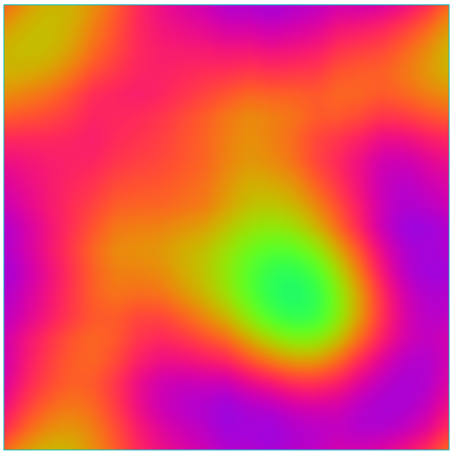

Figure 6.4 The sombrero at $t = 2\pi/5$.

Chapter 7

General one step methods for solving ODEs.

In this chapter we will describe the methods for integrating ordinary differential equations, that we use later when having transformed the PDEs to ODEs. When dealing with nonlinear problems, the restrictions of a given method is often caused by its stability restrictions instead of its asymptotic error rate. Because of this we emphasize on the stability regions of different methods, and in order to be able to compare the different methods easily, we display all the stability regions with the same scaling of the axes. We consider only one step methods because we want the ability to change the step length dynamically while stepping forward in time.

7.1 Euler's Method.

A simple way to solve the initial value problem is by use of Euler's method which is defined by

$$\vec{u}_{n+1} = \vec{u}_n + h\vec{f}(t_n, \vec{u}_n), \quad n = 0, 1, 2, ... \tag{7.1}$$

This can be directly transformed to the Fourier domain,

$$\hat{\vec{u}}_{n+1} = \hat{\vec{u}}_n + h\vec{f}_f(t_n, \hat{\vec{u}}_n), \quad n = 0, 1, 2, ... \tag{7.2}$$

The problems concerning this method are, that it has an asymptotic error of $o(h^2)$ and a very limited stability region as seen in figure 7.1.

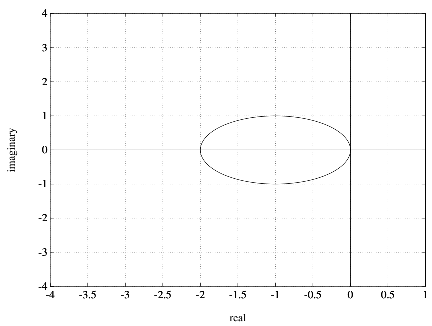

Figure 7.1: Euler stability region.

7.2 Second Order Explicit Runge Kutta.

As an alternative to the simple Euler's method we can use a second order explicit Runge Kutta method.

All the Runge Kutta methods are of the type,

$$\begin{aligned}
\vec{u}_{n+1} &= \vec{u}_n + h \sum_{r=1}^{R} b_r \vec{k}_r \\
\vec{k}_r &= \vec{f}(t_n + c_r h, \vec{u}_n + h \sum_{j=1}^{R} a_{r,j} \vec{k}_j), \quad r = 1, 2, ..., R
\end{aligned}$$

where h is the stepsize, R is the number of terms and $a_{r,j}$, b_j and c_r are constants. Because of this structure it is convenient to use the so-called Butcher array to represent a specific Runge Kutta method, that is the constants $a_{r,j}$, b_j and c_r:

$$\begin{array}{c|c}
c & A_{R,R} \\
\hline
 & b
\end{array}$$

For the second order explicit Runge Kutta that we use we have the Butcher array:

$$\begin{array}{c|cc}
0 & 0 & \\
1 & 1 & 0 \\
\hline
 & \frac{1}{2} & \frac{1}{2}
\end{array}$$

which is equivalent to,

$$\begin{aligned}
\vec{u}_{n+1} &= \vec{u}_n + \tfrac{1}{2} h (\vec{k}_1 + \vec{k}_2) \\
\vec{k}_1 &= \vec{f}(t_n, \vec{u}_n) \\
\vec{k}_2 &= \vec{f}(t_n + h, \vec{u}_n + h \vec{k}_1)
\end{aligned}$$

This can be transformed to the Fourier domain just like Euler's method:

$$\begin{aligned}
\hat{\vec{u}}_{n+1} &= \hat{\vec{u}}_n + \tfrac{1}{2} h (\hat{\vec{k}}_1 + \hat{\vec{k}}_2) \\
\hat{\vec{k}}_1 &= \vec{f}_f(t_n, \hat{\vec{u}}_n) \\
\hat{\vec{k}}_2 &= \vec{f}_f(t_n + h, \hat{\vec{u}}_n + h \hat{\vec{k}}_1)
\end{aligned}$$

The second order Runge Kutta has an error of $o(h^3)$ and a region of stability as shown on figure 7.2.

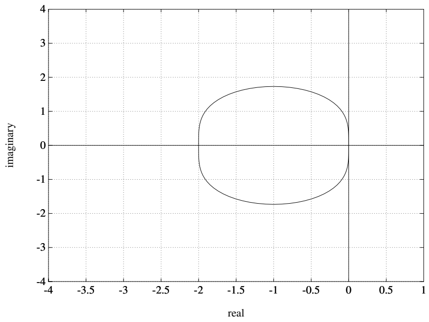

Figure 7.2 RK2 explicit stability region.

7.3 Fourth Order Explicit Runge Kutta.

In order to minimize the error and increase the region of stability we can use the classical fourth order Runge Kutta method,

$$\begin{array}{c|cccc} 0 & 0 & & & \\ \frac{1}{2} & \frac{1}{2} & 0 & & \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

This can be transformed to the Fourier domain just like the previously described second order Runge Kutta and has an error of $o(h^5)$ and the stability region shown on figure 7.3.

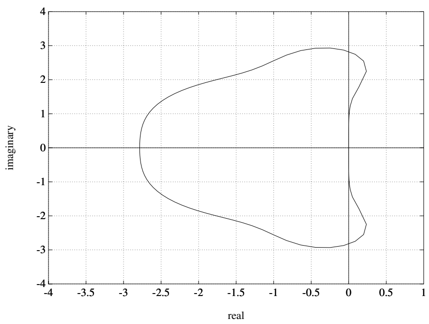

Figure 7.3 RK4 explicit stability region.

7.4 Second Order Semiimplicit Runge Kutta.

The eigenvalues of some problems are very far apart, resulting in a very small step length when using the explicit methods. In order to increase this step length we introduce an semiimplicit RK-method:

$$\begin{array}{c|cc} 0 & 0 & \\ 1 & \beta & 1-\beta \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

As figure 7.4 shows, this has a well suited stability region, depending on beta.

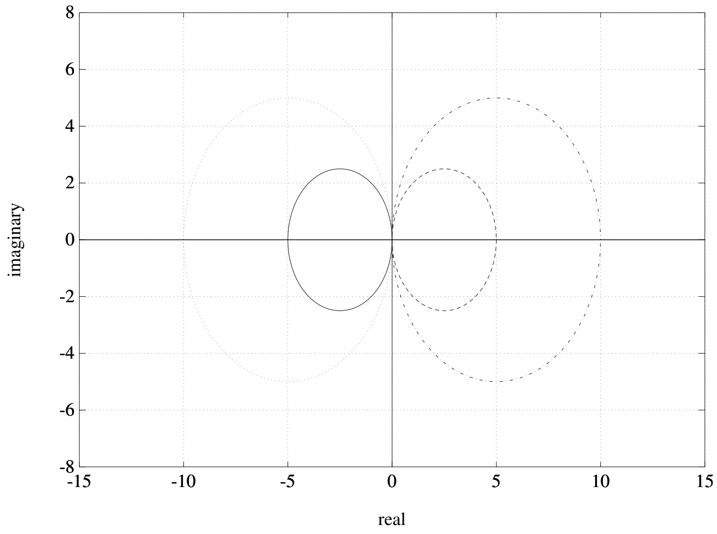

Figure 7.4 Semiimplicit RK2 stability region.
$\beta = 0.7$, inside '——' circle.
$\beta = 0.6$, inside '$\cdot\cdot\cdot$' circle.
$\beta = 0.5$, left halfplane.
$\beta = 0.4$, outside '$- \cdot -$' circle.
$\beta = 0.3$, outside '$- - -$' circle.

The problem involved is though, that $\vec{k}_2$ is defined recursively:

$$\vec{k}_2 = \vec{f}(t_n + h, \vec{u}_n + \frac{1}{2}h(\vec{k}_1 + \vec{k}_2)$$

This means that we have to use an iterative process in order to calculate $\vec{k}_2$,

$$\begin{aligned} \vec{k}_1 &= \vec{f}(t_n, \vec{u}_n) \equiv \vec{f}(T_1, \vec{Y}_1) \\ \vec{k}_2 &= \vec{f}(t_n + h, \vec{u}_n + \frac{1}{2}h(\vec{k}_1 + \vec{k}_2)) \equiv \vec{f}(T_2, \vec{Y}_2) \\ \vec{Y}_2 &= \vec{u}_n + \frac{1}{2}h(\vec{f}(T_1, \vec{Y}_1) + \vec{f}(T_2, \vec{Y}_2)) \qquad (7.3) \\ \vec{F}(\vec{Y}_2) &= \vec{Y}_2 - \frac{1}{2}h\vec{f}(T_2, \vec{Y}_2) - \vec{u}_n - \frac{1}{2}h\vec{f}(T_2, \vec{Y}_2) = \vec{0} \qquad (7.4) \end{aligned}$$

That is, we have to find the 'zero' of $\vec{F}$. A simple way of doing this is to use a fixed point iteration on equation 7.3,

$$\vec{Y}_2^{n+1} = \vec{u}_n + \frac{1}{2}h(\vec{f}(T_1, \vec{Y}_1) + \vec{f}(T_2, \vec{Y}_2^n)), \quad n = 0, 1, \ldots$$

We will later discuss how to choose $\vec{Y}_2^0$. It should be evident that this iteration can be directly transformed to the Fourier domain just like we did for the explicit RK methods.

The problem about the fixed point iteration is, that it often converges very slowly - if it converges at all. In order to speed up the convergence we introduce Newton-Raphson's method,

$$\begin{array}{l} \underline{\underline{J}}_F^{\vec{Y}_2^n} \Delta \vec{Y}^n = -\vec{F}(\vec{Y}_2^n) \\ \vec{Y}_2^{n+1} = \vec{Y}_2^n + \Delta \vec{Y}^n, \quad n = 0, 1, \ldots \end{array}$$

Where $\underline{\underline{J}}_F^{\vec{Y}_2^n}$ is the Jacobi matrix of $\vec{F}$, evaluated at $\vec{Y}_2^n$. Rewriting the Jacobian leads to:

$$\underline{\underline{J}}_F^{\vec{Y}_2^n} = \left\{ \frac{\partial \vec{F}_i}{\partial \vec{Y}_{2,j}} |_{\vec{Y}_{2,j} = \vec{Y}_{2,j}^n} \right\} = \underline{\underline{I}} - \frac{1}{2} h \left\{ \frac{\partial \vec{f}_i}{\partial \vec{u}_j} |_{\vec{u}_j = \vec{Y}_2^n} \right\} = \underline{\underline{I}} - \frac{1}{2} h \underline{\underline{J}}_f^{\vec{Y}_{2,j}^n}$$

The above shown calculations can be directly transformed to the Fourier space. The main difference in whether we are in normal or transformed space is the Jacobian of $\vec{f}$, which we have addressed in a general matter in appendix 8.

We would often expect the Jacobian to change slowly, which encourages one not to calculate a new Jacobian for every time step but instead use an older version of the Jacobian in equation 7.4 and then only calculate a new Jacobian when the Newton-Raphson iteration converges too slowly.

On the Connection Machine the only methods for solving systems of linear equations available at the present time are direct methods. Since these methods basically are sequential, they are not well-suited for a massively parallel computer like the CM. Since our problem furthermore has a coefficient matrix with a dominating diagonal[1], we have reason to believe that an iterative solver would be preferable. We have chosen not to develop a SSOR or CG solver since these will be implemented in a future version of the CMSSL library and since our time is limited.

This leaves us with the direct solvers. The problem with these is that the factorization process is expensive. Due to this and our assumption that the Jacobian varies slowly, we can by use of a simple iterative process avoid factorizing the Jacobian every time we want to solve the system with an non-factorized Jacobian:

$$\begin{array}{ll} (\underline{\underline{J}} + \underline{\underline{E}})\underline{x} = \underline{b} & \Leftrightarrow \\ \underline{\underline{J}}\underline{x} = \underline{b} - \underline{\underline{E}}\underline{x} & \Leftrightarrow \\ \underline{x} = \underline{\underline{J}}^{-1}(\underline{b} - \underline{\underline{E}}\underline{x}) & \end{array}$$

Which leads to a one step iteration:

$$\underline{x}_{k+1} = \underline{\underline{J}}^{-1}(\underline{b} - \underline{\underline{E}}\underline{x}_k), \quad k = 0, 1, \ldots$$

Where a proper initial guess for $\underline{x}$ could be:

$$\underline{x}_0 = \underline{\underline{J}}^{-1}\underline{b}$$

7.4.1 The Choice of Initial Guess for the SIRK2.

A simple choice of initial guess for Y_2 is computed using Euler,

$$\vec{Y}_2 = \vec{u} + h * \vec{k}_1$$

where $k_1 = \vec{f}(t, \vec{u})$.

Alternatively one could use the explicit second order RK as guess of k_2 and then compute Y_2 based on this,

$$\begin{array}{rcl} \vec{k}_2 & = & \vec{f}(t + h, \vec{u} + h\vec{k}_1) \\ \vec{Y}_2 & = & \vec{u} + \frac{1}{2}h(\vec{k}_1 + \vec{k}_2) \end{array}$$

where $k_1 = \vec{f}(t, \vec{u})$.

[1] A least for the a sufficiently small step length.

Another possibility is to use the Jacobian in order to calculate the initial guess,

$$
\begin{aligned}
&\vec{k}_1 = \underline{\underline{J}}_f \vec{u} \\
&\vec{k}_2 = \underline{\underline{J}}_f (\vec{u} + \tfrac{1}{2}h\underline{\underline{J}}_f \vec{u} + \tfrac{1}{2}h\vec{k}_2) \quad \Rightarrow \\
&(\underline{\underline{I}} - \tfrac{1}{2}h\underline{\underline{J}}_f)\vec{k}_2 = \underline{\underline{J}}_f (\vec{u} + \tfrac{1}{2}h\underline{\underline{J}}_f \vec{u}) \Leftrightarrow \\
&\underline{\underline{J}}_F \vec{k}_2 = \underline{\underline{I}}\vec{k}_1 + (\underline{\underline{I}} - \underline{\underline{J}}_F)\vec{k}_1 \Leftrightarrow \\
&\underline{\underline{J}}_F \vec{k}_2 = (2\underline{\underline{I}} - \underline{\underline{J}}_F)\vec{k}_1
\end{aligned}
$$

that is we have to solve a system with a Jacobian that is or approximately[2] is factorized.

7.5 A note on the use of Runge-Kutta methods.

As in section 7.2, the Runge-Kutta methods for numerical integration are described as:

$$
\begin{aligned}
y_{n+1} &= y_n + h\sum_{r=1}^{R} b_r K_r, \\
K_r &= f(x_n + c_r h,\ y_n + h\sum_{j=1}^{R} a_{r,j} K_j)
\end{aligned}
$$

where $y(x)$ is the function to be integrated, in steps of h, where $dy/dx = f(x,y)$, and where $R, \underline{\underline{a}}, \vec{b}, \vec{c}$ defines the method. Since this method always uses K_r as hK_r, and $K_r = f(\vec{a}, b)$, the multiplication hK_r, can be done using hf to calculate K_r. The equations we solve all have f-functions of the form $ag(x,y)$, or $ag(x,y) + bh(x,y)$, where a and b are constants, and therefore the multiplication hf can be done be simply scaling the constants to ah and bh. In cases where a majority of the non-zero elements of $\underline{\underline{a}}$, $\vec{b}$ has the same value, e.g. in the classic fourth order explicit Runge-Kutta, where the constant $1/2$ is common, we adjust the constants and use $ah/2$ and $bh/2$ to save a couple of divisions in the evaluation of K_r. We have done this in all the implementations of Runge-Kutta methods.

[2]Since we use an older factorized Jacobian, the factorization is an approximation of the new Jacobian. This implies that we can use the iterative method described above to solve the system.

Chapter 8

Properties Regarding ODE's Derived from PDE's with one Spatial Dimension, and their Jacobians.

As shown in section 7.4 the Jacobian is often needed in order to be able to make an e.g. Newton-Raphson iteration. We will in this chapter study the eigenvalues of Jacobians in general, the Jacobians of a large nonlinear group of one-dimensional operators, and finally the problem of aliasing.

8.1 The Eigenvalues of the Jacobian.

Theorem 8.1.1 *The eigenvalues of the Jacobian are identical in time and Fourier space.*

$\square$

Proof 8.1.1 *Based on the definition of the discrete Fourier transform it is plain that the N-point forward transform on u can be performed by the matrix-vector multiplication,*

$$
\begin{aligned}
\hat{\vec{u}} &= \underline{\underline{F}}\vec{u} \\
\underline{\underline{F}}_{m+1,n+1} &= e^{-2\pi i \frac{mn}{N}} \\
& \quad m, n \in \{0, 1, \ldots, N-1\}
\end{aligned}
$$

Corresponding considerations for the inverse transform yield

$$
\begin{aligned}
\vec{u} &= \underline{\underline{F}}^{-1}\hat{\vec{u}} \\
\underline{\underline{F}}^{-1}_{m+1,n+1} &= \frac{1}{N} e^{2\pi i \frac{mn}{N}} \\
& \quad m, n \in \{0, 1, \ldots, N-1\}.
\end{aligned}
$$

Calculating the matrix-matrix product between $\underline{\underline{F}}$ and $\underline{\underline{F}}^{-1}$ leads to

$$
\begin{aligned}
(\underline{\underline{F}}\,\underline{\underline{F}}^{-1})_{m+1,n+1} &= \sum_{k=0}^{N-1} e^{-2\pi i \frac{km}{N}} \frac{1}{N} e^{2\pi i \frac{kn}{N}} \\
&= \frac{1}{N} \sum_{k=0}^{N-1} e^{2\pi i \frac{k}{N}(n-m)} \\
&= \begin{cases} 1 & m = n \\ 0 & otherwise \end{cases} \\
& \quad m, n \in \{0, 1, \ldots, N-1\}
\end{aligned}
$$

meaning that $\underline{\underline{F}}$ is regular. If we now take a look at the Jacobian, $\underline{\underline{J}}$, we get

$$
\begin{aligned}
\vec{u}' &= \underline{\underline{J}}\vec{u} \quad \Leftrightarrow \\
\hat{\vec{u}}' &= \underline{\underline{F}}\,\underline{\underline{J}}\vec{u} \\
&= \underline{\underline{F}}\,\underline{\underline{J}}\,\underline{\underline{F}}^{-1}\hat{\vec{u}}
\end{aligned}
$$

where $\underline{\underline{F}}\,\underline{\underline{J}}\,\underline{\underline{F}}^{-1}$ is the Jacobian in the Fourier space. Since F is regular, it is now obvious that the Jacobians in time and Fourier domain, are **similar**. *This means that the eigenvalues of the two Jacobians are identical (see e.g. [8, page 474]).*

$\square$

8.2 The Problem of Aliasing.

When performing the calculation of the convolution sums we have a problem concerning aliasing. As shown in chapter 12 the problem sometimes results in inability to solve the wanted equation.

If we have the two M-point sequences, u and v, that we want to calculate the convolution sums of by means of the discrete Fourier transform, then we get, due to the fact that we use truncated series:

$$
\begin{aligned}
u_j &= \frac{1}{M}\sum_{k=-\frac{M}{2}}^{\frac{M}{2}-1} \hat{u}_k e^{2\pi i\frac{kj}{M}} \\
v_j &= \frac{1}{M}\sum_{k=-\frac{M}{2}}^{\frac{M}{2}-1} \hat{v}_k e^{2\pi i\frac{kj}{M}} \\
w_j &= u_j v_j, \quad j \in \{0,1,\ldots,M-1\} \\
\hat{w}_p &= \sum_{j=0}^{M-1} u_j v_j e^{-2\pi i\frac{jp}{M}} \\
&= \sum_{j=0}^{M-1}[\frac{1}{M}\sum_{k=-\frac{M}{2}}^{\frac{M}{2}-1} \hat{u}_k e^{2\pi i\frac{kj}{M}}][\frac{1}{M}\sum_{l=-\frac{M}{2}}^{\frac{M}{2}-1} \hat{v}_l e^{2\pi i\frac{lj}{M}}]e^{-2\pi i\frac{jp}{M}} \\
&= \frac{1}{M^2}\sum_{j=0}^{M-1}\sum_{k=-\frac{M}{2}}^{\frac{M}{2}-1}\sum_{l=-\frac{M}{2}}^{\frac{M}{2}-1} \hat{u}_k\hat{v}_l e^{2\pi i\frac{kj}{M}} e^{2\pi i\frac{lj}{M}} e^{-2\pi i\frac{jp}{M}} \\
&= \frac{1}{M^2}\sum_{k=-\frac{M}{2}}^{\frac{M}{2}-1}\sum_{l=-\frac{M}{2}}^{\frac{M}{2}-1}\sum_{j=0}^{M-1} \hat{u}_k\hat{v}_l e^{2\pi i\frac{j}{M}(k+l-p)} \\
&= \frac{1}{M}\sum_{k+l=p} \hat{u}_k\hat{v}_l + \frac{1}{M}\sum_{k+l\pm M=p} \hat{u}_k\hat{v}_l, \quad p \in \{-\frac{N}{2}, -\frac{N}{2}+1, \ldots, \frac{N}{2}-1\}
\end{aligned}
$$

where the problem is the second term - the aliasing error. In order to remove this error, we need $\hat{u}_k\hat{v}_l$ to be zero for $\mid k+l-p \mid= M$. Inspired by [9] we can do this by padding with zeroes. If we originally had the $(N+1)$-point sequences $\hat{u}'$ and $\hat{v}'$, then we define:

$$
\hat{u}_k = \begin{cases} \hat{u}'_k & \mid k \mid \le \frac{N}{2} \\ 0 & otherwise \end{cases}
$$

and similarly for v. We are then only interested in the convolution sum, w_p for $|p| \le N/2$ which leads to

$$
\begin{aligned}
&-M < k+l-p < M \Rightarrow \\
&\frac{N}{2} + \frac{N}{2} - -\frac{N}{2} < M \Leftrightarrow \\
&M > \frac{3N}{2}
\end{aligned}
$$

that is we have to perform the discrete Fourier transform on a sequence which is actually 50% longer that the original one - meaning an increase in computer work of a little less that 60% when using FFTs. Since the FFTs available on the CM can only handle powers of 2, we actually have to choose M, and then compute the corresponding maximum N (see table 8.1).

M	N
4	2
8	4
16	10
32	20
64	42
128	84
256	170
512	340
1024	682

Table 8.1: Actual resolution when padding with zeroes.

8.3 Some Properties Concerning the Jacobian.

Theorem 8.3.1 *If*

$$\vec{f} = a\vec{g} + b\vec{h}$$

then we have for the Jacobians:

$$\underline{\underline{J}}_f = a\underline{\underline{J}}_g + b\underline{\underline{J}}_h$$

□

Proof 8.3.1 *If we have the system,*

$$ag(u,x) + bh(u,x)$$

The Jacobian of the N-point discretized system is

$$\begin{aligned} \underline{\underline{J}}_f &= \left\{ \frac{\partial(ag(u,x_i)+bh(u,x_i))}{\partial x_j} \right\}_{i,j=1,2,\ldots,N} \\ &= \left\{ a\frac{\partial(g(u,x_i)}{\partial x_j} \right\}_{i,j=1,2,\ldots,N} + \left\{ b\frac{\partial(h(u,x_i)}{\partial x_j} \right\}_{i,j=1,2,\ldots,N} \\ &= a\underline{\underline{J}}_g + b\underline{\underline{J}}_h. \end{aligned}$$

□

Theorem 8.3.2 *The Jacobian of $u^p \frac{\partial^q u}{\partial x^q}$ is in the time domain,*

$$\underline{\underline{J}}_{l+1,m+1} = \begin{cases} \mathcal{IDFT}((i\vec{k})^q)_{l-m} u_l^p \\ l \neq m \quad \wedge \quad l,m \in \{0,1,\ldots,N-1\} \\ \\ \mathcal{IDFT}((i\vec{k})^q)_0 u_l^p + \mathcal{IDFT}(\mathcal{DFT}(\vec{u}) \odot (i\vec{k})^q)_l p u_l^{p-1} \\ l = m \quad \wedge \quad l,m \in \{0,1,\ldots,N-1\} \end{cases} \tag{8.1}$$

and in the Fourier domain,

$$\begin{aligned} \underline{\underline{J}}_{l+1,m+1} &= v_l v_m \mathcal{DFT}(\mathcal{IDFT}(\vec{v} \odot \vec{\tilde{u}})^{p-1} \odot \mathcal{IDFT}(\vec{v} \odot \vec{\tilde{u}} \odot \frac{p(i\vec{k})^q}{N}) \\ &\quad + \mathcal{IDFT}(\vec{v} \odot \vec{\tilde{u}})^p \frac{(ik_m)^q}{N})_{l-m} \\ &\quad l,m \in \{0,1,\ldots,N-1\} \end{aligned} \tag{8.2}$$

where $\vec{v}$ is the an antialiasing vector that nullifies according to section 8.2.

□

Proof 8.3.2 *In the time domain we get:*

$$
\begin{aligned}
\underline{\underline{J}}_{l+1,m+1} &= \frac{\partial[\mathcal{IDFT}(\mathcal{DFT}(\vec{u}) \odot (i\vec{k})^q) \odot \vec{u}^p]_l}{\partial u_m} \\
&= \begin{cases}
\left[\mathcal{IDFT}(\{e^{-2\pi i \frac{ms}{N}}\}_{s=0,1,\ldots,N-1} \odot (i\vec{k})^q) \odot \vec{u}^p\right]_l \\
l \neq m \;\; \wedge \;\; l,m \in \{0,1,\ldots,N-1\} \\
\\
\left[\mathcal{IDFT}(\{e^{-2\pi i \frac{ms}{N}}\}_{s=0,1,\ldots,N-1} \odot (i\vec{k})^q) \odot \vec{u}^p \right. \\
\left. +\mathcal{IDFT}(\mathcal{DFT}(\vec{u}) \odot (i\vec{k})^q) \odot p\vec{u}^{p-1}\right]_l \\
l = m \;\; \wedge \;\; l,m \in \{0,1,\ldots,N-1\}
\end{cases} \\
&= \begin{cases}
\mathcal{IDFT}((i\vec{k})^q)_{l-m} u_l^p \\
l \neq m \;\; \wedge \;\; l,m \in \{0,1,\ldots,N-1\} \\
\\
\mathcal{IDFT}((i\vec{k})^q)_0 u_l^p + \mathcal{IDFT}(\mathcal{DFT}(\vec{u}) \odot (i\vec{k})^q)_l p u_l^{p-1} \\
l = m \;\; \wedge \;\; l,m \in \{0,1,\ldots,N-1\}
\end{cases}
\end{aligned}
$$

In the Fourier domain we get:

$$
\begin{aligned}
\underline{\underline{J}}_{l+1,m+1} &= \frac{\partial v_l \mathcal{DFT}(\mathcal{IDFT}(\vec{v} \odot \vec{\hat{u}})^p \odot \mathcal{IDFT}(\vec{v} \odot \vec{\hat{u}} \odot (i\vec{k})^q))_l}{\partial u_m} \\
&= \frac{\partial}{\partial u_m}(v_l \mathcal{DFT}(\\
&\quad \left\{ (\frac{1}{N}\sum_{r=0}^{N-1} v_r u_r e^{2\pi i \frac{rs}{N}})^p \frac{1}{N}\sum_{r=0}^{N-1} v_r u_r (ik_r)^q e^{2\pi i \frac{rs}{N}} \right\}_{s=0,1,\ldots,N-1})_l) \\
&= v_l \mathcal{DFT}(\\
&\quad \left\{ p(\frac{1}{N}\sum_{r=0}^{N-1} v_r u_r e^{2\pi i \frac{rs}{N}})^{p-1} \frac{1}{N} v_m e^{2\pi i \frac{ms}{N}} \frac{1}{N}\sum_{r=0}^{N-1} v_r u_r (ik_r)^q e^{2\pi i \frac{rs}{N}} \right. \\
&+ \left. (\frac{1}{N}\sum_{r=0}^{N-1} v_r u_r e^{2\pi i \frac{rs}{N}})^p \frac{1}{N} v_m (ik_m)^q e^{2\pi i \frac{ms}{N}} \right\}_{s=0,1,\ldots,N-1})_l \\
&= v_l \mathcal{DFT}((\frac{p}{N}\mathcal{IDFT}(\vec{v} \odot \vec{\hat{u}})^{p-1} \odot \mathcal{IDFT}(\vec{v} \odot \vec{\hat{u}} \odot (i\vec{k})^q) \\
&+ \mathcal{IDFT}(\vec{v} \odot \vec{\hat{u}})^p \frac{1}{N}(ik_m)^q) \odot \{v_m e^{2\pi i \frac{ms}{N}}\}_{s=0,1,\ldots,N-1})_l \\
&= v_l v_m \mathcal{DFT}(\mathcal{IDFT}(\vec{v} \odot \vec{\hat{u}})^{p-1} \odot \mathcal{IDFT}(\vec{v} \odot \vec{\hat{u}} \odot \frac{p(i\vec{k})^q}{N}) \\
&\quad +\mathcal{IDFT}(\vec{v} \odot \vec{\hat{u}})^p \frac{(ik_m)^q}{N})_{l-m} \\
&\quad l,m \in \{0,1,\ldots,N-1\}
\end{aligned}
$$

where $\vec{v}$ is an antialiasing vector which performs nullification as described in section 8.2.

$\square$

Chapter 9

The Korteweg de Vries Equation.

It all started back in 1834 when an English engineer, John Scott Russell who, moving along a channel, suddenly observed how the sudden stop of a boat generated a large heap of water which moved down the channel. He followed it for one or two miles, where it preserved its shape and gradually diminished, before he lost it. On the basis of this observation Scott Russell initiated a study of the kind of waves which he named waves of translation.

The discussion of the phenomenon continued through the 19th century, and in 1895 the Dutchmen D. J. Korteweg and G. de Vries presented an article in which they derived a mathematical model for waves of the kind Russell had observed. The equation they formulated, now called the Korteweg de Vries equation, was:

$$\frac{\partial u}{\partial t} + \alpha u \frac{\partial u}{\partial x} + \frac{\partial^3 u}{\partial x^3} = 0 \tag{9.1}$$

Among other things, they also showed that the wave with speed V:

$$u(t,x) = \frac{3V}{\alpha} sech^2\left(\frac{\sqrt{V}}{2}(x - Vt)\right) \tag{9.2}$$

satisfied the equation (9.1). The KdV equation, along with the soliton solution is going to be the foundation for the results that we will present in this chapter. We will solve the KdV equation in the time domain as well as in the Fourier domain, and analyze the effect of different time integration methods.

9.1 Transforming the KdV Equation into a System of First-Order ODEs.

We now want to solve the KdV equation with periodic boundary conditions and with the 2π-periodic function g as initial solution:

$$\begin{aligned} \frac{\partial u}{\partial t} &= -\alpha u \frac{\partial u}{\partial x} - \frac{\partial^3 u}{\partial x^3} \\ u(x,0) &= g(x) \\ u(0,t) &= u(2\pi, t), \end{aligned}$$

where we have restricted u to be 2π-periodic in place.

In order to be able to treat the equation numerically, we make a spatial discretization with N points:

$$u(x,t) = u(x_j,t), \;\; x_j = \frac{2\pi j}{N}, \;\; j = 0,1,...,N-1$$

We now make a transformation of the PDE into a system of first-order ODEs by use of the discrete Fourier transform,

Definition 9.1.1

$$\mathcal{DFT}(\vec{f}) = \left\{ \sum_{n=0}^{N-1} f_n e^{-2\pi i n \frac{\omega}{N}} \right\}_{\omega = -\frac{N}{2}, -\frac{N}{2}+1, \ldots, \frac{N}{2}-1}$$

□

in a way similar to the one described in section 5.1:

$$\begin{aligned} \frac{\partial \vec{u}(t)}{\partial t} &= -\alpha \vec{u}(t) \mathcal{IDFT}(i\vec{k} \odot \mathcal{DFT}(\vec{u}(t))) - \\ & \quad \mathcal{IDFT}((i\vec{k})^3 \odot \mathcal{DFT}(\vec{u}(t))) \\ & \quad \vec{k} = \{0, -\frac{N}{2}+1, -\frac{N}{2}+2, \ldots, \frac{N}{2}-1\} \end{aligned} \tag{9.3}$$

Where $\odot$ denotes elementwise multiplication, $\vec{x}^p$ denotes elementwise raise to the power of p and $\vec{u}(t)$ denotes the vector $(u(x_0,t), u(x_1,t), ..., u(x_{N-1},t))$.

Thus, the calculation of $\partial u / \partial t$ requires the use of 3 FFTs.

Alternatively we could calculate $u \partial u / \partial x$ as $\partial u^2 / (2 \partial x)$ yielding:

$$\begin{aligned} \frac{\partial \vec{u}(t)}{\partial t} &= -\alpha \mathcal{IDFT}(\frac{1}{2} i\vec{k} \odot \mathcal{DFT}(\vec{u}(t)^2)) - \\ & \quad \mathcal{IDFT}((i\vec{k})^3 \odot \mathcal{DFT}(\vec{u}(t))) \\ & \quad \vec{k} = \{0, -\frac{N}{2}+1, -\frac{N}{2}+2, \ldots, \frac{N}{2}-1\} \end{aligned} \tag{9.4}$$

which would also require 3 FFTs in order to calculate $\partial u / \partial t$.

If we now make a transformation of the ODEs 9.3 to the Fourier domain (denoted by $\hat{}$) we get:

$$\begin{aligned} \frac{\partial \hat{\vec{u}}(t)}{\partial t} &= -\alpha \hat{\vec{u}}(t) * (i\vec{k} \odot \hat{\vec{u}}(t)) - \\ & \quad (i\vec{k})^3 \odot \hat{\vec{u}}(t) \\ & \quad \vec{k} = \{0, -\frac{N}{2}+1, -\frac{N}{2}+2, \ldots, \frac{N}{2}-1\} \end{aligned} \tag{9.5}$$

where $*$ denotes convolution sum.

Since we have no special routines for calculating convolution sums of two sequences, we would have to use 3 FFTs to do the calculation[1].

[1] Since we could transform the two sequences to the time domain, calculate the product and issue a transform back into the Fourier space.

A transform of the ODE's 9.4 yields:

$$\begin{aligned}
\frac{\partial \hat{\vec{u}}(t)}{\partial t} &= -\alpha(\frac{1}{2} i\vec{k} \odot (\hat{\vec{u}}(t) * \hat{\vec{u}}(t))) - \\
& (i\vec{k})^3 \odot \hat{\vec{u}}(t) \\
& \vec{k} = \{0, -\frac{N}{2}+1, -\frac{N}{2}+2, \ldots, \frac{N}{2}-1\}
\end{aligned} \tag{9.6}$$

Since the convolution sum now has to be performed between two equal sequences, the calculation can be performed by use of only 2 FFTs. This means that we can save roughly 1/3 of the computation time needed to calculate $\partial u/\partial t$ by having transformed it to the Fourier domain and by using $\partial u^2/(2\partial x)$ instead of $u\partial u/\partial x$.

One could argue that the use of $\partial u^2/(2\partial x)$ instead of $u\partial u/\partial x$ is bad, since the squaring of u often results in larger values in the upper Fourier coefficients of u, partly due to the fact that all negative values are made positive, and partly because the steepness of u will increase when squared. We have chosen not to make any conclusions on this matter yet, but instead let the computer results indicate how much of a problem this is.

The solution of the ODE's can be calculated by various methods as shown in chapter 7.

9.2 Solving the ODE's of the Transformed KdV Equation.

We will now address the matter of solving the equations 9.3 and 9.6, since they use the two previously described different methods for calculation of the nonlinear term, and at the same time uses the smallest amount of work possible.

In order to use the standard methods for solving ODE's we need the following formulation, using vector notation for the initial value problem:

$$\vec{u}' = \vec{f}(t, \vec{u}), \quad \vec{u}(t_0) = \vec{U}_0$$

In our case this corresponds just fine with the formulation of equation 9.3 and 9.6 since these actually match the formulation written above. As initial solution we can e.g. use the soliton solution from equation 9.2.

9.2.1 The Matter of Stability.

In order to be able to choose an optimal method of integration, we need to examine the location of the eigenvalues for the system. This is done by calculating the eigenvalues of the Jacobian. Due to theorem 8.1.1, we do not have to analyze the Jacobian in both time and Fourier space. But since we - as discussed earlier - choose to analyze two different methods of calculating of the nonlinear term, namely by $u\partial u/\partial x$ in the time domain and by $\partial u^2/(2\partial x)$ in the Fourier domain, we need to analyze the Jacobian of the systems 9.3 and 9.6.

In the time domain we get, on the basis of equation 9.3 and by utilizing equation 8.1:

$$\underline{\underline{J}}_{l+1,m+1} = \begin{cases} -\alpha \mathcal{IDFT}(i\vec{k})_{l-m} u(t)_l - \mathcal{IDFT}((i\vec{k})^3)_{l-m} \\ l \neq m \quad \wedge \quad l, m \in \{0, 1, \ldots, N-1\} \\ \\ -\alpha \mathcal{IDFT}(i\vec{k})_0 u(t)_m - \alpha \mathcal{IDFT}(\mathcal{DFT}(\vec{u}(t)) \odot i\vec{k})_m \\ +\mathcal{IDFT}((i\vec{k})^3)_0 \\ l = m \quad \wedge \quad l, m \in \{0, 1, \ldots, N-1\}, \end{cases}$$

where u_p denotes element p of $\vec{u}$ starting from 0 and $\vec{k}$ is, as defined earlier: $\{0, 1, ..., \frac{N}{2}-1, 0, -\frac{N}{2}+1, -\frac{N}{2}+2, ..., -1\}$.

When using the term $\partial u^2/(2\partial x)$, we can not utilize equation 8.2, and therefore we get, in the Fourier space, on the basis of equation 9.6, when using the same notation as above:

$$
\begin{aligned}
\underline{\underline{\hat{J}}}_{l+1,m+1} &= \frac{\partial(-\alpha(\frac{1}{2}i\vec{k}\odot(\hat{\vec{u}}(t)*\hat{\vec{u}}(t)) - ((i\vec{k})^3\odot\hat{\vec{u}}(t)))_l}{\partial\hat{u}(t)_m} \\
&\quad l,m\in\{0,1,...,N-1\} \\
&= \frac{\partial(-\alpha(\frac{1}{2}ik_l v_l\mathcal{DFT}(\mathcal{IDFT}(\vec{v}\odot\hat{\vec{u}}(t))^2)_l - ((ik_l)^3\odot\hat{u}(t)_l))}{\partial\hat{u}(t)_m} \\
&\quad l,m\in\{0,1,...,N-1\} \\
&= \begin{cases}
-\alpha(ik_l v_l\mathcal{DFT}(\mathcal{IDFT}(\vec{v}\odot\hat{\vec{u}}(t))\odot \\
\frac{1}{N}v_m\left\{e^{2\pi i\frac{mp}{N}}\right\}_{p=0,1,\dots,N-1}))_l \\
l\neq m \quad l,m\in\{0,1,...,N-1\} \\
\\
-\alpha(ik_m v_m\mathcal{DFT}(\mathcal{IDFT}(\vec{v}\odot\hat{\vec{u}}(t))\odot \\
\frac{1}{N}v_m\left\{e^{2\pi i\frac{mp}{N}}\right\}_{p=0,1,\dots,N-1}))_m - (ik_m)^3 \\
l = m \quad l,m\in\{0,1,...,N-1\}
\end{cases} \\
&= \begin{cases}
-\alpha ik_l v_l v_m\frac{1}{N}\hat{u}(t)_{l-m} \\
l\neq m \quad l,m\in\{0,1,...,N-1\} \\
\\
-\alpha ik_m v_m\frac{1}{N}\hat{u}(t)_0 - (ik_m)^3 \\
l = m \quad l,m\in\{0,1,...,N-1\},
\end{cases}
\end{aligned}
$$

where $\vec{v}$ is an antialiasing vector as described in section 8.2.

We have chosen to use three functions as test functions for the equation. First of all, we have chosen the soliton 9.2 since this is a stationary solution. Secondly, we have chosen a cosine since this is smooth and will split up into solitons. Thirdly, we have chosen a triangle since this has sharp edges, which could give problems with aliasing when splitting up into solitons.

Our numerical experiments have shown that the above functions all have their eigenvalues located on the imaginary axis. This is expected for the soliton but not for the other functions. There is, however, no general tendency toward the eigenvalues being located solely on the imaginary axis since we have created several 'random' functions which resulted in eigenvalues elsewhere.

The location solely on the imaginary axis makes it very easy for us to pick the exact eigenvalue which would tend to give stability problems, since we just have to locate the eigenvalue farthest away from zero. Using this leads to table 9.2, which is computed using antialiasing. It is remarkable how close to the same maximum absolute eigenvalue all the tested functions get when the spatial resolution increases[2]. It seems to make almost no difference (less than 1%) on the location of the maximum absolute eigenvalue for high spatial resolution when we do not use antialiasing[3] (see table 9.1). The reason for this is that the representation of the high modes are very limited[4].

Using the different methods described in chapter 7 leads to - on the basis of the soliton column in table 9.1 - the maximum step length listed in table 9.3.

[2] The eigenvalues for the soliton and the cosine are the same for the calculations, but the maximum absolute eigenvalue of the triangle approaches the maximum absolute eigenvalues of the soliton.

[3] Which is done by setting $\vec{v} = (1, 1, \dots, 1)$.

[4] even for the triangle when the spatial resolution is high.

N	Maximum absolute eigenvalue. Soliton	Cosine	Triangle
8	2.70E+1	2.70E+1	3.18E+1
16	3.40E+2	3.40E+2	2.96E+2
32	3.38E+3	3.38E+3	3.19E+3
64	2.98E+4	2.98E+4	2.91E+4
128	2.50E+5	2.50E+5	2.48E+5
256	2.05E+6	2.05E+6	2.04E+6

Table 9.1: Eigenvalues of testfunctions for KdV equation with aliasing ($\alpha = 1 \quad v = 1$).

N	Maximum absolute eigenvalue. Soliton	Cosine	Triangle
4	7.24E-1	1.00E+0	5.00E-1
8	-2.19E+1	2.71E+1	2.59E+1
16	3.31E+2	3.43E+2	3.41E+2
32	-3.35E+3	3.38E+3	3.37E+3
64	2.97E+4	2.98E+4	2.98E+4
128	-2.50E+5	2.50E+5	2.50E+5
256	-2.05E+6	N/A	N/A

Table 9.2: Eigenvalues of testfunctions for KdV equation without aliasing ($\alpha = 1 \quad v = 1$).

N	Maximum step length. Euler	ERK2	ERK4	SIRK2 ($\beta = \frac{1}{2}$)
8	0	2E-2	1.0E-1	∞
16	0	2E-3	7.9E-3	∞
32	0	2E-4	8.0E-4	∞
64	0	2E-5	9.1E-5	∞
128	0	2E-6	1.1E-5	∞
256	0	3E-7	1.3E-6	∞

Table 9.3: Maximum step length for various methods ($\alpha = 1 \quad v = 1$) without anti-aliasing.

9.3 Implementation of the Solver.

Construction of a program to solve the KdV equation is quite straight forward. The only place where the code becomes a little bit tricky is around the iterative process, when using the SIRK2 method for the time-integration, and at the same time using bit-reversal on the FFT. Due to this, we have chosen not to spend any time explaining the easily understandable parts of the program as well as explaining the programs for the explicit integration - the comments in the programs should be enough.

The interesting part of the program listed in appendix G[5] is the subroutine wave_f_sirk2, which performs the time integration using the semiimplicit second order Runge Kutta described in section 7.4.

The subroutine starts by setting up various vectors to be used with bit reversal etc.. Then it calculates and factorizes[6] the Jacobian of the initial value. Based on this, the initial guess for the iterative process is calculated, and the Jacobian of this system is calculated and factorized. Then the main loop is commenced. The reason why we programmed this 'double' factorization is that we want a Jacobian when entering the main loop so that we only have to recalculate the Jacobian as described in section 7.4.

The main loop starts by calculating the initial guess for the current time step. Then the Newton-Raphson iteration begins. When having performed ten[7] steps in the iteration we recalculate the Jacobian and continue.

When solving the system 7.4, we do it with the one step iteration explained in section 7.4 unless we have just factorized the Jacobian. When we have taken $N/2$ steps[8] with this iteration we refactorize the Jacobian. The iterative process terminates when the maximum absolute change in $\Delta\vec{Y}$ (*dY2*) between two successive steps, relative to the largest absolute element of $\Delta\vec{Y}$, is smaller than ϵ (*epsi*). The reason why we do not use the maximum absolute relative change in $\Delta\vec{Y}$, is that it would be performance inefficient, since we would have to test for division by zero or almost zero.

After this we calculate $\vec{Y}_2$ (*Y2*) and terminate if the maximum absolute change in $\vec{Y}_2$ relative to the maximum absolute element of $\vec{Y}_2$ is smaller than ϵ. We do not use the relative change for the same concerns as above.

When having computed $\vec{Y}_2$ we do a time step and recommence the main loop.

[5] Which is the program for solving the initial value problem for the *extended* Korteweg de Vries equation described in chapter 10 but this is only expanded slightly compared to the version only capable of handling the normal KdV equation as described in section 10.2.

[6] Using the only sensible available solver, the QR-solver, which tends to introduce very little error because Q is orthogonal. This makes the method very well qualified for solving ill-conditioned systems.

[7] An arbitrary guess of what would be reasonable in proportion to the computer work needed to recalculate the Jacobian. It is hard to say anything exact since both the calculation of the Jacobian as well as one iteration are of order $O(N^2)$.

[8] Again an arbitrary guess of when it is reasonable to refactorize the Jacobian. Again an iteration has an order of magnitude, $O(N^2)$ and the factorization is of order $O(N^3)$.

We have chosen to make the Fourier domain our bit-reversed domain, since the time domain is unsuited for this. due to our often displaying pictures[9] in this domain. Working in a bit-reversed domain usually gives no problems, since elementwise operations, matrix-matrix and matrix-vector multiplication along with linear system solvers work as in a non bit-reversed domain if we just remember to keep everything bit-reversed. If we bit-reverse the Fourier coefficients of our initial value along with all constant vectors and matrices, everything else will also get bit-reversed except for the Jacobian, meaning that the only thing to be handled separately is the correct bit-reversal of the Jacobian. The Jacobian needs to be calculated as described in section 9.2.1. The columns of the Jacobian are automatically bit-reversed because of $\hat{\vec{u}}$ (*U, subroutine fjacobi*) being bit-reversed. The bit-reversal of the rows is performed by doing indexing by a bit-reversed index vector[10]. The circular shift is done by indexing with a vector which performs a correct CSHIFT in the bit-reversed domain. This vector is calculated by transforming an index vector from the bit-reversed domain to the normal domain, performing a normal CSHIFT on this vector and then transforming it back into the bit-reversed domain. The elements on the diagonal of the Jacobian will also get bit-reversed but will still reside on the diagonal because when bit-reversing the x-index and y-index of a diagonal element, we still get a diagonal element at the bit-reversed location. The calculation of the Jacobian in the bit-reversed domain is done in the subroutine, fjacobi.

9.4 Solving the Initial Value Problem for the KdV Equation.

When solving the initial value problem for the KdV equation, the matters of interest are in our case how the solution reacts on different initial values, how well the different methods of integration work as well as how large the effect of aliasing is and at last how well we utilize the Connection Machine. In this section we will discuss these issues.

9.4.1 The Reaction to Different Initial Values.

The above described program makes it possible to solve the KdV equation with various initial functions and parameters.

When applying a single soliton, it displaces nicely with a constant velocity, preserving its initial shape. We will comment on the errors introduced in subsection 9.4.3.

When applying a set of solitons to the equation one can se how they displace at their individual speed. Thus one will also see how the faster overtakes the slower one and continues with unaffected speed. The following pictures show how two solitons interact when one moves with the speed 5 and the other moves with the speed 10[11]:

[9] Which naturally can not be shown bit reversed.
[10] An index vector being a vector with FORTRAN indexes in incremental order, $1, 2, \ldots$.
[11] Solved with $\alpha = 1$.

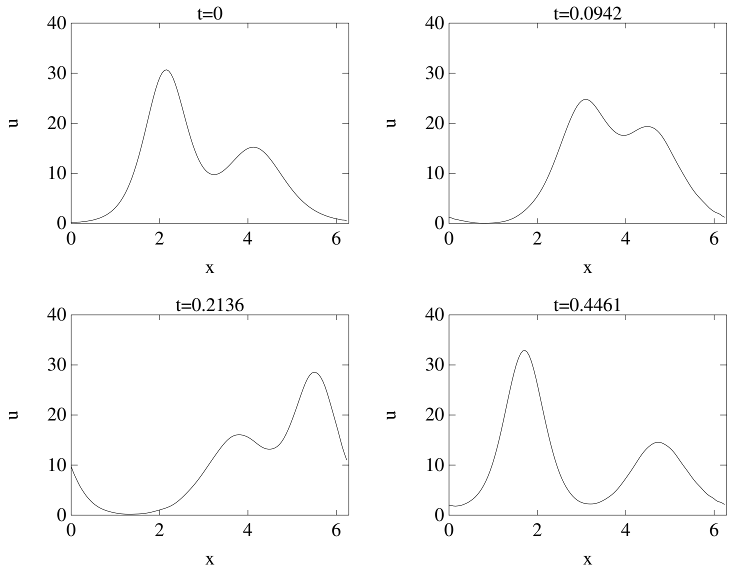

Figure 9.1 Two solitons interacting.

The following color plot shows a similar run with three initial solitons. The color plot shows a t-x plot of the solitons interacting. The different colors represent different amplitudes according to the color bar at the buttom of the figure. The smallest value is at the left end of the color bar.

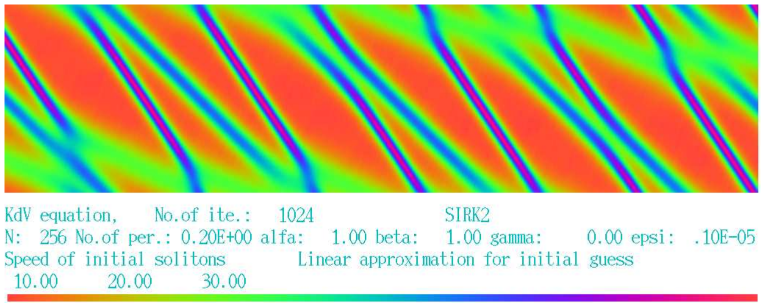

Figure 9.2 Color plot of 3 solitons interacting.

One can see how a faster soliton overtakes a slower one and continues with a shift in phase. The overtaking soliton accelerates while passing and it almost looks as if the two solitons exchanges places.

If instead of applying solitons to the equation, one puts in other functions, these will split up into solitons with different speeds. The following pictures shows how a cosine splits up into solitons. The more unlinear the equation gets, the more solitons the input function gets divided into. The sequence here shown has $\alpha = 20$.

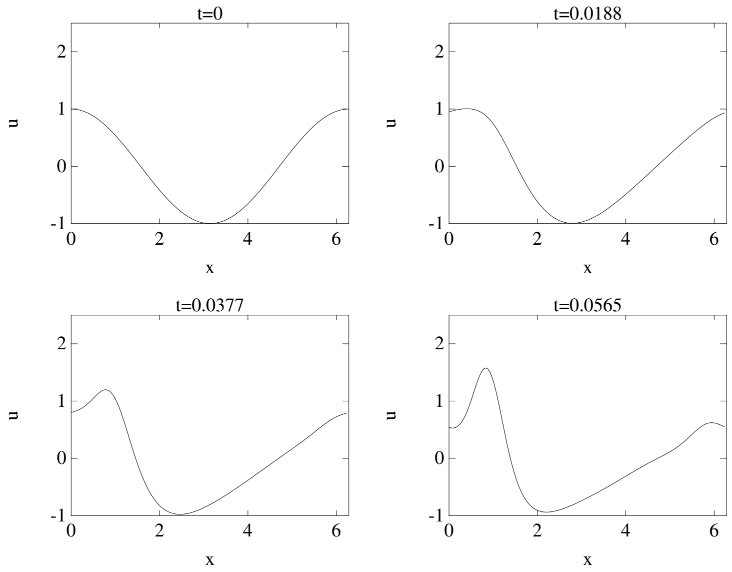

Figure 9.3 Cosine getting split up into solitons.

The cosine 'raises' to a certain point and then oscillations are begun slightly behind the front. These oscillations are caused by the third derivative [6], and they increase until the cosine is completely divided into solitons, each displacing at their own unique speed.

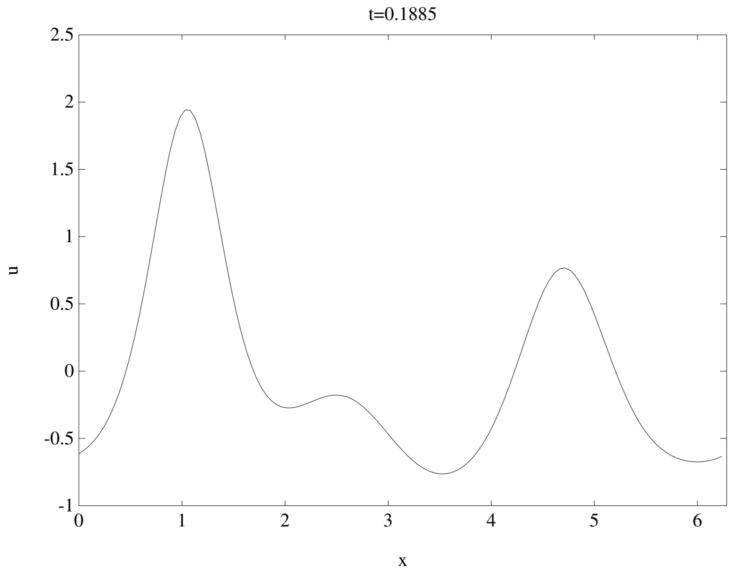

Figure 9.4 Cosine completely lost in solitons.

These solitons all interact as described above and from time to time the cosine is regenerated (possibly with a different phase).

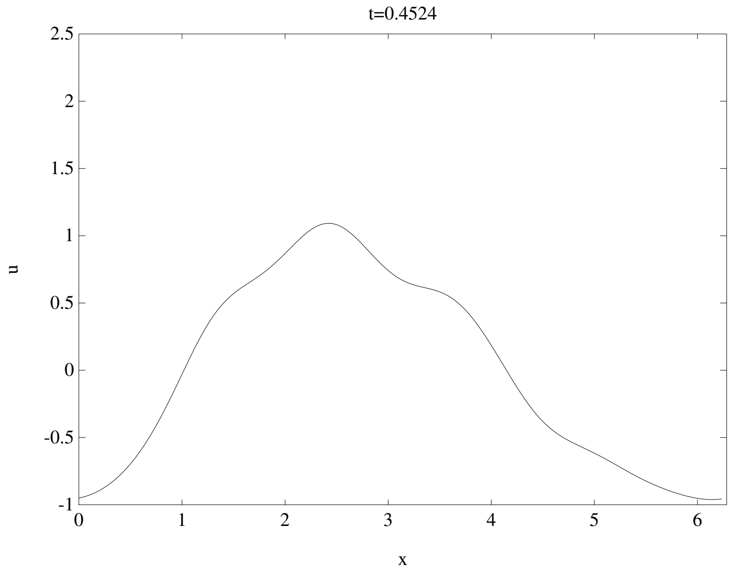

Figure 9.5 Almost cosine again.

The following color plot shows the splitting up of a cosine with $\alpha = 50$. It is remarkable how steady the speed of the individual solitons is.

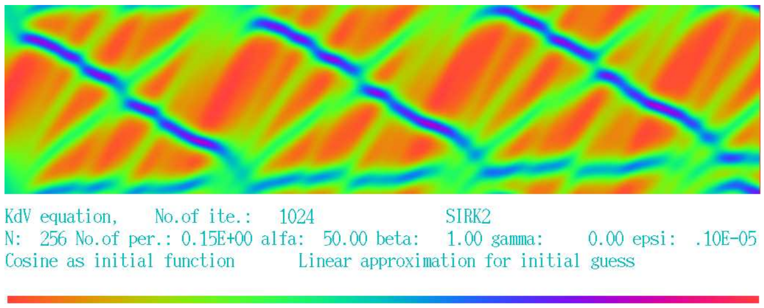

Figure 9.6 Solitons from cosine displacing a unique speed.

In order to see the effects of aliasing and Gibbs it could be interesting to apply a function having a reasonable amplitude at high modes. So we made a triangle and applied it to the equation with $\alpha = 10$.

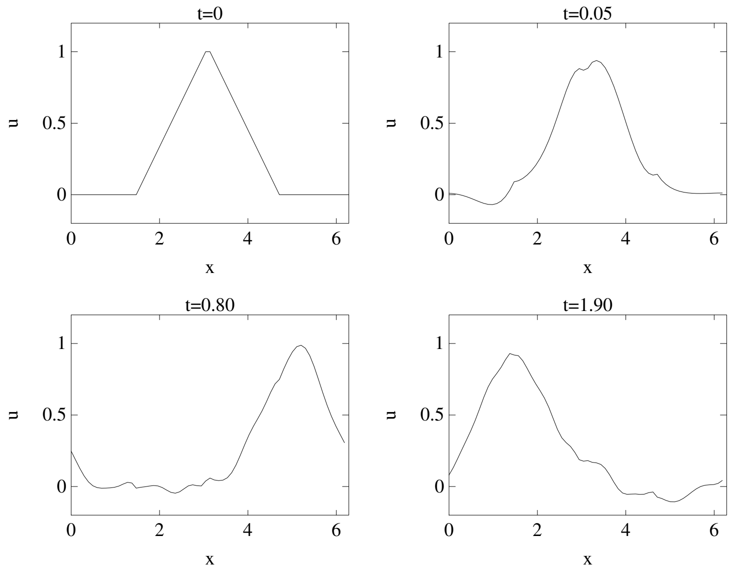

Figure 9.7 Triangle getting split up into solitons.

One can easily see that it is hard for the spectral method to represent the sharp edges, but one can follow the displacement of the triangle, which does not get unstable.

9.4.2 The Difference in using the Operators $u\partial u/\partial x$ and $\partial u^2/(2\partial x)$.

As discussed in section 9.1 we can reduce the computation time when using $\partial u^2/(2\partial x)$ instead of $u\partial u/\partial x$. Table F.1 shows some results using the two different operators. It seems to make no difference on the error, no matter which operator is used. This seems reasonable, since the effects discussed at the end of section 9.1 only become relevant when applying functions that just barely are representable with the given spatial resolution. Because of this we will - from now on - only use the solver using the operator $\partial u^2/(2\partial x)$, and thereby save roughly 1/3 of the computation work.

9.4.3 The Error Rate of the Different Runge Kutta Methods.

In order to analyze the error of the different methods, we must first choose an appropriate way of measuring the errors done by the computation, the rounding error and the truncation error. Since 9.2 is a stationary solution to the KdV equation, we can insert this with an appropriate choice of parameters as the initial value, perform a number of time steps and then compute the error on the basis of knowing how far the soliton actually should have travelled. As a consequence of 9.2 not being periodic, we either have to perform a few time steps with a high velocity, since this means that the error introduced by comparing with 9.2 evaluated at the correct time is negligible, because of the soliton being tall and slim and having boundaries very close to zero, or we have to let the soliton displace itself one or more periods.

The ratio between the largest and smallest element of the discretized soliton is large in both time and Fourier domains, which implies that the use of a relative error measurement will blow up the less important errors on the small elements, since the rounding error on the large element will influence the small elements when performing the discrete Fourier transforms. Therefore, we will instead use the error relative to the largest element as an expression of how well the algorithm and code work.

If we want to verify the order of the truncation error, we need to compute the error inflicted by one time step, meaning that we would have to choose an adequate high velocity, which requires a reasonable high spatial resolution in order to represent the wave. This unfortunately leads to very small step lengths for the explicit method. Tables F.2, F.3 and F.4 shows a single time step with different step lengths[12] for the various Runge Kutta methods implemented. The rounding error seems to dominate the figures for the explicit methods as a result of the step length being small. For the semiimplicit method, a log-log plot of the figures in table F.4 along with ah^3, where a is chosen to fit the data set, is shown in figure 9.8.

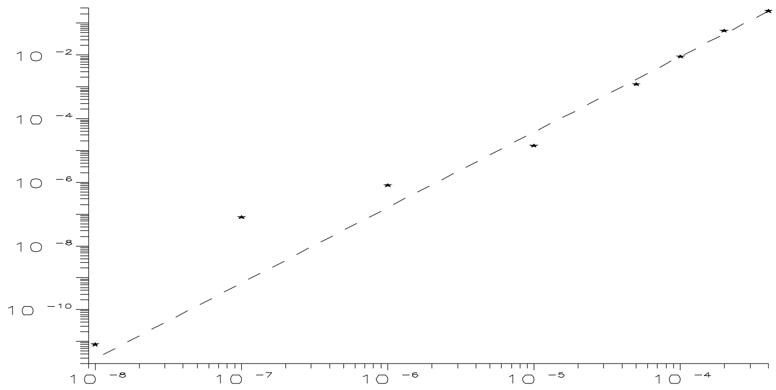

Figure 9.8: Error for semiimplicit Runge Kutta.

One can easily see that the rounding error is dominant when the step length is small, and that the truncation error 'takes over' when the step length increases. Thus for a sufficiently large step length, the actual error shows the truncation error, which is seen to be of order $o(h^3)$ as stated in section 7.4.

[12]These step lengths all fulfill the stability restrictions described in section 9.2.1

Let us now instead take a look at the global error. Table F.5 shows the global error on a soliton of speed 10 after it has traveled one period. For the explicit methods the global error is almost constant. The reason for this must be the error introduced by the small spatial resolution, since the error on the spatial derivative is quite large as a result of the Fourier series being short. This quite constant global error therefore indicates the least possible with the given spatial resolution and soliton. For the semiimplicit method the errors are, due to the large step length, larger than the above perceived minimum. The global error seems in this case to be proportional to h^2. The reason why it is not proportional to h^3 is that we are now looking at the accumulated truncation error.

9.4.4 The Effect on Accuracy when Changing the Spatial Resolution.

In order to analyze the effect of changing the spatial resolution, we have computed the error using a single step and various spatial resolutions. The results presented in table F.7 and shown on figure 9.9 clearly show that the error decreases dramatically when the spatial resolution exceeds a certain value. This makes sense, since the Fourier series suddenly becomes able to represent the soliton correctly which leads to a significant error reduction.

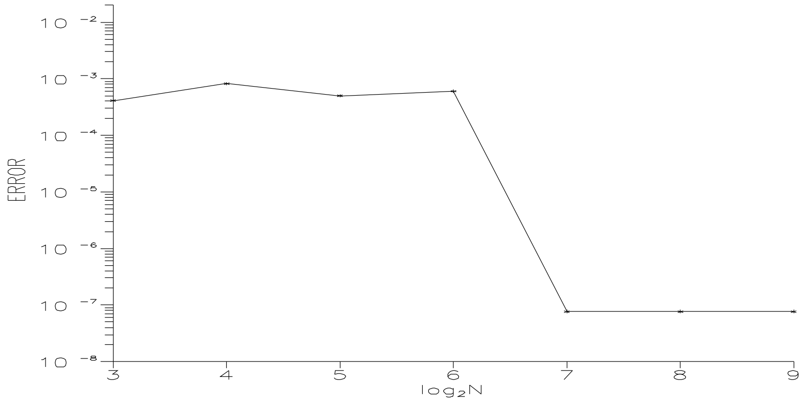

Figure 9.9: Spacial resolution vs. error.

9.4.5 The Stability of the Runge Kutta Methods.

In subsection 9.2.1 along with chapter 7, we applied weak stability theory to the RK methods, and thereby found the maximum step length for the RK methods we have used, when using different spatial resolution. In this subsection we will try to verify this analysis.

We have tried to use the Euler rule, and as expected it turned out that we could not solve the initial value problem, no matter how small we made the step length.

The measurements on the Runge Kutta methods are presented in table F.6 and as expected they confirm the results of section 9.2.1, which indicates that the area containing the eigenvalues does not expand during the displacement of the soliton. It is naturally not possible to use an arbitrarily large step length when using the semiimplicit RK. The problem is no longer stability though, but instead convergence in the iterative process.

The results clearly show a problem with the explicit methods, namely that they need far to many steps in order to solve the initial value problem when the spatial resolutions increases above 64 points. This justifies the computational overhead of a single step with SIRK2 with respect to the explicit methods, since the step length can be increased significantly when using the semiimplicit method.

In this case we actually would not have been able to solve the problem with spatial resolution of 256 and above, if we had not introduced the semiimplicit method.

9.4.6 The Effect of using Antialiasing.

We have computed the effect on the error when not using antialiasing. The results are presented in table F.8, and they lead one to believe that the effect of aliasing on the error is very limited for the KdV equation, since they result in approximately the same errors as the similar runs with antialiasing do.

As described in section 9.2.1 the maximum absolute eigenvalue increase when we do not use antialiasing. This increases the stability problem when using the explicit methods. The effect is documented in table F.9, which shows that the maximum step length decreases significantly in proportion to the maximum step length of the antialiased runs. This can not solely be caused by the initial location of the eigenvalues but must also be caused by their movement and the wrongly computed convolution sums.

9.4.7 The Parameters of the Semiimplicit Runge Kutta.

We started to use the second order semiimplicit RK without Newton-Raphson iteration, and it turned out that the convergence was very slow. After having implemented the Newton-Raphson iteration, the convergence increased tremendously since this method has quadratic convergence. There is no doubt whatsoever that the overhead required to perform an iteration with the Newton-Raphson algorithm in respect to the normal one step iteration, pays. Due to this, we will not discuss the semiimplicit RK without Newton-Raphson iteration any further; instead we will address the matter of choosing the appropriate values for the Newton-Raphson iteration.

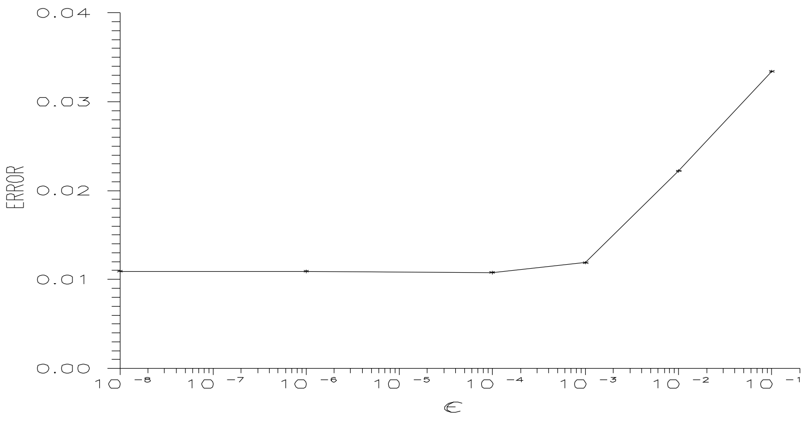

Figure 9.10: ϵ vs. error.

When using the semiimplicit second order Runge Kutta described in section 7.4, we have to make a choice of ϵ, which is used in the stop criteria for the iterative process. Figure 9.10 shows, on the basis of table F.10, the error rate after one period of displacement for various values of ϵ. One can see that the error decreases with ϵ until a certain limit, but that the overall error is small, even for ϵ quite large. This speaks in favor of choosing ϵ large, since the number of iterations required per time step increases quickly when ϵ decreases, as shown in table F.10.

We also have to choose a method for calculating the initial guess for the iterative process. As described in subsection 7.4.1 we can do this by the use of an Euler step, a second order explicit RK step or by a linear approximation on the basis of the Jacobian. Table F.11 shows the number of steps required in the iterative process, for the different methods of calculation of the initial guess. Surprisingly, it turns out that the second order RK results in divergence, even when the other methods converge quite quickly. It also shows that the linear approximation results in a faster convergence than Euler does, and since the computational work required to calculate the initial guess using the linear approximation corresponds to one step in the Newton-Raphson iteration, the linear approximation is preferable to Euler when it results in a reduction in the number of iterations of more than one per time step, which seems to be the case when performing more than a few iterations per time step.

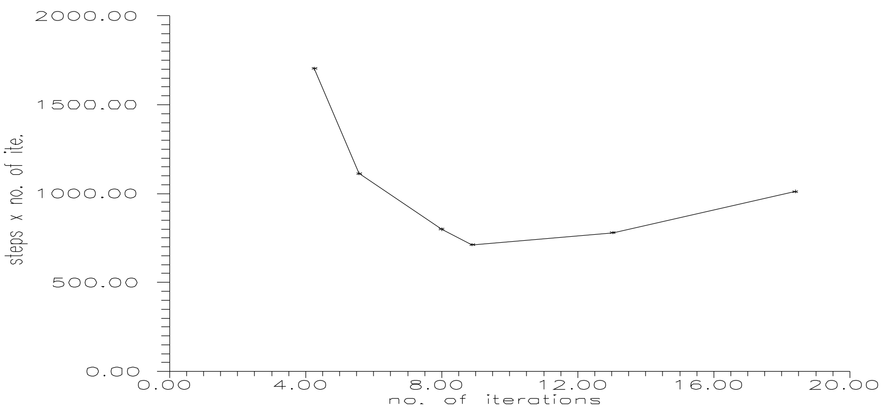

Figure 9.11: Computation cost as a function of average no. of iterations.

Different step lengths result in different number of iterations per time step for a given problem, which means that there exists an optimum no. of iterations (or an optimum step length), when one wishes to minimize the computation time. Based on table F.12, figure 9.11 shows a plot of the average number of iterations needed to perform a time step, versus the product between the number of steps required to do the computation and the average number of iterations. The product gives a reasonable measure of the computation cost, but is far from accurate since it does not show the number of refactorizations performed.

The figure clearly shows that the optimum is around 8 iterations per time step. Since we chose to recalculate the Jacobian whenever 10 iterations had been performed within a single time step, the optimum of an average number of iterations of around eight means that the Jacobian will only be recalculated a few times in the computation.

9.4.8 The Efficiency of the Programs.

In order to fully utilize the Connection Machine, we need to keep all PE's busy simultaneously. When solving 1D problems, this implies the use of vectors of length 1024[13] or more when using the CM present at UNI•C. Since a spatial resolution of 1024 or more normally is unnecessary, the use of the CM will not be optimum. The use of 256 spatial points will thus only use one fourth of the processing power available which could still be acceptable when one wishes to solve a given set of equations.

[13] Since this would keep all PE's busy and satisfy the vector length of four under the Slicewise model.

The essence of this implies that the use of explicit methods for the KdV equation is senseless since a spatial resolution of 256 points would require far to many steps in proportion to the SIRK2 algorithm even though the cost of a single step with SIRK2 is a lot more expensive than one step with an explicit method.

The problem about the semiimplicit method when used with the Newton-Raphson iteration is that we have to solve a system of linear equations which, with the direct methods being the only currently available, is bound to be a sequential process and thus inefficient. Because of this the efficiency of the program is likely to increase if we use an iterative solver or a solver specifically optimized for the Jacobian of the KdV equation.

9.4.9 Concluding Remarks Regarding the Solution of the Initial Value Problem for the KdV Equation.

The main conclusion on solving an initial value problem for the KdV equation is that if we want a reasonable spatial resolution (around 256 points), then the explicit methods are completely unsuited, since it would cost approximately 400 CPU hours to solve the problem where a soliton has to displace itself one period when using ERK4. The semiimplicit RK is a lot better, since it can solve the problem using a little less than 5 CPU minutes.

Furthermore, it is notable that using Euler for the initial guess in the Newton-Raphson iteration is significantly better than using the second order RK for calculation of the initial guess. One should also notice that the measurements show that the optimum number of iterations to perform for every time step seems to be around a dozen and that the effect of having a weak stop criteria (a small ϵ) for the iteration does not seem to make a big difference. Al though these results are not thoroughly verified, they seem to show an interesting tendency.

Chapter 10

Extending the KdV Equation.

As described in the previous chapter the KdV equation can be used for modeling e.q. shallow water waves, but Hasimoto (see [13]) has shown that when taken into account the effect of surface tension then the KdV equation 9.1 for critical values of surface tension must be extended with a fifth order derivative term in order to model the problem with a fair precision. The so called extended KdV equation (shortly EKdV eq.) is

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + \beta\frac{\partial^3 u}{\partial x^3} + \gamma\frac{\partial^5 u}{\partial x^5} = 0. \tag{10.1}$$

This equation is much harder to handle analytically than the ordinary KdV equation and because of this the amount of written material on the subject is very limited.

We will in this chapter show how the EKdV equation reacts when exposed to different combinations of γ and soliton amplitude.

10.1 The Existence of Solitons for the EKdV Equation.

Kawahara concludes in [13] that solitons always exist for the extended KdV equation in the following cases,

$$\gamma\beta > 0, \;\; V\beta > 0, \;\; \epsilon < \epsilon_{max} \approx 2, \tag{10.2}$$

$$\gamma\beta > 0, \;\; V\beta < 0, \;\; \epsilon > \frac{1}{2}, \tag{10.3}$$

$$\gamma\beta < 0, \;\; V\beta > 0, \tag{10.4}$$

where $\epsilon = (|\gamma V|\beta^{-2})^{1/2}$ and V is the velocity of the soliton and ϵ_{max} is a constant close to 2, which Kawahara has found numerically.

V.I. Karpman has proved that 10.2 is incorrect and that solitons therefore can only exist when 10.3 and 10.4 holds, but the dispute has not been settled yet.

10.2 Implementation of the Initial Value Solver.

Adding the fifth order derivative term in the implementation is easily done in the program doing the time integration in the Fourier domain. All there has to be done is to extend the vector *elin* so that it also contains the fifth order derivative. This means that the computational work needed to compute the solution to the initial value problem is unchanged.

As initial value for the computation we use the soliton 9.2 modified so that it includes β as described in [6]. In order to avoid complications when using the square root we apply the absolute value prior to using the square root in equation 9.2.

The code for the program is shown in appendix G.

10.3 Computational Results for the Extended KdV Equation.

The soliton described in equation 9.2 is not a correct soliton for the EKdV equation but since we expect the soliton for the extended equation to be very much like the KdV soliton we use the KdV soliton as initial value and then expect it to distort a little.

We will now describe and comment on some of the results computed by our program. We will often use color plots showing the computational results along with a color bar.

10.3.1 The case $\gamma\beta > 0, \quad V\beta > 0$.

In this case Kawahara claims that solitons do exist for ϵ sufficiently small but Karpman disagrees and claims that solitons never exists.

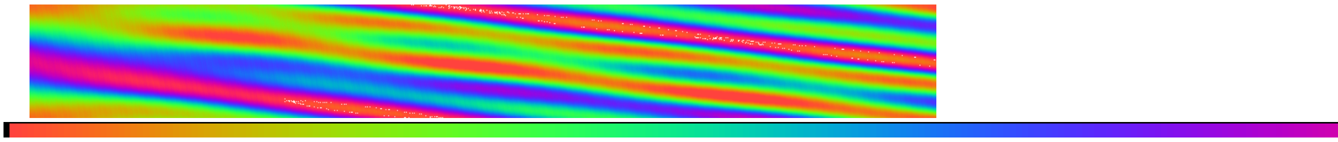

Figure 10.2: t-x color plot for the extended KdV equation. $\beta = 2$, $\gamma = 0.2$, $V = 1$ and the time interval is $[0; 4\pi]$

Figure 10.1 shows the results of computations having $\epsilon = 0.32 \ll \epsilon_{max}$. In this case Kawahara claims that solitons should exist. As seen by the t-x plot this is clearly not the case, since it is easily seen that the KdV soliton almost instantly splits up into waves.

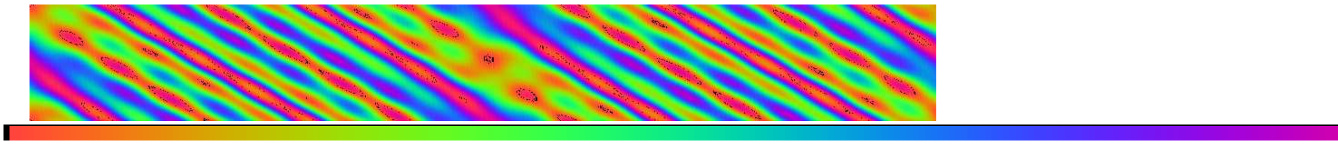

Figure 10.1: t-x color plot for the extended KdV equation. $\beta = 1$, $\gamma = 0.1$, $V = 1$ and the time interval is $[0; 10\pi]$

Figure 10.2 likewise shows how the KdV soliton is split up into waves thereby showing another example of the nonexistence of solitons for $\epsilon = 0.22$. One should notice that the t-x plot shows that the KdV soliton displaces for a short while without breaking up into waves.

When increasing ϵ we e.g. get the figure 10.3, which barely shows the existence of waves instead of a single soliton. This shows that one should be careful when interpreting the figures and concluding that they actually show a soliton[1] since it, as shown above, might begin to dissolve later.

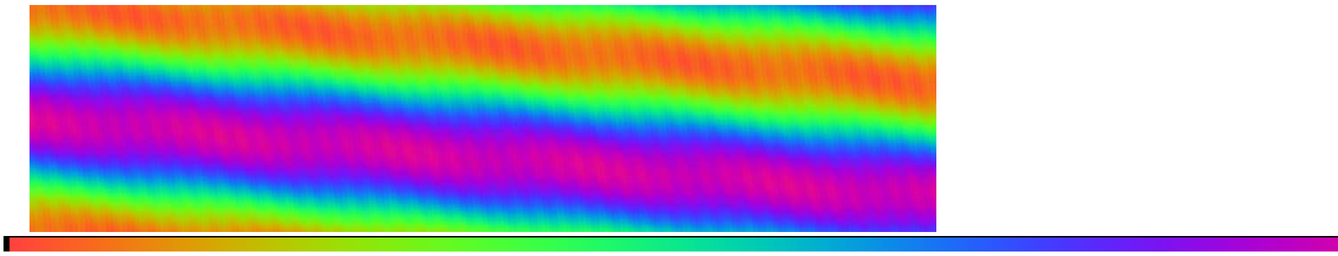

Figure 10.3: t-x color plot for the extended KdV equation. $\beta = 1$, $\gamma = 1$, $V = 1$ and the time interval is $[0; 0.4\pi]$

[1] The judgement of it being a soliton can be made on the basis of preservation of shape and velocity.

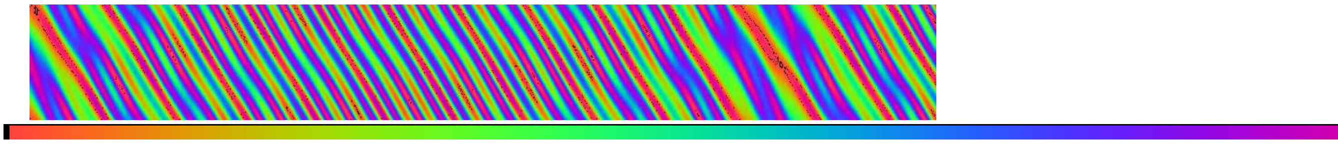

Figure 10.5: t-x color plot for the extended KdV equation. $\beta = -1$, $\gamma = -0.1$, $V = 1$ and the time interval is $[0; 10\pi]$.

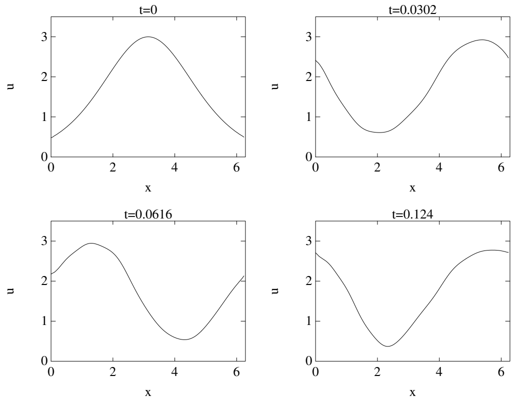

Figure 10.6: x-u plot for the extended KdV equation. $\beta = -1$, $\gamma = -1$, $V = 1$.

If we decrease the influence from the fifth order term and thereby also decreases ϵ then our results shows the movement of a soliton like wave as shown in figure 10.4. The figure shows the coexistence of the main wave and at least one very small rapidly moving wave.

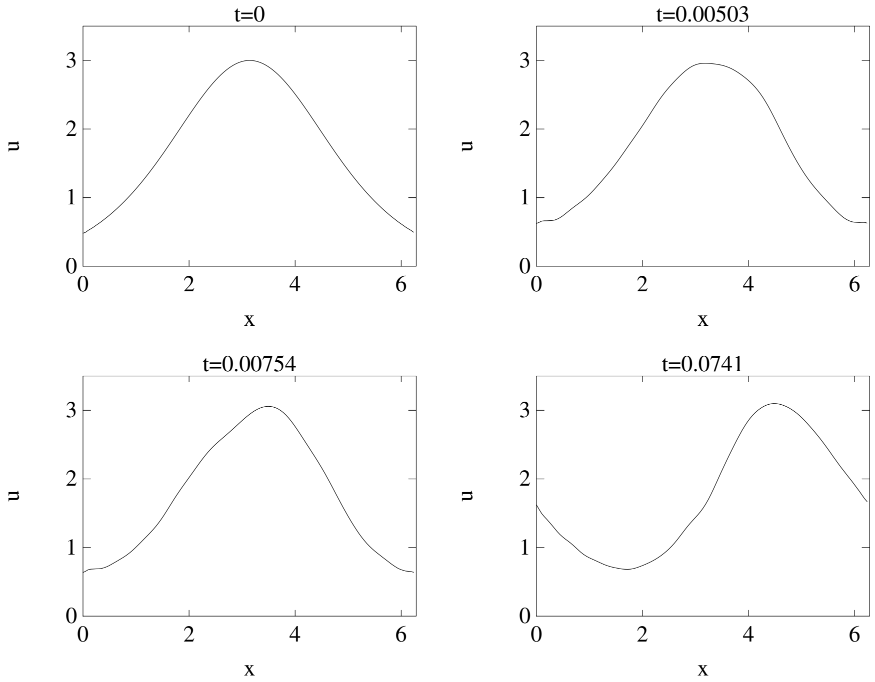

Figure 10.4: x-u plots for the extended KdV equation. $\beta = 1$, $\gamma = 0.01$ and $V = 1$.

It would be likely to consider this result as two solitons originating from the decomposition of the KdV soliton, but seen in the light of the above shown figures

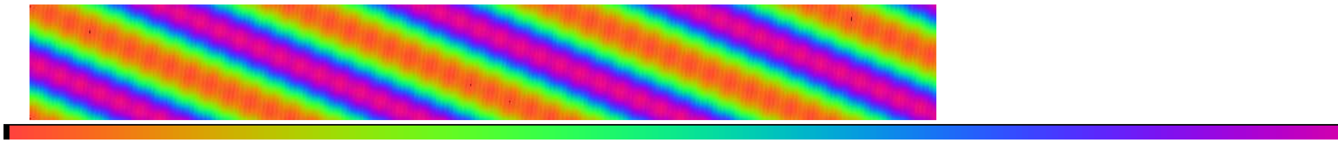

Figure 10.7: t-x color plot for the extended KdV equation. $\beta = 1$, $\gamma = -0.1$, $V = 1$ and the time interval is $[0; 10\pi]$.

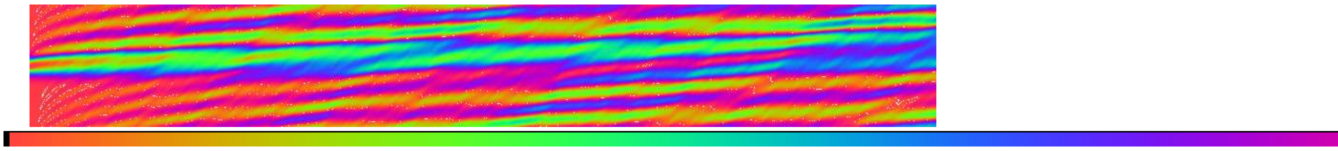

Figure 10.8: t-x color plot for the extended KdV equation. $\beta = 1$, $\gamma = -0.1$, $V = -33.33$ and the time interval is $[0; 0.1\pi]$.

The computations all corresponds with the theories of Karpman. Although they can not be considered a proof of his theories.

The main problem when analyzing the computer results is to determine whether we have a soliton or not. As described above a lot of the computations show soliton-like waves which just deviates by a minimum from being a soliton. We have chosen to accept the presence of a single small wave along with the 'soliton' and explained it as a side effect of the use of the KdV soliton. On the other hand the presence of small oscillations leads us to reject the wave as being a soliton.

The extended Korteweg de Vries equation described in equation 10.1 does without doubt deserve a lot more work and since the computers and numerical methods available today easily allows solution of the initial value problems for this equation. We can only be annoyed that we did not have time to investigate the matter further.

Chapter 11

Properties Regarding ODE's Derived from PDE's with two Spatial Dimensions, and their Jacobians.

In this chapter we deduce some generel properties concerning the Jacobian's of the ODE's derived from 2D PDE's, and the eigenvalues of the Jacobians, and finally we study the problem of aliasing.

11.1 The Eigenvalues of the Jacobian.

In this section we utilize the packing of the 2D problem into a 1D problem which will be described in section 11.1.

Theorem 11.1.1 *The eigenvalues of the Jacobian in the packed domain are identical in time and Fourier space.*

□

Proof 11.1.1 *Based on the definition of the discrete Fourier transform, it is plain that the $N \times M$-point forward transform on $\underline{\underline{u}}$ can be performed by the multiplication,*

$$\begin{aligned}\underline{\underline{\hat{u}}} &= (\underline{\underline{G}}(\underline{\underline{Fu}})^T)^T \\ &= \underline{\underline{FuG^T}}\end{aligned}$$

where $\underline{\underline{F}}$ and $\underline{\underline{G}}$ are the matrices performing one-dimensional transforms,

$$\begin{aligned}F_{k,n} &= e^{-2\pi i \frac{kn}{N}}, & k &\in \{-\tfrac{N}{2}, 1-\tfrac{N}{2}, \ldots, \tfrac{N}{2}-1\} \\ & & n &\in \{0, 1, \ldots, N-1\} \\ G_{l,m} &= e^{-2\pi i \frac{lm}{M}}, & l &\in \{-\tfrac{M}{2}, 1-\tfrac{M}{2}, \ldots, \tfrac{M}{2}-1\} \\ & & m &\in \{0, 1, \ldots, M-1\}.\end{aligned}$$

Representing the transform above in the packed domain leads to

$$\begin{aligned}\hat{\tilde{u}}_{kM+l} &= \hat{u}_{k,l} \\ &= \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} F_{k,n}\tilde{u}_{nM+m}G_{l,m} \\ &= \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} F_{k,n}G_{l,m}\tilde{u}_{nM+m}\end{aligned}$$

which when using matrix notation yields,

$$\begin{aligned}\underline{\underline{\hat{\tilde{u}}}} &= \underline{\underline{H\tilde{u}}}, \\ H_{kM+l,nM+m} &= F_{k,n}G_{l,m} \\ &= e^{-2\pi i(\frac{kn}{N}+\frac{lm}{M})}.\end{aligned}$$

Corresponding considerations for the inverse transform yields

$$H^{-1}_{nM+m,kM+l} = \frac{1}{MN}e^{2\pi i(\frac{nk}{N}+\frac{ml}{M})}.$$

Calculating the matrix-matrix product between $\underline{\underline{H}}$ and $\underline{\underline{H}}^{-1}$ leads to

$$\begin{aligned}HH^{-1}_{kM+l,pM+q} &= \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} H_{kM+l,nM+m}H^{-1}_{nM+m,pM+q} \\ &= \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} \frac{1}{NM}e^{-2\pi i(\frac{kn}{N}+\frac{lm}{M}-\frac{np}{N}-\frac{mq}{M})} \\ &= \sum_{n=0}^{N-1}\frac{1}{N}\left[\frac{1}{M}\sum_{m=0}^{M-1} e^{-2\pi i\frac{m(l-q)}{M}}\right]e^{-2\pi i\frac{n(k-p)}{N}} \\ &= \begin{cases} 1 & k=p \;\wedge\; l=q \\ 0 & otherwise\end{cases}\end{aligned}$$

which means that $\underline{\underline{H}}$ is regular.

If we now take a look at the Jacobian, $\underline{\underline{\tilde{J}}}$, we get,

$$\begin{aligned}
\tilde{\vec{u}}' &= \underline{\underline{\tilde{J}}}\tilde{\vec{u}} \quad \Leftrightarrow \\
\hat{\tilde{\vec{u}}}' &= \underline{\underline{H\tilde{J}}}\tilde{\vec{u}} \\
&= \underline{\underline{H\tilde{J}H^{-1}}}\hat{\tilde{\vec{u}}}
\end{aligned}$$

where $\underline{\underline{H\tilde{J}H}}^{-1}$ is the packed Jacobian in the Fourier space. Since H is regular, it is now obvious that the Jacobians in time and Fourier domains are **similar***. This means that the eigenvalues of the two Jacobians are identical (see e.g. [8, page 474]).*

□

11.2 Some Properties Concerning the Jacobian.

Since the eigenvalues of the Jacobian are not affected by a DFT transform (see theorem 11.1.1), we might as well find the Jacobian in the Fourier domain as in the time domain. We will in this section see that elements of the Jacobian are simplest when found in the Fourier domain, and the rest of this section will therefore take place in the Fourier domain. An $N \times M$-point discretization and following Fourier-transform of the PDE leads to

$$\frac{\partial \underline{\underline{\hat{u}}}}{\partial t} = f(\underline{\underline{\hat{u}}})$$

as we restrict f to be independent of time, i.e. not of the form $f(t, \underline{\underline{\hat{u}}})$. The Fourier domain Jacobi-matrix is defined as

$$\underline{\underline{\hat{J}}} = \frac{\partial f(\underline{\underline{\hat{u}}})}{\partial \underline{\underline{\hat{u}}}}$$

i.e. it has the elements

$$\hat{J}^{k,l}_{n,m} = \frac{\partial f_{k,l}(\underline{\underline{\hat{u}}})}{\partial \hat{u}_{n,m}}, \ k, n \in \{-\frac{N}{2}, \ldots, \frac{N}{2} - 1\}, \ l, m \in \{-\frac{M}{2}, \ldots, \frac{M}{2} - 1\}.$$

A Jacobi-matrix with four indexes is quite unhandy however, and we will deal with this problem later in this section.

If

$$f(\underline{\underline{\hat{u}}}) = ag(\underline{\underline{\hat{u}}}) + bh(\underline{\underline{\hat{u}}}), \ a \in \mathcal{R}, \ b \in \mathcal{R}$$

and the Jacobi-matrices of g and h are $\underline{\underline{\hat{G}}}$ and $\underline{\underline{\hat{H}}}$, we find

$$\begin{aligned}
\hat{J}^{k,l}_{n,m} &= \frac{\partial f_{k,l}(\underline{\underline{\hat{u}}})}{\partial \hat{u}_{n,m}} = a\frac{\partial g_{k,l}(\underline{\underline{\hat{u}}})}{\partial \hat{u}_{n,m}} + b\frac{\partial h_{k,l}(\underline{\underline{\hat{u}}})}{\partial \hat{u}_{n,m}} \\
&= aG^{k,l}_{n,m} + bH^{k,l}_{n,m}, \qquad (11.1)
\end{aligned}$$

and if $f(\underline{\underline{\hat{u}}}) = \underline{\underline{A}} \odot g(\underline{\underline{\hat{u}}})$, where $\underline{\underline{A}}$ is a matrix of the same size as $\underline{\underline{\hat{u}}}$, we find

$$\hat{J}^{k,l}_{n,m} = \frac{\partial f_{k,l}(\underline{\underline{\hat{u}}})}{\partial \hat{u}_{n,m}} = A_{k,l}\frac{\partial g_{k,l}(\underline{\underline{\hat{u}}})}{\partial \hat{u}_{n,m}} = A_{k,l}G^{k,l}_{n,m}.$$

These two things show that the transform of a matrix-function to its Jacobi-matrix is linear.

If the PDE is linear, i.e. of the form

$$\frac{\partial u(x,y,t)}{\partial t} = v(x,y) + \sum_{p=0}^{n}\sum_{q=0}^{m} c_{p,q}\frac{\partial^{p+q}u(x,y,t)}{\partial x^p \partial y^q}$$

where $v(x,y)$ and $c_{p,q}$, $p = 0,1,\ldots,n$, $q = 0,1,\ldots,m$ are known, we found in chapter 6 that

$$f(\underline{\underline{\hat{u}}}) = \underline{\underline{\hat{v}}} + \underline{\underline{A}} \odot \underline{\underline{\hat{u}}},$$

where $\underline{\underline{A}}$ is defined in equation 6.5. For the Jacobi-matrix of f we find

$$\underline{\underline{J}}^{k,l}_{n,m} = \frac{\partial f_{k,l}(\underline{\underline{\hat{u}}})}{\partial \hat{u}_{n,m}} = \frac{\partial}{\partial \hat{u}_{n,m}}(\hat{v}_{k,l} + A_{k,l}\hat{u}_{k,l}) = \begin{cases} A_{k,l}, & (k,l) = (n,m) \\ 0, & (k,l) \neq (n,m) \end{cases} \tag{11.2}$$

Finally we will find the Jacobi-matrix of the ODE,

$$f(\underline{\underline{\hat{u}}}) = (\underline{\underline{\hat{v}}} + \underline{\underline{\underline{A}}}\underline{\underline{\hat{u}}}) * (\underline{\underline{\hat{w}}} + \underline{\underline{\underline{B}}}\underline{\underline{\hat{u}}}),$$

($*$ denotes convolution) which could be derived from the nonlinear PDE consisting of the product of two linear PDE's. When using the approximation

$$f(\underline{\underline{\hat{u}}}) = \mathcal{DFT}(\mathcal{IDFT}(\underline{\underline{\hat{v}}} + \underline{\underline{\underline{A}}}\underline{\underline{\hat{u}}}) \odot \mathcal{IDFT}(\underline{\underline{\hat{w}}} + \underline{\underline{\underline{B}}}\underline{\underline{\hat{u}}})),$$

we will have the aliasing error described in section 11.3. The remedy for the aliasing in this particular approximation will be found later, in chapter 12. In the following, k, l, n, m, p, q, r and s will be used to index $\underline{\underline{\hat{u}}}$, while j and h will be indexes in the time domain.

$$
\begin{aligned}
\hat{J}^{k,l}_{n,m} &= \frac{\partial}{\partial \hat{u}_{n,m}} \mathcal{DFT}(\mathcal{IDFT}(\underline{\underline{\hat{v}}} + \underline{\underline{A}}\underline{\underline{\hat{u}}}) \odot \mathcal{IDFT}(\underline{\underline{\hat{w}}} + \underline{\underline{B}}\underline{\underline{\hat{u}}}))_{k,l} \\
&= \frac{\partial}{\partial \hat{u}_{n,m}} \sum_{j,h} (\frac{1}{NM} \sum_{p,q} (\hat{v}_{p,q} + A_{p,q}\hat{u}_{p,q}) e^{2\pi i(\frac{jp}{N}+\frac{hq}{M})}) \\
&\quad (\frac{1}{NM} \sum_{r,s} (\hat{w}_{r,s} + B_{r,s}\hat{u}_{r,s}) e^{2\pi i(\frac{jr}{N}+\frac{hs}{M})}) e^{-2\pi i(\frac{kj}{N}+\frac{lh}{M})} \\
&= \sum_{j,h} (\frac{1}{NM} A_{n,m} e^{2\pi i(\frac{jn}{N}+\frac{hm}{M})} (\frac{1}{NM} \sum_{r,s} (\hat{w}_{r,s} + B_{r,s}\hat{u}_{r,s}) e^{2\pi i(\frac{jr}{N}+\frac{hs}{M})}) \\
&\quad + \frac{1}{NM} B_{n,m} e^{2\pi i(\frac{jn}{N}+\frac{hm}{M})} (\frac{1}{NM} \sum_{p,q} (\hat{v}_{p,q} + A_{p,q}\hat{u}_{p,q}) e^{2\pi i(\frac{jp}{N}+\frac{hq}{M})})) \\
&\quad e^{-2\pi i(\frac{kj}{N}+\frac{lh}{M})} \\
&= \sum_{j,h} (\frac{1}{NM} \sum_{r,s} \frac{1}{NM} A_{n,m} (\hat{w}_{r,s} + B_{r,s}\hat{u}_{r,s}) e^{2\pi i(\frac{jr}{N}+\frac{hs}{M})} \\
&\quad + \frac{1}{NM} \sum_{p,q} \frac{1}{NM} B_{n,m} (\hat{v}_{p,q} + A_{p,q}\hat{u}_{p,q}) e^{2\pi i(\frac{jp}{N}+\frac{hq}{M})}) \\
&\quad e^{-2\pi i(\frac{j(k-n)}{N}+\frac{h(l-m)}{M})} \\
&= \sum_{j,h} (\frac{1}{NM} \sum_{r,s} (\frac{1}{NM} A_{n,m} (\hat{w}_{r,s} + B_{r,s}\hat{u}_{r,s}) \\
&\quad + \frac{1}{NM} B_{n,m} (\hat{v}_{r,s} + A_{r,s}\hat{u}_{r,s})) e^{2\pi i(\frac{jr}{N}+\frac{hs}{M})}) e^{-2\pi i(\frac{j(k-n)}{N}+\frac{h(l-m)}{M})} \\
&\quad k,n = -\frac{N}{2}, -\frac{N}{2}+1, \ldots, \frac{N}{2}-1, \quad l,m = -\frac{M}{2}, -\frac{M}{2}+1, \ldots, \frac{M}{2}-1
\end{aligned}
$$

This is identified as an IDFT embedded in a DFT, where the elements are shifted, thus

$$
\begin{aligned}
\hat{J}^{k,l}_{n,m} &= \frac{1}{NM} A_{n,m} (\hat{w}_{r,s} + B_{r,s}\hat{u}_{r,s}) + \frac{1}{NM} B_{n,m} (\hat{v}_{r,s} + A_{r,s}\hat{u}_{r,s})) \\
&= \frac{1}{NM} \hat{u}_{r,s} (A_{n,m}B_{r,s} + B_{n,m}A_{r,s}) + \frac{1}{NM} (A_{n,m}\hat{w}_{r,s} + B_{n,m}\hat{v}_{r,s})
\end{aligned}
\tag{11.3}
$$

$$
r = \begin{cases} k-n, & -\frac{N}{2} \leq k-n < \frac{N}{2} \\ k-n+N, & k-n < -\frac{N}{2} \\ k-n-N, & k-n \geq \frac{N}{2} \end{cases}
$$

$$
s = \begin{cases} l-m, & -\frac{M}{2} \leq l-m < \frac{M}{2} \\ l-m+M, & l-m < -\frac{M}{2} \\ l-m-M, & l-m \geq \frac{M}{2}, \end{cases}
$$

i.e. in this nonlinear case the Jacobi-matrix is dependent on $\underline{\underline{\hat{u}}}$.

To ease the handling of the Jacobi-matrix, we now introduce another way of regarding the matrix.

The differential of $f(\underline{\underline{\hat{u}}})$ at $\underline{\underline{\hat{u}}}$ is given by

$$df_{k,l} = \sum_{n,m} \hat{J}^{k,l}_{n,m}\, d\hat{u}_{n,m},$$

so if we define $\tilde{\vec{u}}$ (where $\underline{\underline{\hat{u}}}$ is $N \times M$) by

$$\tilde{u}_{nM+m} = u_{n,m},$$

and $\underline{\underline{\tilde{\hat{J}}}}$ by

$$\tilde{J}_{kM+l,nM+m} = \hat{J}^{k,l}_{n,m},$$

we see that

$$df_{k,l} = \sum_{n,m} \tilde{J}_{kM+l,nM+m}\, d\tilde{u}_{nM+m},$$

and therefore $\underline{\underline{\tilde{\hat{J}}}}\, d\tilde{\vec{u}}$ holds the differential of f on a vector form. This means that $f(\underline{\underline{\hat{u}}})$ can be approximated by

$$f(\tilde{\vec{u}}) = \underline{\underline{\tilde{\hat{J}}}}\, \tilde{\vec{u}}.$$

This implies the important result that, according to [11], it is the eigenvalues of $\underline{\underline{\tilde{\hat{J}}}}$ that determine the stability of $\partial \underline{\underline{\hat{u}}}/\partial t = f(\underline{\underline{\hat{u}}})$.

The chosen reordering has the convenient property, that if $\underline{\underline{\hat{J}}}$ is derived from a linear PDE, then

$$\hat{J}^{k,l}_{n,m} = \begin{cases} A_{k,l}, & (k,l) = (n,m) \\ 0, & (k,l) \neq (n,m) \end{cases} = \tilde{J}_{kM+l,nM+m}$$

i.e. if f is linear, $\underline{\underline{\tilde{\hat{J}}}}$ has all its non zero elements on its diagonal, as in the 1-dimensional case.

Since the two ways of representing the Jacobi-matrix are fully interchangeable, we will use whichever we find convenient in the following.

The reordering of $\underline{\underline{\hat{u}}}$, where the rows are put in continuation of each other, could just as well be achieved by instead putting the columns of $\underline{\underline{\hat{u}}}$ in continuation of each other, i.e. by defining $\tilde{u}_{k+Nl} = \hat{u}_{k,l}$. It is clear that this reordering also would result in a matrix, determining the eigenvalues of stability, and with the property that the elements derived from a linear equation would lie on the diagonal of the Jacobi-matrix.

11.3 The Matter of Antialiasing.

As described in the one-dimensional case we have a problem regarding aliasing when computing convolution sums. When analyzing the matter in two dimensions we get:

$$
\begin{aligned}
u_{m,n} &= \frac{1}{Q}\sum_{l=-\frac{Q}{2}}^{\frac{Q}{2}-1}\frac{1}{P}\sum_{k=-\frac{P}{2}}^{\frac{P}{2}-1}\hat{u}_{k,l}e^{2\pi i\frac{mk}{P}}e^{2\pi i\frac{nl}{Q}}\\
v_{m,n} &= \frac{1}{Q}\sum_{l=-\frac{Q}{2}}^{\frac{Q}{2}-1}\frac{1}{P}\sum_{k=-\frac{P}{2}}^{\frac{P}{2}-1}\hat{v}_{k,l}e^{2\pi i\frac{mk}{P}}e^{2\pi i\frac{nl}{Q}}\\
w_{m,n} &= u_{m,n}v_{m,n}, \quad n\in\{0,1,\ldots,Q-1\} \quad m\in\{0,1,\ldots,P-1\}\\
\hat{w}_{p,q} &= \sum_{n=0}^{Q-1}\sum_{m=0}^{P-1}u_{m,n}v_{m,n}e^{-2\pi i\frac{mp}{P}}e^{-2\pi i\frac{nq}{Q}}\\
&= \sum_{n=0}^{Q-1}\sum_{m=0}^{P-1}[\frac{1}{Q}\sum_{l=-\frac{Q}{2}}^{\frac{Q}{2}-1}\frac{1}{P}\sum_{k=-\frac{P}{2}}^{\frac{P}{2}-1}\hat{u}_{k,l}e^{2\pi i\frac{mk}{P}}e^{2\pi i\frac{nl}{Q}}]\\
&\quad [\frac{1}{Q}\sum_{s=-\frac{Q}{2}}^{\frac{Q}{2}-1}\frac{1}{P}\sum_{r=-\frac{P}{2}}^{\frac{P}{2}-1}\hat{v}_{r,s}e^{2\pi i\frac{mk}{P}}e^{2\pi i\frac{nl}{Q}}]e^{-2\pi i\frac{mp}{P}}e^{-2\pi i\frac{nq}{Q}}\\
&= \frac{1}{P^2Q^2}\sum_{s=-\frac{Q}{2}}^{\frac{Q}{2}-1}\sum_{r=-\frac{P}{2}}^{\frac{P}{2}-1}\sum_{l=-\frac{Q}{2}}^{\frac{Q}{2}-1}\sum_{k=-\frac{P}{2}}^{\frac{P}{2}-1}\\
&\quad \sum_{n=0}^{Q-1}\sum_{m=0}^{P-1}\hat{u}_{k,l}\hat{v}_{r,s}e^{2\pi i\frac{m}{P}(k+r-p)}e^{2\pi i\frac{n}{Q}(l+s-q)}\\
&= \frac{1}{PQ}\sum_{l+s=q}\sum_{k+r=p}\hat{u}_{k,l}\hat{v}_{r,s}\\
&+ \frac{1}{PQ}\sum_{l+s=q}\sum_{k+r\pm P=p}\hat{u}_{k,l}\hat{v}_{r,s}\\
&+ \frac{1}{PQ}\sum_{l+s\pm Q=q}\sum_{k+r=p}\hat{u}_{k,l}\hat{v}_{r,s}\\
&+ \frac{1}{PQ}\sum_{l+s\pm Q=q}\sum_{k+r\pm P=p}\hat{u}_{k,l}\hat{v}_{r,s}
\end{aligned}
$$

where all the terms except are problematic apart from the first one. In order to remove this error, we need $\hat{u}_{k,l}\hat{v}_{r,s}$ to be zero for $\mid k+r-p\mid= P$ and $\mid l+s-q\mid= Q$. Inspired by [9], we can do this by padding with zeroes. If we originally had the $(M+1)\times(N+1)$-point sequences $\hat{u}'$ and $\hat{v}'$, we define

$$
\hat{u}_{k,l} = \begin{cases} \hat{u}'_{k,l} & \mid k\mid\le\frac{M}{2}\wedge\mid l\mid\le\frac{N}{2}\\ 0 & otherwise\end{cases}
$$

and similarly for v. We are then only interested in the convolution sum, $w_{p,q}$ for $\mid p\mid\le\frac{M}{2}\wedge\mid q\mid\le\frac{N}{2}$ which leads to

$$
\begin{array}{lcl}
\mid k+r-p\mid< P\Rightarrow & & \mid l+s-q\mid< Q\Rightarrow\\
\frac{M}{2}+\frac{M}{2}--\frac{M}{2}<P\Leftrightarrow & \wedge & \frac{N}{2}+\frac{N}{2}--\frac{N}{2}<Q\Leftrightarrow\\
P>\frac{3M}{2} & & Q>\frac{3N}{2}
\end{array}
$$

that is we have to perform the discrete Fourier transform on an array which is actually 50% longer than the original one on each axis - meaning an increase in computer work of about 140% when using FFTs.

Chapter 12

The Hasegawa-Mima Equation.

The Hasegawa-Mima equation defined as

$$\frac{\partial}{\partial t}(u-\nabla^2 u) - v_D\frac{\partial u}{\partial x} - J(u,\nabla^2 u) = 0, \tag{12.1}$$
$$J(a,b) = \frac{\partial a}{\partial x}\frac{\partial b}{\partial y} - \frac{\partial a}{\partial y}\frac{\partial b}{\partial x},$$

describes nonlinear phenomena in plasma and in the atmosphere. This nonlinear 2D partial differential equation is named after A.Hasegawa and K.Mima, who derived it in 1977.

12.1 Fourier Transformation of the PDE.

As initial value we use a Gaussian vortex

$$u(x,y,0) = A\exp(\frac{-(x^2+y^2)}{\varsigma^2}) \tag{12.2}$$

where $A=-1$, $\varsigma^2=1$, and where we restrict v_D to be 0.1.

The Fourier transform of the spatial dimensions yields

$$\frac{\partial}{\partial t}\mathcal{F}(u-\nabla^2 u) - v_D\mathcal{F}(\frac{\partial u}{\partial x}) - \mathcal{F}(J(u,\nabla^2 u)) = 0,$$
$$J(a,b) = \frac{\partial a}{\partial x}\frac{\partial b}{\partial y} - \frac{\partial a}{\partial y}\frac{\partial b}{\partial x},$$

where we have used the linearity of the Fourier transform and the fact that time-derivation is transparent to the spatial Fourier transform, as shown in section 2.2.2. Again $\hat{u}$ will denote transformation of the spatial dimensions of u. Using the results about the transform of spatial derivatives from section 2.2.2 yields

$$\mathcal{F}(\nabla^2 u) = \mathcal{F}(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) = \hat{u}(k,l)(-k^2-l^2)$$

where k and l are the Fourier domain variables, and

$$\begin{aligned}
J(u, \nabla^2 u) &= \frac{\partial u}{\partial x}\frac{\partial \nabla^2 u}{\partial y} - \frac{\partial u}{\partial y}\frac{\partial \nabla^2 u}{\partial x} \\
&= \mathcal{F}^{-1}(ik\hat{u})\mathcal{F}^{-1}(il\widehat{\nabla^2 u}) - \mathcal{F}^{-1}(il\hat{u})\mathcal{F}^{-1}(ik\widehat{\nabla^2 u}) \\
&= \mathcal{F}^{-1}(ik\hat{u})\mathcal{F}^{-1}(-il(k^2+l^2)\hat{u}) - \mathcal{F}^{-1}(il\hat{u})\mathcal{F}^{-1}(-ik(k^2+l^2)\hat{u})
\end{aligned}$$

Using this, the transformed PDE turns into

$$\begin{aligned}
&\frac{\partial}{\partial t}(\hat{u}(1+k^2+l^2)) - v_D ik\hat{u} \\
&-\mathcal{F}(\mathcal{F}^{-1}(ik\hat{u})\mathcal{F}^{-1}(-il(k^2+l^2)\hat{u}) - \mathcal{F}^{-1}(il\hat{u})\mathcal{F}^{-1}(-ik(k^2+l^2)\hat{u})) \\
= \;&\frac{\partial}{\partial t}(\hat{u}(1+k^2+l^2)) - v_D ik\hat{u} \\
&-(ik\hat{u}) * (-il(k^2+l^2)\hat{u}) + (il\hat{u}) * (-ik(k^2+l^2)\hat{u}) = 0
\end{aligned}$$

where $*$ denotes convolution. Separating $\partial\hat{u}/\partial t$ yields

$$\begin{aligned}
\frac{\partial \hat{u}}{\partial t} &= \frac{ikv_D}{1+k^2+l^2}\hat{u} - \frac{1}{1+k^2+l^2}\mathcal{F}(\mathcal{F}^{-1}(ik\hat{u})\mathcal{F}^{-1}(il(k^2+l^2)\hat{u}) \\
&\quad -\mathcal{F}^{-1}(il\hat{u})\mathcal{F}^{-1}(ik(k^2+l^2)\hat{u})) \\
&= \frac{ikv_D}{1+k^2+l^2}\hat{u} - \frac{1}{1+k^2+l^2}((ik\hat{u}) * (il(k^2+l^2)\hat{u}) \\
&\quad -(il\hat{u}) * (ik(k^2+l^2)\hat{u}))
\end{aligned}$$

which is a nonlinear ODE, requiring either two convolutions or five Fourier transforms to the computation of $\partial\hat{u}/\partial t$.

12.2 Discretization of the ODE.

An $N \times M$-point discretization of the domain

$$(x, y) \in [-L_x/2; L_x/2[\times [-L_y/2; L_y/2[$$

turns the above into

$$\frac{\partial \underline{\underline{\hat{u}}}}{\partial t} = \underline{\underline{E}}\,\underline{\underline{\hat{u}}} + \underline{\underline{F}}((\underline{\underline{A}}\,\underline{\underline{\hat{u}}}) * (\underline{\underline{B}}\,\underline{\underline{\hat{u}}}) - (\underline{\underline{C}}\,\underline{\underline{\hat{u}}}) * (\underline{\underline{D}}\,\underline{\underline{\hat{u}}}))$$

approximated by

$$\frac{\partial \underline{\underline{\hat{u}}}}{\partial t} = \underline{\underline{E}\hat{u}} + \underline{\underline{F}}\mathcal{DFT}(\mathcal{IDFT}(\underline{\underline{A\hat{u}}})\mathcal{IDFT}(\underline{\underline{B\hat{u}}}) - \mathcal{IDFT}(\underline{\underline{C\hat{u}}})\mathcal{IDFT}(\underline{\underline{D\hat{u}}})) \quad (12.3)$$

where the discretization is done as in equation 2.3.1. This means that

$$
\begin{aligned}
&A_{k,l} = \begin{cases} ikd_x, & k = 0, \pm 1, \ldots, \pm(\frac{N}{2} - 1),\ l = -\frac{M}{2}, \ldots, \frac{M}{2} - 1 \\ 0, & k = -\frac{N}{2},\ l = -\frac{M}{2}, \ldots, \frac{M}{2} - 1 \end{cases} \\
&B_{k,l} = \begin{cases} ild_y((kd_x)^2 + (ld_y)^2), \\ k = -\frac{N}{2}, \ldots, \frac{N}{2} - 1,\ l = 0, \pm 1, \ldots, \pm(\frac{M}{2} - 1) \\ 0,\ k = -\frac{N}{2}, \ldots, \frac{N}{2} - 1,\ l = -\frac{M}{2}, \end{cases} \\
&C_{k,l} = \begin{cases} ild_y, & k = -\frac{N}{2}, \ldots, \frac{N}{2} - 1,\ l = 0, \pm 1, \ldots, \pm(\frac{M}{2} - 1) \\ 0, & k = -\frac{N}{2}, \ldots, \frac{N}{2} - 1,\ l = -\frac{M}{2} \end{cases} \\
&D_{k,l} = \begin{cases} ikd_x((kd_x)^2 + (ld_y)^2), \\ k = 0, \pm 1, \ldots, \pm(\frac{N}{2} - 1),\ l = -\frac{M}{2}, \ldots, \frac{M}{2} - 1 \\ 0,\ k = -\frac{N}{2},\ l = -\frac{M}{2}, \ldots, \frac{M}{2} - 1 \end{cases} \\
&E_{k,l} = \begin{cases} ikd_x v_D(1 + (kd_x)^2 + (ld_y)^2)^{-1}, \\ k = 0, \pm 1, \ldots, \pm(\frac{N}{2} - 1),\ l = -\frac{M}{2}, \ldots, \frac{M}{2} - 1, \\ 0,\ k = -\frac{N}{2},\ l = -\frac{M}{2}, \ldots, \frac{M}{2} - 1 \end{cases} \\
&F_{k,l} = -(1 + (kd_x)^2 + (ld_y)^2)^{-1},\ k = -\frac{N}{2}, \ldots, \frac{N}{2} - 1,\ l = -\frac{M}{2}, \ldots, \frac{M}{2} - 1 \\
&d_x = \frac{2\pi}{L_x},\ d_y = \frac{2\pi}{L_y}.
\end{aligned}
\tag{12.4}
$$

12.3 Solution of the Discretized ODE.

The results of the use of Runge-Kutta methods in the solution of the KdV-equation lead to the consideration of these methods in the 2-dimensional case. When using an implicit method, like SIRK2, with NR-iteration, the Jacobi-matrix of the discretized equation is required. Since the eigenvalues of the Jacobi-matrix may favor more simple methods, like ERK4, or show that a more stable (and expensive) method is required, it is in any case desirable to find the Jacobi-matrix, and thereby get a hint of the location of these eigenvalues.

12.3.1 The Jacobi-Matrix of the Discretized ODE.

In our case $\underline{\underline{f}}$ is the RHS of equation 12.3, i.e.

$$\underline{\underline{f}}(\underline{\underline{\hat{u}}}) = \underline{\underline{E\hat{u}}} + \underline{\underline{F}}((\underline{\underline{A\hat{u}}}) * (\underline{\underline{B\hat{u}}}) - (\underline{\underline{C\hat{u}}}) * (\underline{\underline{D\hat{u}}}))$$

and approximating the convolution with DFT's, we end up with

$$
\begin{aligned}
&\underline{\underline{f}}(\underline{\underline{\hat{u}}}) = \underline{\underline{E\hat{u}}} + \underline{\underline{F}}\mathcal{DFT}(\mathcal{IDFT}(\underline{\underline{A\hat{u}}})\mathcal{IDFT}(\underline{\underline{B\hat{u}}})) \\
&-\underline{\underline{F}}\mathcal{DFT}(\mathcal{IDFT}(\underline{\underline{C\hat{u}}})\mathcal{IDFT}(\underline{\underline{D\hat{u}}}))
\end{aligned}
$$

Using the results of section 11.1 we find for $(k,l) \neq (n,m)$

$$
\begin{aligned}
J^{k,l}_{n,m} &= F_{k,l}\frac{1}{NM}\hat{u}_{r,s}(A_{n,m}B_{r,s} + B_{n,m}A_{r,s}) \\
&- F_{k,l}\frac{1}{NM}\hat{u}_{r,s}(C_{n,m}D_{r,s} + D_{n,m}C_{r,s}) \\
&= \frac{F_{k,l}}{NM}\hat{u}_{r,s}(A_{n,m}B_{r,s} + B_{n,m}A_{r,s} - C_{n,m}D_{r,s} - D_{n,m}C_{r,s}) \\
&\qquad r = \begin{cases} k-n, & -\frac{N}{2} \le k-n < \frac{N}{2} \\ k-n+N, & k-n < -\frac{N}{2} \\ k-n-N, & k-n \ge \frac{N}{2} \end{cases} \\
&\qquad s = \begin{cases} l-m, & -\frac{M}{2} \le l-m < \frac{M}{2} \\ l-m+M, & l-m < -\frac{M}{2} \\ l-m-M, & l-m \ge \frac{M}{2}, \end{cases}
\end{aligned}
$$

and for $(k,l) = (n,m)$ we find

$$
\begin{aligned}
J^{k,l}_{k,l} &= E_{k,l} + \frac{F_{k,l}}{NM}\hat{u}_{0,0}(A_{k,l}B_{0,0} + B_{k,l}A_{0,0} - C_{k,l}D_{0,0} + D_{k,l}C_{0,0}) \\
&= E_{k,l}, \qquad (12.5)
\end{aligned}
$$

since $A_{0,0} = B_{0,0} = C_{0,0} = D_{0,0} = 0$. Knowing that the corresponding $\hat{\tilde{\underline{\underline{J}}}}$ is sparse would be convenient, and therefore this is investigated numerically. Using TITAN - MATLAB, the M-files in appendix C, which generates $\tilde{\underline{\underline{J}}}$, shows that for $N = M = 8$, 25% of the elements are zero, while for $N = M = 16$ only 7% are zero. If $N = M = 32$, as little as 2% of the elements are zero, and on this material, we conclude that $\hat{\tilde{\underline{\underline{J}}}}$ is not sparse, especially when using a higher resolution. The fact that an $N \times N$ discretization leads to a Jacobian with N^4 elements, implies that the use of SIRK2, as in section 9.3, would have an order of magnitude $O(N^a)$, $4 < a < 6$, which is not feasible. The high order of magnitude is caused by the use of a direct method when solving the system of linear equations in connection with the Newton-Raphson iteration. The coefficient matrix is however of the form $\underline{\underline{I}} - h\hat{\tilde{\underline{\underline{J}}}}$, hence the system can probably be solved efficiently using an iterative solver. Since the current version (v. 2.2) of the CMSSL-library (and the recently released v. 3.0) does not feature an iterative solver, and since we have no time for writing one ourselves, we are unable to make use of SIRK2 in the two-dimensional case.

12.3.2 The Eigenvalues of the Jacobi-Matrix of the Discretized ODE.

The fact that $J_{k,l}^{k,l} = E_{k,l}$ means that $\tilde{J}_{kM+l,kM+l} = E_{k,l}$, i.e. the diagonal of $\underline{\underline{\hat{\tilde{J}}}}$ consists of the elements of $\underline{\underline{E}}$. Using the property found in [10], concerning the eigenvalues λ_i for any $N \times N$-matrix $\underline{\underline{A}}$, that

$$\sum_{i=1}^{N} \lambda_i = \sum_{i=1}^{N} a_{i,i},$$

we find for $\underline{\underline{\hat{\tilde{J}}}}$

$$\sum_{i=1}^{NM} \lambda_i = 0,$$

since

$$\sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \sum_{l=-\frac{M}{2}}^{\frac{M}{2}-1} E_{k,l} = \sum_{k=0}^{\frac{N}{2}-1} \sum_{l=-\frac{M}{2}}^{\frac{M}{2}-1} \frac{ikd_x v_D}{1+(kd_x)^2+(ld_y)^2} + \frac{i(-k)d_x v_D}{1+((-k)d_x)^2+(ld_y)^2} = 0,$$

where $d_x = 2\pi/L_x,\ d_y = 2\pi/L_y$.

This convenient property implies that

$$\sum_{i=1}^{NM} \mathrm{Real}(\lambda_i) = 0, \ \sum_{i=1}^{NM} \mathrm{Imag}(\lambda_i) = 0.$$

This means that if one of the eigenvalues has a non-zero real-part, then at least one eigenvalue will have an eigenvalue with a positive real-part. This is not a problem however, as long as the eigenvalues are not too large, which we will see from the following, where $'$ denotes time differentiation. We solve $\tilde{\vec{u}}' = f(\tilde{\vec{u}})$ by use of the approximation

$$\tilde{\vec{u}}' = \underline{\underline{\tilde{J}}}\tilde{\vec{u}}.$$

This set of coupled equations can, by use of a collineatory transformation, be uncoupled to a set of equations of the form

$$y' = \lambda y,$$

where, for each of the equations, λ is one of $\underline{\underline{\tilde{J}}}$'s eigenvalues. Applying a one-step method to this yields

$$y_{n+1} = R(\lambda h) y_n,$$

where, for the explicit Runge-Kutta methods, $R(z)$ is a polynomial. The error of this is bound by

$$||e_{n+1}|| \leq ||R(\lambda h)||||e_n||,$$

and since the exact solution of

$$y' = \lambda y$$

is [1]

$$y(t_n + h) = e^{\lambda h} y(t_n)$$

then the error $||e_{n+1}|| \leq ||R(\lambda h)||||e_n||$ must be less than $||e^{\lambda h}||||e_n||$, hence

$$||R(\lambda h)|| \leq ||e^{\lambda h}||$$

must hold. When $\mathrm{Real}(\lambda) > 0$, then this will hold for any ERK method, since their $R(z)$, according to [11], is a truncated Taylor series of e^z. For those $\mathrm{Real}(\lambda) \leq 0$, the region of absolute stability of the ERK should include λh, to ensure stability.

This favorizes the use of an explicit Runge-Kutta method; there is however one thing that still must be considered before choosing such a method, and that is the maximum of the eigenvalues in the left half-plane. The stability domains of the ERK's are quite small, in the sense that large eigenvalues in the left halfplane will make the largest possible step length unrealistically small.

[1] This exact solution cannot be used to solve the equation directly, since it is not feasible to find the matrices to do the collineatory transformation of $\underline{\underline{\tilde{J}}}$ to $\mathrm{diag}(\lambda)$, and since the approximation $\tilde{\vec{u}}' = \underline{\underline{\tilde{J}}}\tilde{\vec{u}}$ is too inexact.

The eigenvalues can be found numerically, but since the size of $\underline{\underline{\hat{J}}}$ is NM by NM, even a modest spatial resolution of 64 by 64 will require the finding of 4096 eigenvalues, which we find unrealistic, since we have no CM-routines to find eigenvalues. For $N = 8, 16, 32$ and $M \leq N$, we have found the eigenvalues of $\underline{\underline{\hat{J}}}$, where $\underline{\underline{\hat{u}}}$ is the Fourier transformed of the initial guess given in equation 12.2. For $M = N$, we have evaluated $\underline{\underline{u}}$ in the domain given in the beginning of the previous section: $L_x = L_y = 25$, for $M = N/2$ we have reduced L_y by a factor 2, and for $M = N/4$ by a factor 4. The calculations were done using MATLAB on an Ardent TITAN, again using the M-files in appendix C. In all cases, the eigenvalues are situated around zero; in all four quadrants, the imaginary part of the eigenvalues is less than 1.1, while the real part is less than 0.015. On this material we assume that the eigenvalues, with the given initial value of $\underline{\underline{u}}$, and with a relevant resolution (up to say $NM = 1024^2, N \geq M$), will be small enough to allow a reasonable steplength, using an ERK. On the basis of these considerations, we will try to solve the HM-equation using the fourth order explicit Runge-Kutta.

12.4 Implementation of a PDEsolver, *pde2d*, using ERK4.

The implementation is quite easy, since the I/O-section basically is the same as in the linear 2D-case. We therefore reuse the I/O-section of the program *linpde2d*, combined with the routines for writing resolution etc. on the solution plots, taken from the KdV-solver. What is left to code, falls into three groups: 1) computation of the initial value of the solution (we restrict this to only one function), 2) evaluation of the right side of the ODE, and 3) calculation of the step in time, which is based on ERK4. re 1) The initial value of the solution is the one given in equation 12.2. The computation of this is quite straightforward, x and y lie in the same domain as in the calculation of the eigenvalues, i.e. $-25/2 \leq x < 25/2$, $-25/2 \leq y < 25/2$. re 2) Contrary to the linear case, the ODE's RHS is not constant, and as seen in section 12.2 this particular ODE requires five Fourier transforms in the calculation of the RHS. These transforms are done using our own real-to-complex FFT. The six constants used in the evaluation of the RHS are initialized as in section 12.2; however, since these constant arrays are all used elementwise in the Fourier domain, and since arrays in this domain are bit-reversed, these six constants must be bitreversed as well. This is done with the usual dummycall of the CMSSL-FFT. The considerations concerning the six constants of the RHS being only half sized, are the same as in the linear case, described in 6.2. Since the nonlinear term of the ODE is the one that will cause a possible instability, we have given the user of the program the option of setting the coefficient to the linear term of the equation, v_D, since a more dominant linear term probably will make the solution more stable. re 3) The implementation of ERK4 is done using a routine that takes the Fourier coefficients of the actual solution, and the six constants, and returns the Fourier coefficients of the next solution after four evaluations of the ODE's RHS. Our ERK4 implementation has been slightly changed, as described in appendix 7.4.1. Because of the similarity between this program and *linpde2d*, we have not described this program thoroughly, but instead written plenty of comments in the new routines: *make_u0*, *make_ODEconst*, *ODE_RHS*, and *ERK4_6*.

The program can by the way be changed easily, to solve (or at least try to solve) any 2D PDE, that can be Fourier transformed to a set of ODE's of the form $\partial\underline{\underline{\hat{u}}}/\partial t = f(\underline{\underline{\hat{u}}})$ (provided that the programmer can write the code to evaluate f). This would require reprogramming of *make_u0* (the initial solution), especially *ODE_RHS*, and *make_ODEconst*, and if *make_ODEconst* returns n constants instead of six, the parameterlist in *ERK4_6* and the declarations in *PDE_solver* must be changed to this number.

12.5 Improving the Solver with Antialiasing.

Applying equation 12.1 to the above solver results in instability at about $t = 1$, even though the solution should be stable up to $t = 1000$ or more. Thinking that this instability can not only be caused by the insufficiency of ERK4, we look for other reasons. The first one to come to mind is that the instability is caused by the approximation 12.3, where the two convolutions are displaced by five truncated Fourier-series. As described in 11.3 this will cause aliasing, and the remedy, called the antialiasing, is simply to set the modes causing this error to zero. Since the error causing term is

$$g(\underline{\underline{\hat{u}}}) = \underline{\underline{F}}\mathcal{DFT}(\mathcal{IDFT}(\underline{\underline{A\hat{u}}})\mathcal{IDFT}(\underline{\underline{B\hat{u}}}) - \mathcal{IDFT}(\underline{\underline{C\hat{u}}})\mathcal{IDFT}(\underline{\underline{D\hat{u}}})),$$

then the antialiasing is done by not resetting the lowest $[2N/3]$ respectively $[2M/3]$ modes ($[\cdot]$ truncates towards zero), i.e. resetting the modes

$$-N/2, \ldots, 1-[N/3],\ 1+[N/3], \ldots, N/2,$$

and likewise for M, in the terms $\underline{\underline{A\hat{u}}}$, $\underline{\underline{B\hat{u}}}$, $\underline{\underline{C\hat{u}}}$, $\underline{\underline{D\hat{u}}}$, and $g(\underline{\underline{\hat{u}}})$, which can be done by simply resetting the factors to these modes in $\underline{\underline{A}}$, $\underline{\underline{B}}$, $\underline{\underline{C}}$, $\underline{\underline{D}}$, $\underline{\underline{F}}$. It is very convenient that the antialiasing can be done this way, since the evaluation of g is unchanged, and since the Jacobian is still found in the same way, just using other constants.

12.5.1 The Eigenvalues of the Antialiased ODE.

Since we have changed five of the six constants describing the ODE, we will analyze how this effects the eigenvalues of the ODE. The elements of the Jacobian are for $(k,l) \neq (n,m)$ still

$$\begin{aligned}
\tilde{J}_{kM+l,nM+m} &= \frac{F_{k,l}}{NM}\hat{u}_{r,s}(A_{n,m}B_{r,s} + B_{n,m}A_{r,s} - C_{n,m}D_{r,s} - D_{n,m}C_{r,s}) \\
& r = \begin{cases} k-n, & -\frac{N}{2} \le k-n < \frac{N}{2} \\ k-n+N, & k-n < -\frac{N}{2} \\ k-n-N, & k-n \ge \frac{N}{2} \end{cases} \\
& s = \begin{cases} l-m, & -\frac{M}{2} \le l-m < \frac{M}{2} \\ l-m+M, & l-m < -\frac{M}{2} \\ l-m-M, & l-m \ge \frac{M}{2}, \end{cases}
\end{aligned}$$

and for $(k,l) = (n,m)$

$$\tilde{J}_{kM+l,kM+l} = E_{k,l}.$$

Since $\underline{\underline{E}}$ is not affected by the antialiasing, the diagonal of $\underline{\underline{\hat{J}}}$ is the same. This means that we still have the property that

$$\sum_{i=1}^{NM} \mathrm{Real}(\lambda_i) = 0,\ \sum_{i=1}^{NM} \mathrm{Imag}(\lambda_i) = 0.$$

To get a better idea of the maximum possible step-length using ERK4, we now attempt to literally encircle the location of the eigenvalues, using Gershgorin's discs described in [10]. Our $\underline{\underline{\hat{\tilde{J}}}}$ has NM eigenvalues, encircled by NM discs, described by

$$c_i = \tilde{J}_i, \; r_i = \sum_{j \neq i} \tilde{J}_{i,j}.$$

Since

$$\begin{aligned} c_{kM+l} &= E_{k,l} = \begin{cases} ikd_x v_D(1+(kd_x)^2+(ld_y)^2)^{-1}, \\ k=0,\pm 1,\ldots,\pm(\frac{N}{2}-1), \; l=-\frac{M}{2},\ldots,\frac{M}{2}-1, \\ 0, \; k=-\frac{N}{2}, \; l=-\frac{M}{2},\ldots,\frac{M}{2}-1 \end{cases} \\ d_x &= \frac{2\pi}{L_x}, \; d_y = \frac{2\pi}{L_y} \end{aligned} \tag{12.6}$$

the centres c_i are placed symmetrically on the imaginary axis, with

$$c_i = 0, \; i = -M\frac{N+1}{2},\ldots,-M\frac{N-1}{2}-1,-\frac{M}{2},\ldots,\frac{M}{2}-1,$$

and the remaining $(N-2)M$ non-zero centres limited by

$$max(|E_{k,l}|) = max(\frac{|kd_x v_D|}{1+(kd_x)^2}), \; k=0,\pm 1,\ldots,\pm(\frac{N}{2}-1),$$

which clearly has the maximum $|v_D|/2$, when $kd_x = \pm 1$. N and L_x may very well have values making it impossible to have $kd_x = \pm 1$, i.e.

$$|c_{max}| \leq \frac{|v_D|}{2}$$

hence this upper limit is independent of the spatial resolution, the domain and of $\underline{\underline{\hat{u}}}$. Since $E_{k,l} = E_{k,-l}$, $c_{kM+l} = c_{kM-l}$, hence all the centres lie in pairs of two, which means that only $max(r_{kM+l}, r_{kM-l})$ is interesting, thereby reducing the numbers of radii to be taken into consideration by a factor two. As to the radii, things are more tricky, and therefore we partly investigate these numerically. But first some analytical considerations. Since $|c_{max}| \leq |v_D|/2$, any $r_i \geq |v_D|$ will encircle all the centres, making the discs form a single area. A property of Gershgorin's discs is that if P discs form a separate area, then this area will encircle P eigenvalues, which means that if $r_{max} \geq |v_D|$, then it has no interest, if some of the radii are zero or small. The discs can form a single area in other situations, as long as n discs have $r_i > |v_D|/(2n)$, which makes it likely that the circles form a single area.

In our case the radii of Gershgorin's discs are (the following summation does not have to exclude the term corresponding to $kM+l = nM+m$, since this is zero).

$$\begin{aligned}
r_{kM+l} &= \sum_{n,m} \left|\frac{F_{k,l}}{NM}\hat{u}_{r,s}(A_{n,m}B_{r,s} + B_{n,m}A_{r,s} - C_{n,m}D_{r,s} - D_{n,m}C_{r,s})\right| \\
&= \frac{|F_{k,l}|}{NM}\sum_{n,m} |\hat{u}_{r,s}||A_{n,m}B_{r,s} + B_{n,m}A_{r,s} - C_{n,m}D_{r,s} - D_{n,m}C_{r,s}| \\
& \quad k = -\frac{N}{2}, -\frac{N}{2}+1, \ldots, \frac{N}{2}-1, \quad l = -\frac{M}{2}, -\frac{M}{2}+1, \ldots, \frac{M}{2}-1 \qquad (12.7) \\
& \quad r = \begin{cases} k-n, & -\frac{N}{2} \leq k-n < \frac{N}{2} \\ k-n+N, & k-n < -\frac{N}{2} \\ k-n-N, & k-n \geq \frac{N}{2} \end{cases} \\
& \quad s = \begin{cases} l-m, & -\frac{M}{2} \leq l-m < \frac{M}{2} \\ l-m+M, & l-m < -\frac{M}{2} \\ l-m-M, & l-m \geq \frac{M}{2}, \end{cases}
\end{aligned}$$

This shows that whenever $F_{k,l} = 0$, then $r_{kM+l} = 0$. Since the original (not antialiased) $\underline{\underline{F}}$ is

$$F_{k,l} = \frac{-1}{1+(kd_x)^2+(ld_y)^2} < 0, \; k = -\frac{N}{2}, \ldots, \frac{N}{2}-1, \; l = -\frac{M}{2}, \ldots, \frac{M}{2}-1,$$

then the zero elements of $\underline{\underline{F}}$ will be those and only those reset by the antialiasing. The number of zeros in $\underline{\underline{F}}$ is $NM - (1+2[N/3])(1+2[M/3])$ or roughly $5NM/9$, i.e. more than half the radii do not have to be considered (others may be zero as well, but that is difficult to predict), at least not as long as the discs form a single area. We also note that the radii are independent of v_D, which means that a small v_D will pull the discs together, while a large v_D will separate them. Since this is about all we can say about these radii, we will find them numerically. The Matlab m-file in appendix C is a straightforward (read: not very efficient) implementation of equation 12.7. The results in table 12.1 are found using $\underline{u}$ from equation 12.2, for $N = M = 512$ only 25% of the radii are found and r_{max} is just a local maximum.

$N = M$	r_{max}
8	0.0048
16	0.097
32	0.82
64	3.37
128	10.9
256	27.5
512	> 60

Table 12.1: Maximum radii of Gershgorin discs, using the Gaussian vortex

Using $v_D = 0.1$, and remembering that $|c_{max}| < |v_D|/2$, we see that when using a realistic resolution, the discs will form a single area, and that this area is limited within a range of 27.5, for $N = M = 256$, and probably not much more than 60 for $N = M = 512$. On the basis of these small radii, we conclude that ERK4 will allow a reasonable step length, at least as long as the eigenvalues of the solution at $t > 0$ do not move further away from zero.

If we want to examine these radii at a higher resolution, and possibly use $\underline{\underline{\hat{u}}}(t)$, $t > 0$, we will have to use the CM. We will therefore find a strategy for the calculation of the radii that will run at least fairly efficient on the CM. First we notice that $\underline{\underline{\hat{u}}}_{r,s} = \underline{\underline{\hat{u}}}^*_{-r,-s}$, and that $\underline{\underline{A}}_{n,m}\underline{\underline{B}}_{r,s} = -\underline{\underline{A}}_{n,m}\underline{\underline{B}}_{-r,-s}$, and $\underline{\underline{C}}_{n,m}\underline{\underline{D}}_{r,s} = -\underline{\underline{C}}_{n,m}\underline{\underline{D}}_{-r,-s}$, thus

$$\begin{aligned}
r_{kM+l} &= \frac{|F_{k,l}|}{NM}\sum_{n,m}|\hat{u}_{r,s}||A_{n,m}B_{r,s} + B_{n,m}A_{r,s} - C_{n,m}D_{r,s} - D_{n,m}C_{r,s}| \\
&= \frac{|F_{k,l}|}{NM}\sum_{n,m}|\hat{u}_{-r,-s}||A_{n,m}B_{-r,-s} + B_{n,m}A_{-r,-s} \\
&\quad -C_{n,m}D_{-r,-s} - D_{n,m}C_{-r,-s}|
\end{aligned}$$

where k, l, r and s are defined as above. Suppose we wanted to use this to find r_{kM+l}, $k = l = 0$. Then $r = -n$ and $s = -m$ and therefore

$$\begin{aligned}
r_{kM+l} &= \frac{|F_{k,l}|}{NM}\sum_{n,m}|\hat{u}_{-r,-s}||A_{n,m}B_{-r,-s} + B_{n,m}A_{-r,-s} \\
&\quad -C_{n,m}D_{-r,-s} - D_{n,m}C_{-r,-s}| \\
&= \frac{|F_{k,l}|}{NM}\sum_{n,m}|\hat{u}_{n,m}||A_{n,m}B_{n,m} + B_{n,m}A_{n,m} \\
&\quad -C_{n,m}D_{n,m} - D_{n,m}C_{n,m}|,\ (k,l) = (0,0). \qquad (12.8)
\end{aligned}$$

We notice that since $\underline{\underline{A}}, \ldots, \underline{\underline{D}}$ are purely imaginary, we can just as well use their imaginary parts, as this will simplify the multiplications. When incrementing k, $-r$ would decrement, and likewise for l and $-s$. That means, that if we make a copy of $\underline{\underline{A}}$, $\underline{\underline{B}}$, $\underline{\underline{C}}$, $\underline{\underline{D}}$, called $\underline{\underline{A}}_c$, $\underline{\underline{B}}_c$, $\underline{\underline{C}}_c$, $\underline{\underline{D}}_c$ (c for circular shift), and shift the elements of these copies and $\underline{\underline{\hat{u}}}$ forward once, along the axis of incrementation, then

$$\begin{aligned}
r_{kM+l} &= \frac{|F_{k,l}|}{NM}\sum_{n,m}|\hat{u}_{n,m}||A_{n,m}B_{cn,m} + B_{n,m}A_{cn,m} \\
&\quad -C_{n,m}D_{cn,m} - D_{n,m}C_{cn,m},| \qquad (12.9)
\end{aligned}$$

with the restriction $(k,l) = (0,1)$ or $(k,l) = (1,0)$. When continuing this process, k or l will reach $N/2-1$ or $M/2-1$. If the next value of k or l at this point is chosen as $-N/2$ or $-M/2$ respectively, and $\underline{\underline{A}}_c$, $\underline{\underline{B}}_c$, $\underline{\underline{C}}_c$, $\underline{\underline{D}}_c$ and $\underline{\underline{\hat{u}}}$ is shifted once as in the beginning, then r_{kM+l} will be calculated for negative values of k and l as well. In a computer implementation the elements with negative indexes, would be placed in continuation of the elements with positive indexes, i.e. the first and third, and the second and fourth quadrants of $\underline{\underline{A}}$ will be shifted. This makes it very simple to implement the above (figure 12.1 is pseudo FORTRAN-90). As a start we would initialize A, B, C, D to the imaginary part of $\underline{\underline{A}}$, $\underline{\underline{B}}$, $\underline{\underline{C}}$ and $\underline{\underline{D}}$, F to $-\underline{\underline{F}}/N/M$ and u to $|\underline{\underline{\hat{u}}}|$, which would make all the arrays real-valued, and F positive.

```
r = 0
A_c = A
B_c = B
C_c = C
D_c = D
for k = 0, N - 1
  for l = 0, M - 1
    if F(k,l) <> 0 then
      r(kM + l) = F(k,l) * sum(u * abs(AB_c + A_cB - CD_c - D_cC))
    endif
    u = cshift(u DIM = 2, SHIFT = 1)
    A_c = cshift(A_c, DIM = 2, SHIFT = 1)
    B_c = cshift(B_c, DIM = 2, SHIFT = 1)
    C_c = cshift(C_c, DIM = 2, SHIFT = 1)
    D_c = cshift(D_c, DIM = 2, SHIFT = 1)
  end
  u = cshift(u DIM = 1, SHIFT = 1)
  A_c = cshift(A_c, DIM = 1, SHIFT = 1)
  B_c = cshift(B_c, DIM = 1, SHIFT = 1)
  C_c = cshift(C_c, DIM = 1, SHIFT = 1)
  D_c = cshift(D_c, DIM = 1, SHIFT = 1)
end
```

Figure 12.1: Pseudo program to calculate radii of Gershgorin discs.

On a CM, the cshifts will execute very fast (since the variables would be layout as news); the only thing taking time would be the calculation in the if-statement. The elementwise matrix operations will all be local, causing the CM-200 to yield it's maximum flop-rate, while the sum-operation can only be partly paralleled, requiring a great deal of communication (it is done by successively adding one half of the elements to the other). From this we estimate that the CM will execute this algorithm yielding 200 Mflops/s. The order of magnitude for this algorithm is $O(N^2M^2)$, and since about 5/9 of F is zero, and since *sum* of a $N \times M$-array costs about NM flops, the total number of flops is roughly $4NM/9(1+7)NM$ or $4N^2M^2$ flops. With $N = M = 256$, this is 16 Gflops, which would take 80 CPU-seconds, which is OK. Unfortunately, we have no the time to implement the algorithm, and therefore we will have to base our assumptions of the radii on the results in table 12.1.

12.6 Testing the Improved Solver.

Applying equation 12.1 to the improved solver with $L_x = L_y = 25$, $N = M = 64$, $t_{max} = 400$ and $h = 1$ results in a stable solution, with a behavior fitting the description in [12]. We have therefore investigated the solver's response to this equation more thoroughly.

12.6.1 Relating the Solutions to Physics.

To ensure that we really are solving the Hasegawa-Mima equation, we have made some qualitative observations on the solution, first using a Gaussian vortex ($u(0)$ in equation 12.1) with a negative amplitude. From [12] we know that this vortex is not a stable solution, and as a consequence of this waves will be emitted from the vortex, and the amplitude of the solution will increase (towards zero), which can be seen using the solver. This disintegration has an important consequence: the center of the vortex is negative and increasing, hence $\nabla^2 u$ is positive and decreasing, and since (this is another property shown in [12])

$$\forall x, y, t : \ \frac{\partial}{\partial t}(\nabla^2 u - u + v_D y) = 0,$$

the vortex will have to move downwards to maintain this so-called potential vorticity. As u disintegrates, $\nabla^2 u - u$ will decrease, making the downward movement decrease. A third property shown in [12] is that the center of the vortex will have the speed $-v_D$ along the x-axis, which means that the vortex initially should move downwards and left, and later on just move left. This movement is clearly seen using the solver.

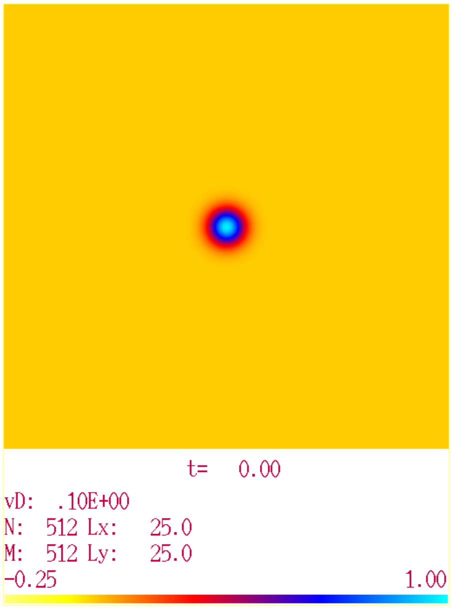

Figure 12.2 The initial Gaussian vortex.

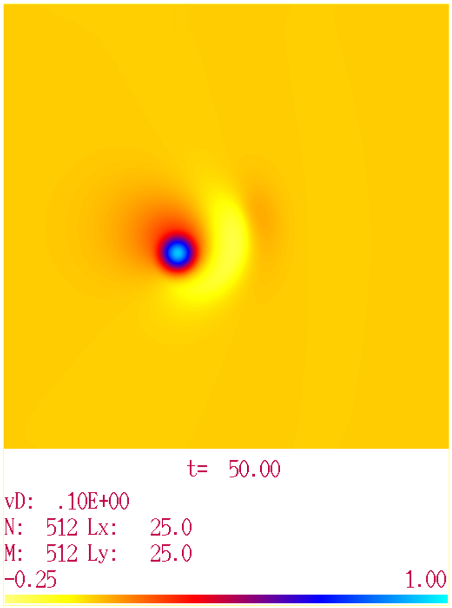

Figure 12.3 The vortex moving left downwards, while emitting waves.

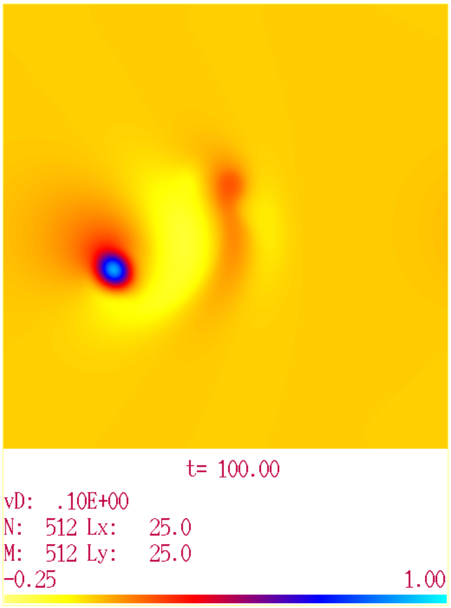

Figure 12.4 The vortex moving further left downwards.

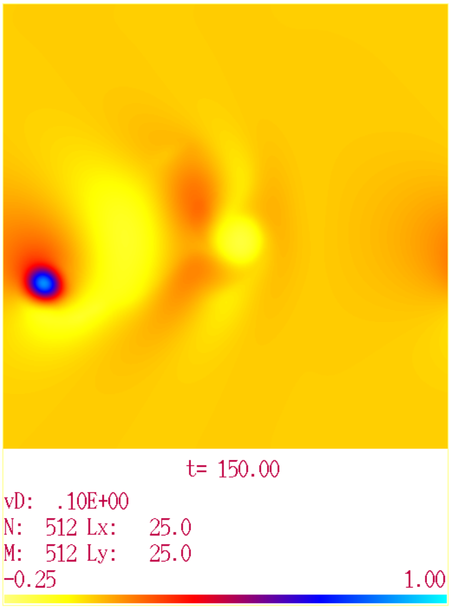

Figure 12.5 The vortex now moving left, still emitting waves.

12.6.2 The Stability and Floprate of the Solver.

As seen in subsection 12.5.1 it is the eigenvalues with the non-positive real-parts that determine the maximum step length, and since the centers of the Gershgorin's discs, found in section 12.5.1, lie from $-c_{max}/2$ to $c_{max}/2$, $c_{max} = |v_D|/2$ on the imaginary axis, it is likely that it is the eigenvalues located on the imaginary axis which restricts the step-length. This assumption makes it possible to predict a step length giving a stable solution. The stability domain of ERK4 stretches out to ± 2.8 on the imaginary axis, which means that any

$$h < h_{stab} = \frac{2.8}{max(|c_i| + r_i)}, \;\; i = 1, \ldots, NM$$

should give a stable solution (c_i, r_i is the i'th Gershgorin's disc). The maximum distance between to Gershgorin centers is v_D, and since we use $v_D = 0.1$, the centers lie so close to each other (compared to the maximum radii at higher resolutions) that we might as well use the criterion

$$h < h_{stab} = \frac{2.8}{c_{max} + r_{max}}, \;\; c_{max} = \frac{|v_D|}{2}, \tag{12.10}$$

where r_{max} can be found in table 12.1). This stability criterion should be understood in the way that $h < h_{stab}$ ensures stability, but $h > h_{stab}$ does not ensure instability. Table 12.2 shows h_{stab} for a number of resolutions, along with some measurements of step lengths giving both stable and unstable solutions. These results are found using the Gaussian vortex, $v_D = 0.1$ and $N/M = L_x/L_y$. For a number of resolutions we have found a step length h_{max} giving stability for $t < t_{max}$. At $t = t_{max}$ the vortex has disintegrated, leaving only waves unable to cause instability, thus we assume that (with the vortex), if the solution is stable for $0 < t < t_{max}$, then it will be stable for $t > 0$. h_{instab} is a step length giving instability, thus the maximum step length lies between h_{max} and h_{instab}. Comparing the measurements with h_{stab} shows that as the resolution grows, h_{stab} tend to fit into the gab between h_{max} and h_{instab}. We can think of two explanations for this. The first is that the movement of the eigenvalues does not increase significantly as the resolution grows, which means that the estimate h_{stab} gets better as resolution grows. This would be a convenient property, and further research of how the eigenvalues depend on time and resolution is therefore desirable. The latter is that the movement of the eigenvalues increases with resolution, in this case we would need information of the eigenvalues at $t > 0$. This could be done either by use of the Gershgorin discs, or calculation of the eigenvalues by brute force (as we write this, the CMSSL v. 3.0, with routines for finding eigenvalues, are being installed on the CM).

N	M	L_x	h_{stab}	h_{max}	t_{max}	h_{instab}
32	32	25	3.2	10	800	12
64	64	25	0.81	1	800	3/2
128	128	25	0.25	1/3	800	1/2
256	256	25	0.10	1/7	800	1/6
512	256	50	N/A	1/10	800	1/6
512	512	50	<0.05	1/20	800	1/15
1024	512	100	N/A	1/20	900	1/15

Table 12.2: Maximum step-length solving the HM-equation with ERK4.

The efficiency of the program is bound by the efficiency of the FFT's, since the majority of the calculations are FFT's. The rest of the calculations are local operations, therefore we will expect the flop-rates for a $N \times M$-job to be somewhat higher than the corresponding flop-rate for a $N \times M$ real to complex FFT. As seen in table 12.3 (based on appendix H) this is the case, but still, $N = M = 1024$ yields only 88 Mflops/s, which is a modest 15% of the peak performance. On the basis of this we conclude that due to the vast communication overhead of the 2D-FFT's the spectral method does not at all exploit the full potential of the Connection Machine. As an alternative to the CM, the spectral method could be implemented on a non distributed system, for instance a vector processor, which do FFT's at maximum performance. This is however not recommendable. The reason why is that the design of today's most powerful vector processors is facing the limits posed by fundamental physics (the minimum possible size of communication lines, and the speed of light), while massively parallel computers still have an enormous potential for increasing their computational capacity. To put it short, the present and the future massively parallel computers are superior to the vector processors, and it is therefore acceptable to implement methods with an efficiency of 15% on a massively parallel computer.

N	M	CM-elapsed-time/step [s]	Mflops/s
128	128	0.20	37
256	128	0.27	57
256	256	0.49	67
512	128	0.49	67
512	256	0.92	75
512	512	1.89	76
1024	128	0.89	77
1024	256	1.8	80
1024	512	3.5	86
1024	1024	7.1	88

Table 12.3: Flop-rates solving the HM-equation with ERK4.

Chapter 13

Concluding Remarks.

This document describes our work with the solution of the initial value problem for one and two dimensional nonlinear partial differential equations. The results can be summarized as:

- The spectral method has been implemented on the CM-200 and has proved to present reliable and useful solutions when used in conjunction with Runge Kutta methods.
- The second order semiimplicit Runge Kutta integration improved with Newton-Raphson iteration is an efficient method for integrating equations describing one dimensional stiff systems, like the Korteweg de Vries equation. The method, in our implementation, is however in the case of two (or for that matter three) dimensional problems only useful for equations completely unfit for other methods of integration.
- The results on extending the KdV equation by at fifth order derivative rejects a part of the theory presented by Kawahara. On the other hand does all the numerical computations fit in with the theory of Karpman. One should be aware that numerical results can not be considered proof and because of this we can only conclude that Karpman *seems* to be right.
- The Hasegawa-Mima equation was found to be an example of a two dimensional problem well suited for the simple fourth order explicit Runge Kutta integration.
- There is a fundamental difficulty in implementing a global method like the spectral method efficiently on a distributed system like the Connection Machine. In spite of this we conclude that the computational superiority of these systems compared to conventional central systems e.g. vector processors, nevertheless favors the implementation of the spectral method on a distributed system such as the CM-200.

We have a number of items appropriate for further investigation. The fact that we conclude that the spectral method should be implemented on a massively parallel computer even with a significant loss of efficiency, motivates a comparison with finite difference methods, since they easily implement efficiently on the Connection Machine.

We suggest further analysis of the eigenvalues of the Hasegawa-Mima equation, in order to find a method of integration with an optimum step length. The success of using the explicit fourth order Runge Kutta method in the Hasegawa-Mima equation leads us to suggest an improvement of this method, by extending it to use a variable step length, or by adding a fifth term, making it possible to (eventually dynamically) adapt the region of stability to fit the moving eigenvalues of the equation. The performance of the second order semiimplicit Runge Kutta method could be improved significantly by replacing the QR factorization by an iterative solver, and a development of such a solver is therefore desirable. Even better would be a solver that based on the properties of the Jacobian is specifically tailored for the system to be solved. An example of such a property is that if any row of the KdV Jacobian is shifted cyclic once, it will equal its predecessor.

It could also be interesting to investigate new methods for solving ordinary differential equations based on the use of multiple simultaneously evaluations of the ODE's right hand side, since the Connection Machine and other massively parallel computers probably would be well suited for this type of computations.

It would furthermore be relevant to implement the methods with use of the real to complex fast Fourier transforms found in the new CMSSL library v. 3.0. Since our own real to complex transform operates at up to 93% of the theoretically possible, we predict that this will not change the performance of the solvers significantly.

Chapter 14

Acknowledgments.

We would like to thank our supervisor Professor P.G.Thomsen for his commitment and inspiring guidance regarding the numerical questions associated with the implementation of the algorithm. We would also like to express our gratitude to our second supervisor, M.Sc. J.S.Hesthaven for clarifying explanations of the physics described by the investigated equations:

Futhermore we would like to thank the following.

- Kristian Eriksen, who has given valuable linguistic aid.
- Everybody at the Numerical Institute of the Technical University of Denmark, who provided a pleasant atmosphere.
- The application engineers from Thinking Machines Corporation supporting the Connection Machine, who at all times have been helpful, especially whenever the CM behaved in an unpredictable manner.
- The staff at UNI•C has shown an overwhelming willingness to help with all kinds of computer related problems.

Finally we thank whoever made nature pose these challenging nonlinear equations.

Bibliography

[1] Elliott Rao. *FAST TRANSFORMS Algorithms, Analyses, Applications.* Academic Press, 1982.

[2] E. Oran Brigham. *The Fast Fourier Transform and its applications.* Prentice-Hall International, Inc., 1988.

[3] Thinking Machines Corp.. *CMSSL v. 2.2 manual.*

[4] S. Lennart Johnsson, Robert L. Krawitz, Roger Frye and Douglas MacDonald. *Cooley-Tukey FFT on the Connection Machine.* Thinking Machines Corp., 1989.

[5] Thinking Machines Corp.. *Connection Machine Model CM-2 Technical Summary.* Thinking Machines Corp., 1990.

[6] Norman J. Zabusky. *A Synergetic Approach to Problems of Nonlinear Dispersive Wave Propagation and Interaction.* Nonlinear partial differential equations. Academic Press, 1967.

[7] J. D. Lambert. *Computational Methods in Ordinary Differential Equations.* John Wiley & Sons, 1973.

[8] Kendall E. Atkinson. *An Introduction to Numerical Analysis.* John Wiley & Sons, 1989.

[9] C. Canuto, M.Y. Hussaini, A. Quarteroni, T.A. Zang. *Spectral Methods in Fluid Dynamics.* Springer-Verlag, 1987.

[10] Wilkinson, J.H. (1965): *The algebraic Eigenvalue problem.* Oxford University press, London.

[11] Hairer, E., Wanner, G. (1991): *Solving ordinary differential Equations II.* Springer Verlag, Berlin.

[12] Hesthaven, J.S. (1991): *Dynamics of Coherent Structures in the Hasegawa-Mima Equation.* M.Sc. Dissertation, Technical University of Denmark.

[13] Takuji Kawahara. *Oscillatory Solitary Waves in Dispersive Media.* Journal of the Physical Society of Japan, Vol 33, No. 1, July, 1972.

[14] V.I. Karpman. *On the existence of Kawahara solitons.* Risø-R-611.

Appendix A

FFT Performance.

In this appendix we will present some time and performance measurements for the CMSSL FFT as well as for our own real-to-complex FFT in 1D and 2D. For the 2D matter we will also present timings for the inverse transform. All timings are in seconds and all performance measurements are in Mflops/sec.. The number of flops are calculated on basis of the Cooley-Tukey radix-2 algorithm, with is not entirely correct since the CMSSL library also uses radix 4 and 8 algorithms. The deviation in the use of radix-2 for the amount of work on the higher radix algorithms is pretty small though. The flops required to do a real-to-complex FFT by use of a half length complex-to-complex FFT is calculated by use of some of the information in appendix H.

A.1 One-dimensional Transforms.

We here present timings for the one-dimensional transforms.

Real-to-complex FFT, output in normal order.				
Size	Minimum	Mean	Variance	Mflops/s
128	.0031459	.0031772	.0000000	0.4
256	.0032693	.0033018	.0000000	0.8
512	.0035785	.0036066	.0000000	1.6
1024	.0037354	.0037686	.0000000	3.5
2048	.0040768	.0050664	.0000947	7.0
4096	.0049870	.0050550	.0000000	13
8192	.0067538	.0068772	.0000000	20
16384	.0101371	.0101400	.0000000	29
32768	.0188239	.0188264	.0000000	33
65536	.0332446	.0332482	.0000000	40
131072	.0661790	.0671437	.0000787	43
262144	0.1299407	0.1318806	.0001408	46

Table A.1: 1D Real-to-complex FFT, normal output.

Real-to-complex FFT, output in bit-reversed order.				
Size	Minimum	Mean	Variance	Mflops/s
128	.0024021	.0024073	.0000000	0.5
256	.0025293	.0025315	.0000000	1.0
512	.0026546	.0026563	.0000000	2.2
1024	.0028152	.0028180	.0000000	4.6
2048	.0029487	.0029505	.0000000	9.7
4096	.0038760	.0039669	.0000000	16
8192	.0051153	.0052480	.0000000	26
16384	.0069292	.0069320	.0000000	42
32768	.0131559	.0131599	.0000000	47
65536	.0260181	.0260223	.0000000	51
131072	.0516218	.0525493	.0000785	55
262144	0.1027338	0.1037839	.0000842	58

Table A.2: 1D Real-to-complex FFT, bit-reversed output.

CMSSL FFT, output in normal order.				
Size	Minimum	Mean	Variance	Mflops/s rate
128	.0023232	.0023579	.0000000	0.9499677
256	.0026609	.0026635	.0000000	1.9222307
512	.0027913	.0028250	.0000000	4.0777751
1024	.0031224	.0031244	.0000000	8.1933503
2048	.0033166	.0033191	.0000000	16.9681588
4096	.0040196	.0041224	.0000000	29.8072209
8192	.0057285	.0057316	.0000000	46.4507421
16384	.0109290	.0109319	.0000000	52.4556045
32768	.0177129	.0186826	.0000923	65.7722856
65536	.0356441	.0356477	.0000000	73.5373427
131072	.0682170	.0691827	.0000913	80.5195041
262144	0.1411083	0.1414926	.0000131	83.3716887

Table A.3: 1D CMSSL FFT, normal output.

CMSSL FFT, output in normal order.				
Size	Minimum	Mean	Variance	Mflops/s rate
128	.0015524	.0015872	.0000000	1.4112201
256	.0016766	.0017122	.0000000	2.9902577
512	.0018392	.0018740	.0000000	6.1470260
1024	.0019607	.0019953	.0000000	12.8297779
2048	.0022058	.0022078	.0000000	25.5086674
4096	.0026627	.0026787	.0000000	45.8715596
8192	.0032666	.0032693	.0000000	81.4348986
16384	.0060119	.0060151	.0000000	95.3318894
32768	.0118077	.0118099	.0000000	104.0479549
65536	.0233610	.0233644	.0000000	112.1979681
131072	.0461580	.0470581	.0000792	118.3762150
262144	.0940416	.0943831	.0000111	124.9850847

Table A.4: 1D CMSSL FFT, bit-reversed output.

CMSSL FFT, output in bit-reversed order. Multiple FFTs simultaneously.			
No.	Size	Time	Mflops/s rate
1	16	.12151E-02	0
4	16	.11779E-02	1
16	16	.11779E-02	2
64	16	.11778E-02	9
256	16	.10250E-02	40
1024	16	.42760E-03	383
4096	16	.17113E-02	383
16384	16	.70419E-02	373
65536	16	.29567E-01	355
262144	16	.11826E+00	354
1	32	.13029E-02	0
4	32	.13031E-02	1
16	32	.13026E-02	5
64	32	.13574E-02	19
256	32	.22610E-03	453
1024	32	.89620E-03	457
4096	32	.36872E-02	444
16384	32	.15047E-01	436
65536	32	.61724E-01	425
262144	32	.24687E+00	425
1	64	.14277E-02	1
4	64	.14283E-02	3
16	64	.14279E-02	11
64	64	.18728E-02	33
256	64	.42680E-03	576
1024	64	.17146E-02	573
4096	64	.69783E-02	563
16384	64	.28704E-01	548
65536	64	.11796E+00	533
1	128	.15528E-02	1
4	128	.15522E-02	6
16	128	.16699E-02	21
64	128	.21144E-02	68
256	128	.11219E-02	511
1024	128	.45189E-02	508
4096	128	.18328E-01	501
16384	128	.75780E-01	484
65536	128	.31479E+00	466

Table A.5: 1D CMSSL FFT, Multiple simultaneously FFTs.

CMSSL FFT, output in bit-reversed order. Multiple FFTs simultaneously.			
No.	Size	Time	Mflops/s rate
1	256	.16776E-02	3
4	256	.16772E-02	12
16	256	.20982E-02	39
64	256	.41187E-02	80
256	256	.24926E-02	526
1024	256	.10187E-01	515
4096	256	.41264E-01	508
16384	256	.16824E+00	499
1	512	.18386E-02	6
4	512	.18565E-02	25
16	512	.24712E-02	75
64	512	.84251E-02	88
256	512	.49233E-02	599
1024	512	.20169E-01	585
4096	512	.81686E-01	578
16384	512	.33328E+00	566
1	1024	.19612E-02	13
4	1024	.22950E-02	45
16	1024	.46936E-02	87
64	1024	.17018E-01	96
256	1024	.12056E-01	554
1024	1024	.49251E-01	532
4096	1024	.19911E+00	527
1	2048	.21399E-02	26
4	2048	.28288E-02	80
16	2048	.94346E-02	96
64	2048	.33879E-01	106
256	2048	.26633E-01	541
1024	2048	.10855E+00	531
4096	2048	.43842E+00	526
1	4096	.25890E-02	47
4	4096	.52672E-02	93
16	4096	.18893E-01	104
64	4096	.69899E-01	113
256	4096	.53061E-01	593
1024	4096	.21628E+00	582

Table A.6: 1D CMSSL FFT, Multiple simultaneously FFTs.

CMSSL FFT, output in bit-reversed order. Multiple FFTs simultaneously.			
No.	Size	Time	Mflops/s rate
1	8192	.32667E-02	82
4	8192	.10441E-01	102
16	8192	.37493E-01	114
64	8192	.14223E+00	120
256	8192	.12403E+00	550
1024	8192	.50427E+00	541
1	16384	.60116E-02	95
4	16384	.20762E-01	110
16	16384	.76976E-01	119
64	16384	.28418E+00	129
256	16384	.26850E+00	547
1	32768	.11806E-01	104
4	32768	.41098E-01	120
16	32768	.15625E+00	126
64	32768	.58618E+00	134
256	32768	.53564E+00	587
1	65536	.23361E-01	112
4	65536	.84042E-01	125
16	65536	.31205E+00	134
64	65536	.11927E+01	141
1	131072	.46155E-01	121
4	131072	.17024E+00	131
16	131072	.64179E+00	139
64	131072	.23840E+01	150
1	262144	.94038E-01	125
4	262144	.33990E+00	139
16	262144	.13038E+01	145
1	524288	.19010E+00	131
4	524288	.69734E+00	143
16	524288	.26060E+01	153
1	1048576	.37950E+00	138
4	1048576	.14148E+01	148
1	2097152	.77641E+00	142
4	2097152	.28278E+01	156
1	4194304	.16358E+01	141
1	8388608	.31437E+01	153

Table A.7: 1D CMSSL FFT, Multiple simultaneously FFTs.

A.2 Two-dimensional Transforms.

Here we present timings and performance measurements for the two-dimensional transforms. The arrays transformed are quadratic with a side length of 'size'.

Real-to-complex 2D-FFT. Output in normal order.		
Size	Time	Mflops/s
2	.0012698	-
4	.0027893	-
8	.0038213	-
16	.0052084	-
32	.0055899	2.7
64	.0069728	10
128	.0129673	25
256	.0317118	46
512	0.1312009	50
1024	0.5105331	55

Table A.8: 2D real-to-complex FFT, normal output.

Real-to-complex 2D-FFT. Output in bit-reversed order.		
Size	Time	Mflops/s
2	.0012709	-
4	.0026269	-
8	.0032107	-
16	.0042816	-
32	.0046355	3.3
64	.0057310	12
128	.0086802	37
256	.0231943	63
512	.0890423	73
1024	0.3386630	84

Table A.9: 2D real-to-complex FFT, bit-reversed output.

Complex-to-real 2D-IFFT. Output in normal order.		
Size	Time	Mflops/s
2	.0012882	-
4	.0027489	-
8	.0038069	-
16	.0052343	-
32	.0056114	2.7
64	.0070082	10
128	.0130794	25
256	.0321338	46
512	0.1328270	50
1024	0.5173065	55

Table A.10: 2D complex-to-real IFFT, normal output.

Complex-to-real 2D-FFT. Output in bit-reversed order.		
Size	Time	Mflops/s
2	.0012875	-
4	.0025366	-
8	.0031233	-
16	.0041575	-
32	.0044945	3.4
64	.0055775	13
128	.0084136	38
256	.0225698	65
512	.0861310	75
1024	0.3268901	87

Table A.11: 2D complex-to-real IFFT, bit-reversed output.

CMSSL 2D-FFT. Output in normal order.		
Size	Time	Mflops/s
2	.0003143	.06363
4	.0022366	.07153
8	.0024623	0.38988
16	.0034784	1.47194
32	.0037292	6.86474
64	.0049748	24.70049
128	.0122354	46.86729
256	.0496463	52.80232
512	0.1902763	61.99658
1024	0.7934540	66.07667

Table A.12: 2D CMSSL FFT, normal output.

CMSSL 2D-FFT. Output in bit-reversed order.		
Size	Time	Mflops/s
2	.0002772	.07215
4	.0019820	.08072
8	.0022081	0.43476
16	.0024578	2.08316
32	.0027074	9.45557
64	.0033412	36.77720
128	.0090938	63.05835
256	.0362055	72.40447
512	0.1444433	81.66859
1024	0.5737849	91.37361

Table A.13: 2D CMSSL FFT, bit-reversed output.

CMSSL 2D-IFFT. Output in normal order.		
Size	Time	Mflops/s
2	.0003281	.06095
4	.0022130	.07230
8	.0024384	0.39370
16	.0034911	1.46659
32	.0037061	6.90753
64	.0049080	25.03667
128	.0123354	46.48735
256	.0500322	52.39506
512	0.1918006	61.50388
1024	0.7995305	65.57448

Table A.14: 2D CMSSL IFFT, normal output.

CMSSL 2D-IFFT. Output in bit-reversed order.		
Size	Time	Mflops/s
2	.0003259	.06136
4	.0019942	.08023
8	.0022197	0.43249
16	.0024698	2.07304
32	.0027194	9.41384
64	.0033785	36.37117
128	.0091939	62.37179
256	.0365910	71.64166
512	0.1459689	80.81502
1024	0.5798655	90.41545

Table A.15: 2D CMSSL IFFT, bit-reversed output.

Appendix B

Some visualization utilities.

In order to be able understand our 2D results, we needed a utility that could present snapshots of our problem in a movie-like manner. The way we chose to do this was by having our program store the snapshots in a TIFF-file, and then making a utility which could display this TIFF-file. It turned out, though, that the normal disk connected too the front-end was to slow for displaying the pictures sufficiently fast. We therefore made two utilities - one which transferred the pictures to the Data Vault, and one which played the movie from the Data Vault. The Data Vault is a set off 42 high speed disks connected directly to the Connection Machine by a high speed bus, the CMIO bus. The Data Vault is capable of transferring data at a sustained rate above 25 Mbytes/sec. We will now describe the two utilities 'tiff2dv' and 'dv2movie'.

B.1 The Program, 'tiff2dv'.

The program, tiff2dv, listed in section B.3 transfers a tiff file from the front-end to a data file on the Data Vault. The program has two input parameters, first the tiff file name with extension, and secondly the Data Vault file name **without** extension. What the program does is, to read each image from the tiff file using the tiff library supplied by Thinking Machines Corp., and writes it out to a Data Vault file with extension '.imag'. The picture dimensions are written to the file with extension '.ser' along with the color map of the first picture. Because of this, the program is limited to contain conform pictures with the same color map. Furthermore, the program is limited so that the color map has to be of length 256 which is the standard length when using colors.

The program is written in C* because the most advanced buffered I/O routines for the Data Vault are only usable from C*. Since efficient use of the framebuffer implies the use of fieldwise data storage, there is no reason not to use C*[1].

B.2 The Program, 'dv2movie'.

The program, dv2movie listed in section B.4 displays a Data Vault picture file on the framebuffer, or any X-terminal, as a movie. The input parameters and switches for the program are:

`dv2movie [-repeat] [-enter] [-center] [-times n] file name.`

Where the meaning of commands and switches are,

-repeat	When all the pictures in the file have been displayed, rewind the file and start all over.
-enter	When the first file has been the displayed, ask for the next file name and display this file.
-center	Display the film centered on the framebuffer. This does not work when using an X-terminal.
-times *n*	Display each picture *n* times. Thereby slowing down the speed of the projection.
file name	The name of the Data Vault file (without extension) to be displayed.

The program first opens the '.ser' file and reads the dimensions of the picture along with the color map. Then it allocates memory, initializes the display and it starts reading and displaying the images from the Data Vault.

[1]Slicewise data storage is only available in CM-fortran. Fieldwise is available in all the CM programming languages.

This program is also written in C* for the same reasons as described for the program, 'tiff2dv'.

B.3 Code Listing of 'tiff2dv.cs'.

```
#include <stdio.h>
#include <stdlib.h>
#include <cm/cmsr.h>
#include "cmtiff.h"
#include <string.h>
#include <cm/cm_file.h>
#include <cm/paris.h>
#include <cm/cm_param.h>

#define no_of_proc 8192

unsigned int round2(int n)
/* function which returns the smallest power of 2 larger than n */
  {
  unsigned int i;
  float k;
  i = 2;
  k = n;
  while ((k=k/2) > 1) i*=2;
  return i;
  }

main (ac, av)
  char **av;
  int ac;
  {

  CMSR_image_info_t info;
  int dv_out, height, width, depth, image_no, status;
  char file1[100],file2[100];
  float red[256], green[256], blue[256];

  unsigned int axes_array[2], bufsize;
  CM_field_id_t film,buffer;
  CM_geometry_id_t image_geometry_id;
  CM_vp_set_id_t image_vp_set_id;

  /* Test for correct input */
  if (ac != 3)
    {
    printf("usage: %s input-file output-file \n", av[0]);
    exit(1);
    }

  strcat(strcpy(file1,av[2]),".imag");
  strcat(strcpy(file2,av[2]),".ser");

  info = CMSR_allocate_image_info();

  /* Get picture data */
  status = CMSR_image_get_info(av[1],info,0);

  if (status == -1)
    {
    printf("Error opening TIFF-file!\n");
    exit(1);
    }

  height = CMSR_image_height(info);
  width = CMSR_image_width(info);
  depth = CMSR_image_depth(info);

  /* Read color map */
  CMSR_image_color_map(info,red,green,blue);

  /* Create description file */
  if ((dv_out = CMFS_creat(file2,0666)) == -1)
    {
    printf("Error creating '.ser'-file!\n");
    exit(1);
    }
```

Appendix C

The Computation of the Eigenvalues of the Hasegawa-Mima Equation.

```
function u=make_u0(N,M,Lx,Ly);
% Sets up the initial value for the PDE.
% no. of rows (x-resolution) is N,
% no. of cols (y-resolution) is M.

A = -1;
s2 = 1;
xvec=(-N/2:N/2-1)'*Lx/N;
yvec=(-M/2:M/2-1)*Ly/M;

x=ones(xvec)*yvec;
y=xvec*ones(yvec);

u = A*exp( -( x.^2 + y.^2)/s2 );

function J=jac_2d(N,M,vD,Lx,Ly)
  pi2i = 2*pi*sqrt(-1);
  [A,B,C,D,E,F] = make_AF(N,M,vD,Lx,Ly);
  fu0=fft2(make_u0(N,M,Lx,Ly));
  F=F/N/M;
  J=zeros(N*M);
  r=0;s=0;
  for k=1:N,
    for l=1:M,
      p = k;
      for n=1:N,
        q = l;
        for m=1:M,
          J(r+l,s+m) = (A(n,m)*B(p,q) + B(n,m)*A(p,q) ...
                     -  C(n,m)*D(p,q) - D(n,m)*C(p,q)) ...
                     * F(k,l) * fu0(p,q);
          q=q-1; if (q==0), q=M;end;
        end;
        p=p-1; if (p==0), p=N;end;
        s=s+M;
      end;
      s=0;
      J(r+l,r+l) = E(k,l);
    end;
    r=r+M;
  end;

function [A,B,C,D,E,F]=make_AF(N,M,vD,Lx,Ly);
% sets up the constants used in jac_2d.m and br.m
  i=sqrt(-1);
  pi2Lx = 2*pi/Lx;
  pi2Ly = 2*pi/Ly;
  k=i*fftshift(pi2Lx*[0 1-N/2:N/2-1]');
  l=i*fftshift(pi2Ly*[0 1-M/2:M/2-1]);
  k2=fftshift(pi2Lx*[-N/2:N/2-1]').^2;
  l2=fftshift(pi2Ly*[-M/2:M/2-1]).^2;
  A=k*ones(l);
  C=ones(k)*l;
```

Appendix D

Real Fourier Transform code for CM.

D.1 Code for One-dimensional Real-to-Complex FFT.

```
C     ***************************************************************
C     Subroutines to perform real to complex FFT using CMSSL routines
C     ***************************************************************

      SUBROUTINE realfft_setup(Nh,d1,d2,perm,setup_id,reverse_flag)

C     Calculates constant vectors, setup ids etc.
C     MUST be called prior to calling realfft.
C
C     The input parameters are:
C       Nh           : The length of the packed array.
C       reverse_flag : Describes wether or not to use bit reversal
C                      Valid values are: CMSSL_normal, CMSSL_bit_reversed
C
C     The output parameters are:
C       d1,d2    : Constant vectors of type double complex and length Nh
C       perm     : Permutation vector of type integer and length Nh
C       setup_id : Setup id for CMSSL routines of type integer
C     All outputs are passed to the realfft routine.
C     All vectors MUST by SEND.

      INTEGER :: Nh, setup_id, reverse_flag, ier
      INTEGER, ARRAY(Nh) :: perm
      DOUBLE COMPLEX, ARRAY(Nh) :: d1, d2, cperm
      DOUBLE PRECISION :: pi

      INCLUDE '/usr/include/cm/cmssl-cmf.h'

cmf$  layout d1(:send)
cmf$  layout d2(:send)

cmf$  layout perm(:send)
cmf$  layout cperm(:send)

      pi = 4*ATAN(1.d0)

C     Calculate constant vector according to algorithm.
      d1 = 0.5*DCMPLX(1+SIN(-(pi*[0:Nh-1])/Nh),-COS(-(pi*[0:Nh-1])/Nh))
      d2 = 0.5*DCMPLX(1-SIN(-(pi*[0:Nh-1])/Nh),COS(-(pi*[0:Nh-1])/Nh))

C     Call CMSSL setup routine.
      setup_id = fft_setup(d1,'CTOC',ier)

C     Make permutation vector and bit reverse all required vectors
      IF (reverse_flag .NE. CMSSL_normal) THEN
C       Index vector in incremental order in bit reversed domain.
        cperm=[1:Nh]

C       Transform to normal domain
        CALL fft_detailed(cperm,'CTOC',CMSSL_nop,CMSSL_bit_reversed,
     &                    CMSSL_normal,CMSSL_noscale,
```

Appendix E

Linear PDE Solver code for CM.

E.1 The 1-dimensional Solver.

```
      PROGRAM lin_pde_1d_solver

C     Initial value of solutions are assumed to have 2pi as period.

      implicit none

      INCLUDE '/usr/include/cm/cmssl-cmf.h'
      INCLUDE '/usr/include/cm/display-cmf.h'
      INCLUDE '/usr/include/cm/paris-configuration-fort.h'

      INTEGER :: imax, ier, i, uflag
      INTEGER, PARAMETER :: N=1024, maxdegree = 12
      DOUBLE PRECISION :: pi2
      DOUBLE COMPLEX, ARRAY(N) :: e, u, ru
      DOUBLE PRECISION, ARRAY(maxdegree) :: pde
      LOGICAL :: new_pde
      DOUBLE PRECISION :: h, h_old, per
      REAL :: minu, conu
      INTEGER :: setup_id
      INTEGER, ARRAY(1280,N) :: pic

cmf$  layout  e(:send)
cmf$  layout  u(:send)
cmf$  layout  ru(:send)
cmf$  layout pic(:send,:send)
cmf$  layout pde(:serial)

C     Reset descriptor of actual PDE
      e = 0
      DO i=1,maxdegree
        pde(i) = 0
      END DO

      pi2 = 8 * atan(1.d0)
      setup_id = fft_setup(u,'CTOC',ier)
      CALL FSR_select_display_menu(0, 1280,N)

      PRINT *,'Enter no. of periods: '
      READ *, per

      h_old = 0

      DO WHILE (per .NE. 0)

        CALL make_pde(maxdegree,pde,new_pde)

        PRINT *,'Enter no. of steps: '
        READ (*,'(I4)') imax
        DO WHILE (imax .GT. 1279)
          PRINT *,'Enter no. of steps: (max 1279) '
          READ (*,'(I4)') imax
        END DO
```

Appendix F

Results of the KdV programs.

This chapter shows the computational results calculated for the KdV equation.

$h/(2\pi)$	Max. error using $u\partial u/\partial x$	Max. error using $\partial u^2/(2\partial x)$
1.57E-2	1.10E-2	1.09E-2
3.14E-2	2.85E-3	2.85E-3
6.28E-2	8.44E-4	8.43E-4

Table F.1: $u\partial u/\partial x$ versus $\partial u^2/(2\partial x)$ under one period of displacement ($\alpha = 1,\ N = 64,\ v = 10,\ \epsilon = 10^{-6}$).

$h/(2\pi)$	Max. error
3E-7	2.42E-7
1E-7	8.00E-8
1E-8	7.99E-9
1E-9	7.99E-10
1E-10	7.99E-11

Table F.2: Single step of ERK2 with single soliton of speed 100 ($\alpha = 1,\ N = 128$).

$h/(2\pi)$	Max. error
2E-6	1.61E-6
1E-6	8.09E-7
1E-7	8.00E-8
1E-8	7.99E-9
1E-9	7.99E-10

Table F.3: Single step of ERK4 with single soliton of speed 100 ($\alpha = 1,\ N = 128$).

$h/(2\pi)$	Max. error
4E-4	2.40E-1
2E-4	5.95E-2
1E-4	8.99E-3
5E-5	1.20E-3
1E-5	1.41E-5
1E-6	8.14E-7
1E-7	8.00E-8
1E-8	7.99E-9

Table F.4: Single step of SIRK2 with single soliton of speed 100 ($\alpha = 1, \quad N = 128, \quad \epsilon = 10^{-8}$).

Method	$h/(2\pi)$	Max. error
RK2	2E-5	2.69E-4
	1E-5	2.87E-4
	4E-6	2.92E-4
RK4	8E-4	2.95E-4
	4E-5	1.36E-4
	1E-5	2.93E-4
SIRK2	1E-3	1.09E-2
	4E-4	2.00E-3
	1E-4	3.96E-4

Table F.5: One period displacement of single soliton with speed 10. ($\alpha = 1, \quad N = 32, \quad \epsilon = 10^{-8}$).

N	Method	Max h stable	Min h unstable
8	ERK2	1.57E-1	2.09E-1
16	ERK2	2.86E-3	3.14E-3
32	ERK2	1.90E-4	2.02E-4
8	ERK4	3.14E-1	6.28E-1
16	ERK4	2.09E-2	2.51E-2
32	ERK4	2.51E-3	3.14E-3
64	ERK4	2.73E-4	3.14E-4
32	SIRK	$\leq$1.27E0	N/A
64	SIRK	3.14E-1	6.28E-1
128	SIRK	3.14E-1	6.28E-1
256	SIRK	3.14E-1	6.28E-1
512	SIRK	3.14E-1	6.28E-1

Table F.6: Stability of anti aliased runs ($\alpha = 1 \quad v = 1$).

N	Max. error
8	4.12E-4
16	8.17E-4
32	4.98E-4
64	6.03E-6
128	7.63E-8
256	7.62E-8
512	7.62E-8

Table F.7: Spatial resolution vs. maximum error. Single step using SIRK2 ($\alpha = 1, \quad v = 100/3, \quad \epsilon = 10^{-8}, \quad h = 2\pi 10^{-5}$)

$h/(2\pi)$	Max. error
4E-4	2.58E-1
2E-4	5.95E-2
1E-4	8.99E-3
5E-5	1.28E-3
1E-5	1.41E-5
1E-6	8.14E-7
1E-7	8.00E-8
1E-8	7.99E-9

Table F.8: Single step of SIRK2 with single soliton of speed 100 and without anti aliasing ($\alpha = 1, \quad N = 128, \quad \epsilon = 10^{-8}$).

N	Method	Max h stable	Min h unstable
8	ERK2	5.24E-3	6.28E-3
8	ERK4	4.19E-2	-
16	ERK2	4.19E-4	4.83E-4
16	ERK4	5.24E-3	5.71E-3
32	ERK4	6.83E-4	7.14E-4

Table F.9: Stability of non anti aliased runs ($\alpha = 1 \quad v = 1$).

ϵ	Max. error	mean of no. of ite.
1E-8	1.09E-2	11.9
1E-6	1.09E-2	8.0
1E-4	1.08E-2	4.6
1E-3	1.19E-2	2.8
1E-2	2.22E-2	1.8
1E-1	3.34E-2	1.0

Table F.10: One period of displacement of soliton with speed 10 using SIRK2. ($\alpha = 1, \quad N = 64$).

$h/(2\pi)$	Initial guess	Mean of no. of ite.
1.26E-1	Euler	7.0
	RK2	∞
	Lin. approx.	4.1
6.28E-2	Euler	5.3
	RK2	∞
	Lin. approx.	3.4
3.14E-2	Euler	4.2
	RK2	∞
	Lin. approx.	2.8
6.28E-3	RK2	4.9

Table F.11: The use of different initial guess for SIRK2. ($\alpha = 1, \quad v = 1 \quad N = 64, \quad \epsilon = 10^{-5}$).

no. of steps a period	Mean of no. of ite.
55	18.40
60	13.03
80	8.90
100	8.00
200	5.57
400	4.26

Table F.12: The use of different step lengths for SIRK2. ($\alpha = 1, \quad v = 10 \quad N = 64, \quad \epsilon = 10^{-6}$).

Appendix G

Code KdV eq. Solver.

```
      PROGRAM kdv_program
C     Program to solve the KdV eq. in the frequency domain
C     using different Runge Kutta methods.

      implicit none

      INCLUDE '/home/numi/nmibtr/cmssl/cmssl-cmf.h'
      INCLUDE '/usr/include/cm/display-cmf.h'

      INTEGER          :: N, N_old, step, stride, nosech2,
     &                     ini_flag, mat_flag, int_flag
      CHARACTER*30     :: matfilename
      DOUBLE PRECISION :: pi2, h, per, epsi
      REAL, ARRAY(256) :: red,green,blue,trin

cmf$  layout red(:serial)
cmf$  layout green(:serial)
cmf$  layout blue(:serial)

C     Disable safety in order to increase performance
      CALL CM_set_safety_mode(0)

      pi2 = 8*ATAN(1.d0)
      N_old = -1

C     Enter the spatial resolution, N
      PRINT *,'Enter N (max 1024) '
      READ (*,'(I4)') N

      DO WHILE (N .NE. 0)

C       Setup the display or resetup the display
        IF (N .NE. N_old) THEN
          IF (N_old .NE. -1) THEN
            CALL FSR_deselect_display
          ENDIF
          N_old = N
          CALL FSR_select_display_menu(8,1280,N+150)
          CALL FSR_set_display_color_map('rainbow')
          CALL FSR_DISPLAY_READ_COLOR_MAP(red,green,blue,256)
          red(1)=1
          green(1)=1
          blue(1)=1
          CALL FSR_DISPLAY_WRITE_COLOR_MAP(red,green,blue,256)

        ENDIF

C       Enter for how long time the solution should be
C       calculated.
        PRINT *,'Enter no. of periods (relative to speed 1): '
        READ *,per

C       Enter the number of interacting solitons,
C       0 means cosine as initial function.
C       -1 means triangle as initial function.
        PRINT *,
     &   'Enter no. of solitons (0 for cosine or triangle): '
        READ *,nosech2
```

Appendix H

Flops Required to Solve the HM-Equation.

The following shows the flops required to calculate a time step, using ERK4 to integrate the HM-equation in the Fourier domain with a given spatial resolution. The required flops to do a $N \times M$ real to complex FFT using our own FFT-routine are also presented to put the numbers in perspective.

$\log_2(NM)$	Mflops/2DFFT	Mflops/timestep
10	0.015	0.36
11	0.033	0.76
12	0.071	1.63
13	0.15	3.47
14	0.32	7.34
15	0.69	15.5
16	1.5	32.6
17	3.1	68.6
18	6.5	144
19	14	300
20	29	627

Remembering that our algorithm represents a NM resolution as a $NM/2$ complex array, the flops are found as follows. A complex addition is considered 1 flop, a complex multiplication is considered 3 flops, since the floating point processors of the CM does complex addition and multiplication at this rate, compared to real operations. This leads to $5/2N\log_2(N)$ flops for the Cooley-Tukey implementation of the complex to complex N-length FFT, as seen in [4]. A $N \times M/2$2D complex FFT will therefore cost

$$N\frac{5}{2}\frac{M}{2}\log_2\frac{M}{2} + \frac{M}{2}\frac{5}{2}N\log_2 N$$

flops which means that our real to complex FFT costs

$$NM(\frac{5}{4}\log_2(MN) - \frac{5}{4}) + \frac{7NM}{2}$$

flops, this can be found in the mid column. By simply inspecting the code for the HM-solver, we see that the evaluation of the RHS of the ODE derived from the HM-equation costs 5 FFT's and $24NM/2$ flops, and the ERK4 time step costs four evaluations of the ODE's RHS plus $10NM/2$, which means that a timestep using ERK4 in total costs $NM(25\log_2(NM) + 98)$ flops, hence the order of magnitude for a time step is $O(NM\log_2(NM))$.

Appendix I

Code for Hasegawa Mima Solver.

```
      PROGRAM PDE_2D_solver

C     Solves the ODE with du/dt given in the routine ODE_RHS
C     where u is in the Fourier-domaine.
C     Initial value of u (in the time-domain) is computed in make_u0
C     ODE-specific constants used in ODE_RHS are made in make_ODEconst

      implicit none

      INCLUDE '/usr/include/cm/display-cmf.h'
      INCLUDE '/usr/include/cm/paris-configuration-fort.h'

      INTEGER :: N, M, N_old, M_old, t_res, single_flag, tiff_flag, pic_stride, i
      DOUBLE PRECISION :: t, Lx, Ly, vD, pi2
      REAL, ARRAY(256) :: red,green,blue,trin
      CHARACTER*20 :: filename

cmf$  layout red(:serial)
cmf$  layout green(:serial)
cmf$  layout blue(:serial)
cmf$  layout trin(:send)

      N_old = -1
      M_old = -1

      PRINT *,'Enter N: (Must be a power of 2)'
      READ *,N

      DO WHILE ((N .GT. 0) .AND. (N .LE. 1024))

        PRINT *,'Enter M: (Must be a power of 2)'
        READ *,M

        PRINT *, 'Enter length of x-domain: '
        READ *, Lx

        PRINT *, 'Enter length of y-domain: '
        READ *, Ly

        PRINT *,'Enter vD :'
        READ *,vD

        IF ((N .NE. N_old) .OR. (M .NE. M_old)) THEN
          IF (N_old .NE. -1) THEN
            CALL FSR_deselect_display
          ENDIF
          N_old = N
          M_old = M
          CALL FSR_select_display_menu(8,MAX(N,256),M+180)
C         CALL FSR_set_display_color_map('rainbow')
          pi2 = 8*ATAN(1.d0)
          trin(1:30) = [0:29]
          trin(31:70) = [30:108:2]
          trin(71:256) = [109:294]
          trin = trin / 294.d0
```