Filipe da Silva Oliveira

Numerical Package for Large-Scale Transport Coefficients in Two-Dimensional Incompressible Flow

U. PORTO

FC FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Department of Mathematics

Faculty of Sciences of University of Porto

June 2015

Filipe da Silva Oliveira

Numerical Package for Large-Scale Transport Coefficients in Two-Dimensional Incompressible Flow

A thesis submitted to the Faculty of Sciences, University of Porto in partial fulfillment of the requirements for the degree of Master in Mathematical Engineering, under the supervision of Prof. Dr. Sílvio Marques de Almeida Gama and Prof. Dr. Maria João Pinto Sampaio Rodrigues.

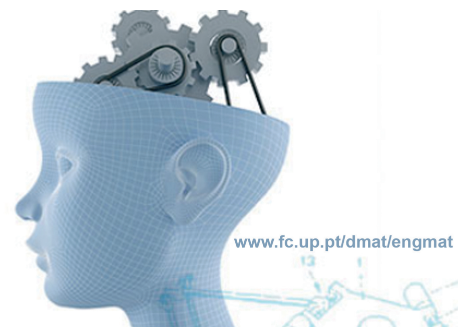
Department of Mathematics
Faculty of Sciences of University of Porto

June 2015

I would like to dedicate this work to my parents and my brother.

Acknowledgements

The culmination of this work necessitated the presence of some very significant people, and therefore it is of utmost importance to dedicate this space for their acknowledgement. First and foremost, I would like to thank my parents for the opportunity that they provided me with, and for their unconditional support throughout my studies. Of equal importance was the constant support and guidance that I received from my coordinators, Prof. Dr. Sílvio Marques de Gama and Prof. Dr. Maria João Pinto Sampaio Rodrigues, to whom I am really thankful. Last but not least, I would like to thank my lovely and admirable girlfriend, friends and brother for the inspiration and motivation they gave me throughout these two years.



Thesis completed under the Master in Mathematical Engineering
Department of Maths
Faculty of Sciences of the University of Porto
http://www.fc.up.pt/dmat/engmat

Abstract

Computational fluid dynamics, usually known as CFD, is a branch of fluid mechanics that uses numerical analysis and algorithms to solve and analyze problems that involve fluid flows. The main objective of this thesis is to develop a pseudo-spectral computational tool to compute eddy viscosities and large-scale coefficients related to the two-dimensional incompressible Navier-Stokes equations, given a molecular viscosity and a basic flow which must be parity-invariant and a sixfold rotational symmetric. For this purpose, different numerical techniques are going to be taken into consideration in order to solve each set of equations associated with the eddy viscosity and the large-scale transport coefficients. Another feature within this package allows one to compute and follow in real time the time evolution of the vorticity of a flow, given a random periodic flow as initial condition.

Contents

1 Introduction **1**
1.1 Introduction . . . 1
1.2 Notations . . . 1
1.3 Fourier Series in 2 dimensions . . . 2

2 My_fft2d and my_derop modules **5**
2.1 My_fft2d module . . . 5
2.1.1 my_fft2d_setup_r2c - subroutine that setups the FFT environment . . . 5
2.1.2 my_fft2d_close_r2c - subroutine that deallocates all memory associated to the FFT environment . . . 6
2.1.3 Discrete forward/backward Fourier transform . . . 6
2.1.4 Verification and validation of my_fft2d_dir and my_fft2d_inv subroutines . . . 9
2.2 My_derop module . . . 10
2.2.1 Differential operators in Fourier domain . . . 11
2.2.2 my_derop_setup - subroutine that setups the differentiation environment . . . 11
2.2.3 der_in_x_hat - subroutine that computes the partial derivative with respect to x . . 13
2.2.4 der_in_y_hat - subroutine that computes the partial derivative with respect to y . . 13
2.2.5 lap_hat - subroutine that computes the Laplacian . . . 14
2.2.6 bilap_hat - subroutine that computes two times the Laplacian . . . 14
2.2.7 J_hat - subroutine that computes the Jacobian between two functions $f(x, y)$ and $g(x, y)$ 14
2.2.8 Verification and validation of my_derop.f95 subroutines . . . 16

3 Numerical computation of the eddy viscosity and the large-scale transport coefficients **19**
3.1 Computation of the eddy viscosity coefficient . . . 20
3.1.1 Simple iterative method - algorithm . . . 20
3.1.2 Simple iterative method - modified algorithm . . . 22
3.1.3 Second-order Runge-Kutta method - algorithm . . . 25
3.1.4 Slaved leap-frog method - algorithm . . . 27
3.1.5 Slaved leap-frog method - modified algorithm . . . 29
3.2 Computation of the large-scale transport coefficient a . . . 30
3.3 Computation of the large-scale transport coefficient c . . . 31
3.4 Results . . . 32
3.4.1 Simple iterative method . . . 32
3.4.2 Second-order Runge-Kutta method . . . 33
3.4.3 Slaved leap-frog method . . . 34
3.4.4 Comparison between simple iterative method and the slaved leap-frog method . . . 34

4 Numerical computation of the time evolution of the vorticity of a flow **37**
4.1 Computation of the time evolution of the vorticity of a flow . . . 37
4.2 Results . . . 40

5 Conclusion and future work **43**

A Appendix - Discrete Orthogonality Relation **47**

B Appendix - FFT **49**
B.1 Subroutine my_fft2d_setup . . . 49
B.2 Subroutine my_fft2d_close . . . 49
B.3 Subroutine my_fft2d_dir . . . 49
B.4 Subroutine my_fft2d_inv . . . 50

C Appendix - Differential Operators in Fourier domain **51**
C.1 Subroutine my_derop_setup . . . 51
C.2 Subroutine der_in_x_hat . . . 51
C.3 Subroutine der_in_y_hat . . . 52
C.4 Subroutine lap_hat . . . 52
C.5 Subroutine bilap_hat . . . 52
C.6 Subroutine J_hat . . . 53

D Appendix - Numerical methods **55**
D.1 Subroutine iterative_method_hat . . . 55
D.2 Subroutine rk2_hat . . . 56
D.3 Subroutine slaved_leap_frog . . . 57

List of Tables

1.1 Notation and its meaning. 2

2.1 Fourier coefficients of u (Equation 2.1.3), i.e., $\hat{u}_{pq}$. 9
2.2 Fourier coefficients related to Equation (2.1.3) from numerical computation. 10
2.3 Magnitude of the errors obtained from the numerical experiences based on Equation (2.1.3) where the original values in u_{old} are compared to the values in u, which were calculated by applying to u_{old} the forward followed by a backward FFT, i.e., $u = \mathrm{FFT}^{-1}(\mathrm{FFT}(u_{old}))$. . . 10
2.4 Magnitude of the errors obtained from numerical experiences based on expression (2.2.5) where the numerical solution u is compared to the exact solution u_e, which was calculated using the expression (2.2.6). 17

3.1 Correct transport coefficients values related to different ν and associated with the hexagonal decorated basic flow. 32
3.2 Transport coefficients computed using the simple iterative method with $\nu = 1$ and $\mathrm{TOL} = 10^{-5}$. 32
3.3 Maximum absolute difference between $\hat{Q}_{pq}^{n+1}$ and $\hat{Q}_{pq}^{n}$ evolution while computing Q_1 using the simple iterative method with $N_x = N_y = 64$ and $\mathrm{TOL} = 10^{-5}$. 33
3.4 Maximum absolute difference between $\hat{Q}_{pq}^{n+1}$ and $\hat{Q}_{pq}^{n}$ evolution while computing Q_1 using the RK2 method with $N_x = N_y = 64$ and $\mathrm{TOL} = 10^{-5}$. 34
3.5 Transport coefficients computed using the slaved leap-frog method with different resolutions and $\mathrm{TOL} = 10^{-11}$. 34
3.6 Numerical solutions of the transport coefficients for $\nu = 1$ and $\mathrm{TOL} = 10^{-5}$. 35

List of Figures

2.1.1 Hermitian symmetry illustration. The red dots are complex conjugate of the blue dots. . . . 7
2.1.2 Illustration of the data layout in the matrix uhat for a two-dimensional $N_x = 16$ by $N_y = 16$ real-to-complex Fourier transform. 8
2.2.1 Differentiation arrays and matrices. 12
2.2.2 Computation of $\mathcal{J}(f, g)$ flowchart. 15

3.1.1 Simple iterative method flowchart. 21
3.1.2 Flowchart to compute the r.h.s. of Equation 3.0.3. 22
3.1.3 Dealising illustration assuming $N_x = N_y = 16$ where the blue dots represent all wavenumbers that will be contaminated by aliases whenever a non-linear computation is performed. 23
3.1.4 Modified simple iterative method flowchart. 24
3.1.5 RK2 method flowchart. 26
3.1.6 Slaved leap-frog method flowchart. 28
3.1.7 Modified slaved leap-frog method flowchart. 29
3.2.1 Flowchart to compute the r.h.s. of Equation (3.2.3). 30
3.3.1 Flowchart to compute the r.h.s. of Equation (3.3.1). 31
3.4.1 Hexagonal decorated flow contour plot. 32
3.4.2 Simple iterative method and slaved leap-frog method execution time assuming $\nu = 1$, and the hexagonal decorated basic flow. 35

4.1.1 Function $f(k)$ plot with its roots where $k_c = \sqrt{\mu_4/\mu_6}$. 38
4.1.2 Modified slaved leap-frog method flowchart for the computation of the time evolution of the vorticity of a flow. 39
4.1.3 Algorithm flowchart to compute the NLT. 40
4.2.1 Plot of the evolution over time of $\partial^2\psi$ for $N_x = N_y = 256$, $a = 1$, $c = 0$, $\mu_4 = -0.05$ and $\mu_6 = 0$ with a random periodic initial condition ψ^0. 41

Chapter 1

Introduction

This chapter will provide a brief introduction about the topic covered throughout the chapters that follow. Additionally, it provides guidelines on how the thesis is organized and the notation that is used.

1.1 Introduction

In computational fluid dynamics the motion of a viscous fluid can be described using different models. The traditional one is known as the Navier-Stokes equations which is based on a set of partial differential equations. These equations were derived independently by Claude-Louis Navier in 1822 and George Gabriel Stokes in 1845. However, they are too difficult to solve analytically and so, computational resources are used nowadays to solve them through a variety of techniques like finite difference, finite volume, finite element and spectral methods.

The work presented in this thesis explores some of the techniques mentioned above, and makes usage of the Navier-Stokes equations in order to develop a software package in Fortran 95. This computational tool computes eddy viscosities [10] and large-scale coefficients related to the two-dimensional incompressible Navier-Stokes equations, given a molecular viscosity and a basic flow that possesses a parity-invariant and a sixfold rotation symmetry. Another feature within this package allows one to compute and follow in real time the time evolution of the vorticity of a flow, given a random periodic flow as initial condition.

This thesis is organized in the following way: In this chapter, some important notions related to the Fourier Series are revised. Afterwards, Chapter 2 presents a set of subroutines which were developed and are the foundation of all work presented in Chapter 3 and Chapter 4. The following two chapters, Chapter 3 and Chapter 4, present the strategies that were taken into consideration in order to compute the large-scale transport coefficients and the time evolution of the vorticity in a two-dimensional incompressible flow, respectively. At the end, a conclusion about the numerical methods that were tested and also some notes about future work are made in Chapter 5.

1.2 Notations

Table 1.1 shows the notation used throughout this document and its meaning.

Notation	Meaning
∂_x	∂/∂_x
∂_y	∂/∂_y
∂^2	$\partial_x\partial_x + \partial_y\partial_y$ (Laplacian Operator)
$\cdot$	Dot product operator
$\mathcal{J}(\cdot,\cdot)$	Jacobian
$\tilde{\mathcal{A}}$	Linearized Navier-Stokes operator
Ψ	Basic flow
ν_{eddy}	Eddy viscosity coefficient
ν	Molecular viscosity
$\imath$	Imaginary unity, $\sqrt{-1}$
$\tilde{p}$	$2\pi p/L_x$
$\tilde{q}$	$2\pi q/L_y$
k^2	$\tilde{p}^2 + \tilde{q}^2$
N_x	Number of collocation points with respect to the x-axis
N_y	Number of collocation points with respect to the y-axis
L_x	Period of a periodic function with respect to the x-axis
L_y	Period of a periodic function with respect to the y-axis

Table 1.1: Notation and its meaning.

It was also assumed in this work as a convention that all variables in the subroutines that end their name with _hat correspond to variables which are in the Fourier domain. In the same way, all subroutines that end with _hat are subroutines which return their result as Fourier coefficients.

1.3 Fourier Series in 2 dimensions

A brief explanation about the Fourier series will be given now. The main intention is to revise briefly some important notions and thus, for a more detailed analysis and explanation see [6]. The Fourier series of a function $u(x,y)$ with period L_x in the independent variable x, and period L_y in the independent variable y is given by

$$u(x,y) = \sum_{p=-\infty}^{\infty} \sum_{q=-\infty}^{\infty} \hat{u}^e_{pq} e^{\imath\left(\frac{2\pi}{L_x}p, \frac{2\pi}{L_y}q\right)\cdot(x,y)}, \tag{1.3.1}$$

where $p \in \mathbb{Z}$, $q \in \mathbb{Z}$, $\imath = \sqrt{-1}$, symbol $\cdot$ represents the dot product operator and

$$\hat{u}^e_{pq} = \frac{1}{L_x L_y} \int_0^{L_x} \int_0^{L_y} u(x,y) e^{-\imath\left(\frac{2\pi}{L_x}p, \frac{2\pi}{L_y}q\right)\cdot(x,y)} dydx. \tag{1.3.2}$$

Note that $\hat{u}^e_{pq}$ is the (p,q) exact Fourier coefficient and Equation (1.3.1) is the backward Fourier transform whereas Equation (1.3.2) corresponds to the forward Fourier transform. Since the Fourier transforms will be used in a computational context, it makes more sense to talk about the discrete truncated Fourier series. Therefore, let $u_{k_x k_y}(x,y)$ be the truncated Fourier series defined as:

$$u_{k_x k_y}(x,y) = \sum_{p=-k_x}^{k_x} \sum_{q=-k_y}^{k_y} \hat{u}^e_{pq} e^{\imath\left(\frac{2\pi}{L_x}p, \frac{2\pi}{L_y}q\right)\cdot(x,y)},$$

with k_x, $k_y \in \mathbb{Z}$. Furthermore, the discrete truncated form is given by

$$u(x_i,y_j) = \sum_{p=-k_x}^{k_x} \sum_{q=-k_y}^{k_y} \hat{u}^c_{pq}\, e^{\imath(\frac{2\pi}{L_x}p, \frac{2\pi}{L_y}q)\cdot(x_i,y_j)},$$

with

$$\hat{u}^c_{pq} = \frac{1}{N_x\,N_y}\sum_{i=1}^{N_x}\sum_{j=1}^{N_y} u(x_i, y_j)\, e^{-\imath(\frac{2\pi}{L_x}p, \frac{2\pi}{L_y}q)\cdot(x_i, y_j)}, \tag{1.3.3}$$

$$x_i = \frac{L_x}{N_x} i \qquad (i = 1, 2, \cdots, N_x)\,,$$

and

$$y_j = \frac{L_y}{N_y} j \qquad (j = 1, 2, \cdots, N_y)\,,$$

where $N_x = 2k_x + 1$ and $N_y = 2k_y + 1$. Here, $\hat{u}^c_{pq}$ represents the (p, q) collocated Fourier coefficient and it is possible to show its relationship with $\hat{u}^e_{pq}$. To do so, recall Equation (1.3.3),

$$\hat{u}^c_{pq} = \frac{1}{N_x\,N_y}\sum_{i=1}^{N_x}\sum_{j=1}^{N_y}\left(\sum_{\alpha=-\infty}^{\infty}\sum_{\beta=-\infty}^{\infty}\hat{u}^e_{\alpha\beta} e^{\imath\left(\frac{2\pi}{L_x}\alpha, \frac{2\pi}{L_y}\beta\right)\cdot(x_i, y_i)}\right) e^{-\imath(\frac{2\pi}{L_x}p, \frac{2\pi}{L_y}q)\cdot(x_i, y_j)}. \tag{1.3.4}$$

By keeping in mind that

$$x_i = \frac{L_x}{N_x} i \;\; (i = 1, 2, \cdots, N_x) \text{ and } y_j = \frac{L_y}{N_y} j \;\; (j = 1, 2, \cdots, N_y)\,,$$

then it is possible to rewrite Equation (1.3.4) as follows:

$$\hat{u}^c_{pq} = \frac{1}{N_x\,N_y}\sum_{\alpha=-\infty}^{\infty}\sum_{\beta=-\infty}^{\infty}\hat{u}^e_{\alpha\beta}\sum_{i=1}^{N_x}\sum_{j=1}^{N_y} e^{\imath\left((\alpha-p)\frac{2\pi}{N_x}i + (\beta-q)\frac{2\pi}{N_y}j\right)}. \tag{1.3.5}$$

The discrete orthogonality relation (see Appendix A) shows that in Equation (1.3.5) it is only required to take into account the terms which

$$\alpha - p = m\,N_x \text{ and } \beta - q = n\,N_y$$

with $m = 0, \pm1, \pm2, \ldots,\; n = 0, \pm1, \pm2, \ldots,\; -k_x \le p \le k_x,\; -k_y \le q \le k_y,\; \alpha \in \mathbb{Z}$ and $\beta \in \mathbb{Z}$. Now, by separating in Equation (1.3.5) all terms with $m = 0$ or $n = 0$ from all others, it is possible to write that

$$\hat{u}^c_{pq} = \hat{u}^e_{pq} + \sum_{m\in Z^*}\sum_{n\in Z^*}\hat{u}^e_{(p+m\,N_x)(q+n\,N_y)}.$$

The second term is the so called alias and they appear whenever $u(x, y)$ in the physical domain contains higher frequencies than the maximum frequency present in the discrete truncated Fourier transform. In other words, N_x and N_y are smaller than they should be and they must be greater in order to overcome the alias problem. The notion of aliases will be essential in Chapter 3 and Chapter 4.

Chapter 2

My_fft2d and my_derop modules

This chapter presents some of the main subroutines which were developed in this work. These subroutines are divided in two different modules called my_fft2d and my_derop. The former is responsible for computing the forward and backward Fourier transform based on the package FFTW - Fast Fourier Transform in the West. The latter is composed by subroutines which compute derivatives in the Fourier domain. These two modules will be crucial for the work that was developed afterwards and which is presented in Chapter 3 and Chapter 4.

2.1 My_fft2d module

The subroutines which compose the my_fft2d.f95 module will be shown in this section and the way of using them will also be explained by means of examples (see Appendix B for the implementation code). The objective of this module is to offer a more user friendly interface than the one given by the FFTW. Additionally, the Fourier transforms are presented once again but with small changes in the expressions that were made due to implementation conveniences.

For starters, the module mentioned before is based on the FFTW 3.3.4 library or earlier versions. Thus, any program that uses this module must be linked with this library. On Unix systems, you need to link with -lfftw3 -lm. FFTW library, which is a collection of C routines for computing discrete Fourier transforms, can be downloaded from [1] and it can be used in C or Fortran programs. Furthermore, FFTW employs dynamic algorithms which adapt to the hardware in use in order to maximize performance. The algorithms involved are $O(n\,log\,n)$ and they get maximum efficiency for arrays whose size can be factored into small primes (2, 3, 5 and 7). For readers interested in FFTW, see [2] for more details.

2.1.1 my_fft2d_setup_r2c - subroutine that setups the FFT environment

In order to use the real-to-complex or complex-to-real subroutines, i.e., a forward or a backward Fourier transform subroutines, a fast Fourier transform (FFT) environment must be created and this is achieved by calling the subroutine my_fft2d_setup_r2c at the beginning of the program. The arguments that must be provided are a real matrix u of dimensions $N_x \times N_y$ and a complex matrix uhat of dimensions $(N_x/2+1) \times N_y$. It is advisable to only initialize these matrices after calling my_fft2_setup_r2c subroutine since u and uhat might be changed by this subroutine. Another important remark is the fact that both forward and backward FFT subroutines take advantage of the Hermitian symmetry and thus, uhat only needs roughly half of the memory that the matrix u needs to use. At the end of the execution of the subroutine, two different plans are returned, the fft2d_dir_plan and fft2d_inv_plan, which contain all the information needed to compute a forward and backward FFT, respectively. The interface of my_fft2d_setup_r2c subroutine is:

```
subroutine my_fft2d_setup_r2c (fft2d_dir_plan,fft2d_inv_plan, u, uhat, &
        Nx, Ny)
        integer,     intent(in)  :: Nx, Ny
        complex(8),  dimension(0:Nx/2, 0:Ny-1), intent(in) :: uhat
        real(8),     dimension(0:Nx-1, 0:Ny-1), intent(in) :: u
        integer(8),  intent(out) :: fft2d_dir_plan, fft2d_inv_plan
```

```
return end subroutine my_fft2d_setup_r2c
```

Arguments:

fft2d_dir_plan	:	An object that contains the data that FFTW needs to compute the forward FFT;
fft2d_inv_plan	:	An object that contains the data that FFTW needs to compute the backward FFT;
u	:	A real matrix;
uhat	:	A complex matrix;
Nx	:	Number of rows present in matrix u;
Ny	:	Number of columns present in matrix u.

2.1.2 my_fft2d_close_r2c - subroutine that deallocates all memory associated to the FFT environment

It is a good practice to free all resources which are not being used anymore. Consequently, all my_fft2d_setup_r2c calls should be paired with an invocation of my_fft2_close_r2c to prevent memory leaks. It is imperative that no use of any forward or backward FFT subroutines is made without setting up the environment again, otherwise the program might crash. An example of good usage is given bellow.

```
program simple_example
    implicit none
    integer, parameter :: Nx = 64 !Nx: power of 2 to maximize performance
    integer, parameter :: Ny = Nx !Ny: power of 2 to maximize performance
    integer(8) :: fft2d_dir_plan, fft2d_inv_plan
    real(8),     dimension(0:Nx-1, 0:Ny-1) :: u
    complex(8), dimension(0:Nx/2, 0:Ny-1) :: u_hat

    call my_fft2d_setup_r2c (fft2d_dir_plan,fft2d_inv_plan, u, u_hat,&
        Nx, Ny)
    !<All FFT subroutines calls goes here>
    call my_fft2d_close_r2c (fft2d_dir_plan,fft2d_inv_plan)

    stop 'end of main'
end program simple_example
```

The interface of my_fft2_close_r2c subroutine is:

```
subroutine my_fft2d_close_r2c (fft2d_dir_plan,fft2d_inv_plan)
    integer(8), intent(inout) :: fft2d_dir_plan, fft2d_inv_plan
return end subroutine my_fft2d_close_r2c
```

Arguments:

fft2d_dir_plan	:	An object that contains the data that FFTW needs to compute the forward FFT;
fft2d_inv_plan	:	An object that contains the data that FFTW needs to compute the backward FFT.

2.1.3 Discrete forward/backward Fourier transform

Let $u(x, y)$ be a function from $\mathbb{R}^2$ to $\mathbb{R}$. Assume that u is periodic in each independent variable, i.e.

$$u\left(x + L_x, y + L_y\right) = u(x, y),$$

for given constants $L_x > 0$ and $L_y > 0$, called the periods in x and y, respectively. Define

$$u_{ij} = u\left(x_i, y_j\right),$$

with

$$x_i = \frac{L_x}{N_x} i \;\; \left(i = 0, 1, 2, \cdots, N_x - 1\right),$$

and

$$y_j = \frac{L_y}{N_y} j \;\; (j = 0, 1, 2, \cdots, N_y - 1)\,,$$

where N_x and N_y are the number of collocation points in x and y direction, respectively. The forward discrete Fourier transform is

$$\hat{u}_{pq} = \frac{1}{N_x\, N_y} \sum_{i=0}^{N_x-1} \sum_{j=0}^{N_y-1} u_{ij}\, e^{-\imath(\tilde{p},\tilde{q})\cdot(x_i,y_j)} \tag{2.1.1}$$

and the backward discrete Fourier transform is

$$u_{ij} = \sum_{p=-\frac{N_x}{2}+1}^{N_x/2} \; \sum_{q=-\frac{N_y}{2}+1}^{N_y/2} \hat{u}_{pq}\, e^{\imath(\tilde{p},\tilde{q})\cdot(x_i,y_j)} \tag{2.1.2}$$

where $\tilde{p} = \frac{2\pi}{L_x} p$ and $\tilde{q} = \frac{2\pi}{L_y} q$.

Note that in Equation (2.1.1) i starts in zero instead of one and it goes to $N_x - 1$ rather than N_x, as it was seen in Chapter 1. There are also a few changes with respect to the indices in Equation (2.1.2). The only reason behind this is due to implementation convenience as it will be seen in this chapter. These equations are in fact the ones used in the my_fft2d_dir and my_fft2d_inv, respectively.

my_fft2d_dir - subroutine that computes the forward Fourier transforms

After setting up the FFT environment it is possible to invocate my_fft2d_dir to perform a forward Fourier transform. Given a discretized function u with $N_x \times N_y$ collocation points, my_fft2d_dir subroutine returns in uhat the Fourier coefficients associated with u. In a formal way, this routine computes Equation (2.1.1). Since $u \in \mathbb{R}^{N_x \times N_y}$ only roughly half of the Fourier coefficients will be computed, more precisely the ones which belong to the first and fourth quadrant (see Figure 2.1.1). The remaining ones are just the complex conjugates due to the Hermitian symmetry, i.e. $\hat{u}_{-p,-q} = \hat{u}^*_{p,q}$, and do not need to be computed or stored in memory (see [2]).

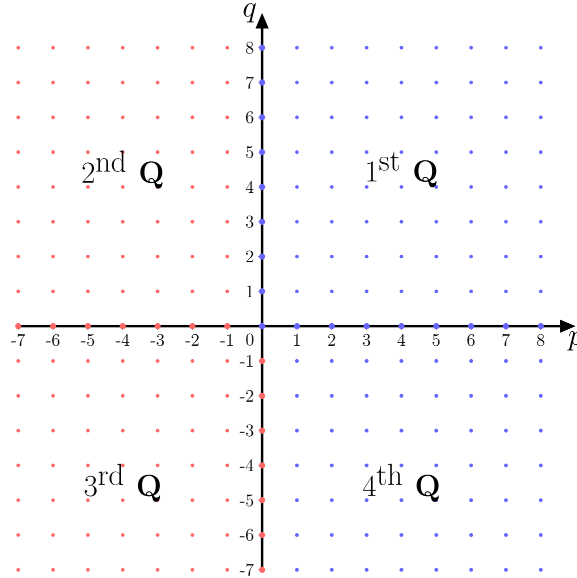


Figure 2.1.1: Hermitian symmetry illustration. The red dots are complex conjugate of the blue dots.

These coefficients are organized in such a way that the wave numbers (p, q) where p goes from zero to $N_x/2$ and q goes from zero to $N_y/2$ correspond to the index (p, q) in memory, i. e., $\hat{u}_{pq}$ =uhat[p,q] for $p = 0, 1, 2, \cdots, N_x/2$ and $q = 0, 1, 2, \cdots, N_y/2$. For the remaining wave numbers, the memory map is $(p, N_y + q)$.

In other words, $\hat{u}_{pq}$ =uhat[p,$N_y + q$] for $p = 0, 1, 2, \cdots, N_x/2$, and $q = -N_y/2 + 1, -N_y/2 + 2, \cdots, -1$ (see Figure 2.1.2).

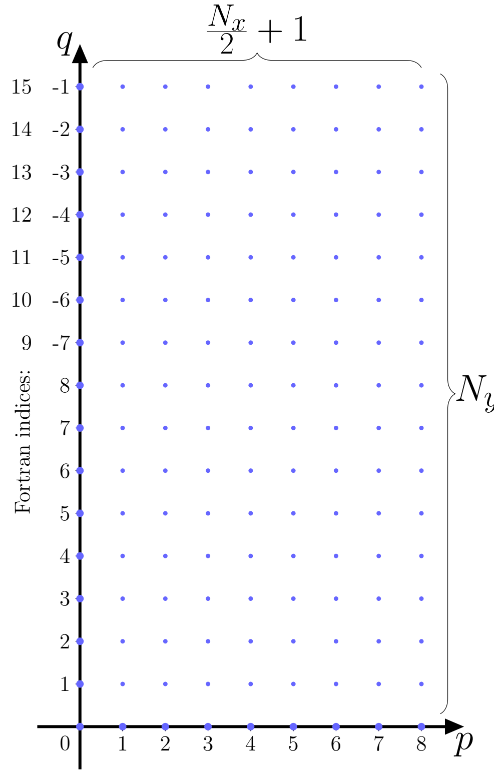


Figure 2.1.2: Illustration of the data layout in the matrix uhat for a two-dimensional $N_x = 16$ by $N_y = 16$ real-to-complex Fourier transform.

The interface of this subroutine is:

```
subroutine my_fft2d_dir (uhat, u, plan, Nx, Ny)
          integer,     intent(in) :: Nx, Ny
          integer(8),  intent(in) :: plan
          real(8),     dimension(0:Nx-1, 0:Ny-1), intent(in)  :: u
          complex(8),  dimension(0:Nx/2, 0:Ny-1), intent(out) :: uhat
end subroutine my_fft2d_dir
```

Arguments:

uhat	:	Fourier coefficients of u;
u	:	Matrix with the collocation points of the discretized function u;
plan	:	An object that contains the data that FFTW needs to compute the forward FFT;
Nx	:	Number of rows present in matrix u;
Ny	:	Number of columns present in matrix u.

my_fft2d_inv - subroutine that computes the backward Fourier transform

The backward Fourier transform is achieved through the subroutine my_fft2d_inv. In other words, to recover u from the Fourier domain and set it back to the physical domain a backward FFT is needed. In a formal way, this subroutine computes Equation (2.1.2) and its result is returned in the variable u. The interface associated with this subroutine is the following:

```
subroutine my_fft2d_inv (u, uhat, plan, Nx, Ny)
```

```fortran
        integer,      intent(in)  :: Nx, Ny
        integer(8),   intent(in)  :: plan
        complex(8),   dimension(0:Nx/2, 0:Ny-1), intent(in)   :: uhat
        real(8),      dimension(0:Nx-1, 0:Ny-1), intent(out)  :: u
end subroutine my_fft2d_inv
```

Arguments:

u	:	Discretized function u in the physical domain;
uhat	:	Fourier coefficients of u;
plan	:	An object that contains the data that FFTW needs to compute the backward FFT;
Nx	:	Number of rows present in matrix u;
Ny	:	Number of columns present in matrix u.

2.1.4 Verification and validation of my_fft2d_dir and my_fft2d_inv subroutines

An easy way to verify if the implementation of my_fft2d_dir and my_fft2d_inv subroutines are correct is to compute a forward followed by a backward Fourier transform and see if the original values are recovered. Assume $N_x = N_y = 8$ and let

$$u(x, y) = \sum_{p=0}^{p_{max}} \sum_{q=-q_{max}}^{q_{max}} 2 \times a(p, q) \times \cos(px + qy) + 2 \times b(p, q) \times \sin(px + qy) \qquad (2.1.3)$$

where

$$\begin{aligned} p_{max} &= N_x/2 - 1, \\ q_{max} &= N_y/2 - 1, \\ a(p, q) &= (p + q + 4), \\ b(p, q) &= (p + q + 4)^2. \end{aligned}$$

Thereby,

$$\begin{aligned} u(x, y) = \; & 8 + 10\cos(x) + 50\sin(x) + 12\cos(2x) + 72\sin(2x) + 14\cos(3x) + 98\sin(3x) \\ & + 16\cos(y) + 32\sin(y) + 12\cos(x + y) + 72\sin(x + y) + 14\cos(2x + y) + 98\sin(2x + y) \\ & + 16\cos(3x + y) + 128\sin(3x + y) + 16\cos(2y) + 64\sin(2y) \\ & + 14\cos(x + 2y) + 98\sin(x + 2y) + 16\cos(2x + 2y) + 128\sin(2x + 2y) \\ & + 18\cos(3x + 2y) + 162\sin(3x + 2y) + 16\cos(3y) + 96\sin(3y) \\ & + 16\cos(x + 3y) + 128\sin(x + 3y) + 18\cos(2x + 3y) + 162\sin(2x + 3y) \\ & + 20\cos(3x + 3y) + 200\sin(3x + 3y) + 4\cos(x - 3y) + 8\sin(x - 3y) \\ & + 6\cos(2x - 3y) + 18\sin(2x - 3y) + 8\cos(3x - 3y) + 32\sin(3x - 3y) \\ & + 6\cos(x - 2y) + 18\sin(x - 2y) + 8\cos(2x - 2y) + 32\sin(2x - 2y) \\ & + 10\cos(3x - 2y) + 50\sin(3x - 2y) + 8\cos(x - y) + 32\sin(x - y) \\ & + 10\cos(2x - y) + 50\sin(2x - y) + 12\cos(3x - y) + 72\sin(3x - y) \end{aligned} ,$$

and the corresponding Fourier coefficients are presented in Table 2.1. Note that all wave numbers that are not shown in the mentioned table are assumed to be equal to zero.

p/q	0	1	2	3	4	-3	-2	-1
0	8	$8-16\imath$	$8-32\imath$	$8-48\imath$	0	$8+48\imath$	$8+32\imath$	$8+16\imath$
1	$5-25\imath$	$6-36\imath$	$7-49\imath$	$8-64\imath$	0	$2-4\imath$	$3-9\imath$	$4-16\imath$
2	$6-36\imath$	$7-49\imath$	$8-64\imath$	$9-81\imath$	0	$3-9\imath$	$4-16\imath$	$5-25\imath$
3	$7-49\imath$	$8-64\imath$	$9-81\imath$	$10-100\imath$	0	$4-16\imath$	$5-25\imath$	$6-36\imath$

Table 2.1: Fourier coefficients of u (Equation 2.1.3), i.e., $\hat{u}_{pq}$.

In Table 2.2, the numerical results obtained by calling my_fft2d_dir subroutine are presented. Note that these results were rounded to two decimal places due to the limited space within the table. By comparing Tables 2.2 to Table 2.1, it is possible to conclude that the results given numerically are correct and they only have an error due to the precision of the machine.

p/q	0	1	2
0	$8.00 + 0.00\imath$	$8.00 - 16.00\imath$	$8.00 - 32.00\imath$
1	$5.00 - 25.00\imath$	$6.00 - 36.00\imath$	$7.00 - 49.00\imath$
2	$6.00 - 36.00\imath$	$7.00 - 49.00\imath$	$8.00 - 64.00\imath$
3	$7.00 - 49.00\imath$	$8.00 - 64.00\imath$	$9.00 - 81.00\imath$
4	$0.00 + 0.00\imath$	$2.09 \times 10^{-14} + 1.39 \times 10^{-14}\imath$	$2.31 \times 10^{-14} + 6.44 \times 10^{-15}\imath$

(a)

p/q	3	4	-3
0	$8.00 - 48.00\imath$	$2.31 \times 10^{-14} + 0.00\imath$	$8.00 + 48.00\imath$
1	$8.00 - 64.00\imath$	$2.04 \times 10^{-14} + 7.11 \times 10^{-15}\imath$	$2.00 - 4.00\imath$
2	$9.00 - 81.00\imath$	$2.40 \times 10^{-14} + 3.55 \times 10^{-15}\imath$	$3.00 - 9.00\imath$
3	$10.00 - 100.00\imath$	$1.51 \times 10^{-14} - 7.11 \times 10^{-15}\imath$	$4.00 - 16.00\imath$
4	$1.55 \times 10^{-14} - 2.94 \times 10^{-15}\imath$	$0.00 + 0.00\imath$	$1.55 \times 10^{-14} + 2.94 \times 10^{-15}\imath$

(b)

p/q	-2	-1
0	$8.00 + 32.00\imath$	$8.00 + 16.00\imath$
1	$3.00 - 9.00\imath$	$4.00 - 16.00\imath$
2	$4.00 - 16.00\imath$	$5.00 - 25.00\imath$
3	$5.00 - 25.00\imath$	$6.00 - 36.00\imath$
4	$2.31 \times 10^{-14} - 6.44 \times 10^{-15}\imath$	$2.09 \times 10^{-14} - 1.39 \times 10^{-14}\imath$

(c)

Table 2.2: Fourier coefficients related to Equation (2.1.3) from numerical computation.

Furthermore, let u_{old} be a matrix with the collocation points of Equation (2.1.3) and u a matrix with the result after applying to u_{old} the forward followed by a backward Fourier transform, i.e., $u = \text{FFT}^{-1}\left(\text{FFT}\left(u_{old}\right)\right)$. The maximum absolute error between u and u_{old}, i. e. $||u - u_{old}||_{max}$, is in the order of 10^{-13} which is a good indicator that the subroutines are working properly. Several identical tests were made based on Equation (2.1.3) but varying the problem size. As it can be seen from Table 2.3, the magnitude of the errors are around 10^{-8}.

N_x	N_y	$O\left(\vert\vert u - u_{old}\vert\vert_{max}\right)$
64	128	10^{-9}
128	64	10^{-10}
128	256	10^{-8}
128	512	10^{-7}

Table 2.3: Magnitude of the errors obtained from the numerical experiences based on Equation (2.1.3) where the original values in u_{old} are compared to the values in u, which were calculated by applying to u_{old} the forward followed by a backward FFT, i.e., $u = \text{FFT}^{-1}\left(\text{FFT}\left(u_{old}\right)\right)$.

2.2 My_derop module

In this section, the subroutines which compose my_derop.f95 module will be shown and an explanation will be provided as to how to use them by means of examples. Additionally, some notions about differentiation in the Fourier domain are presented. All subroutines implementation code are presented in Appendix C.

2.2.1 Differential operators in Fourier domain

In order to understand how differentiation in the Fourier domain works, it is important to recall the backward Fourier transform given by Equation (2.1.2). To compute $\partial_x u$, it follows

$$\begin{aligned}\partial_x u_{ij} &= \partial_x \left(\sum_{p=-\frac{N_x}{2}+1}^{N_x/2} \sum_{q=-\frac{N_y}{2}+1}^{N_y/2} \hat{u}_{pq}\, e^{\imath(\tilde{p},\tilde{q})\cdot(x_i,y_j)} \right) \\ &= \sum_{p=-\frac{N_x}{2}+1}^{N_x/2} \sum_{q=-\frac{N_y}{2}+1}^{N_y/2} \imath\, \tilde{p}\, \hat{u}_{pq}\, e^{\imath(\tilde{p},\tilde{q})\cdot(x_i,y_j)}. \end{aligned} \quad (2.2.1)$$

It is also possible to deduce the formulas for $\partial_y u$, $\partial^2 u$ and $\partial^2\partial^2 u$ by using the same reasoning. Thus,

$$\partial_y u_{ij} = \sum_{p=-\frac{N_x}{2}+1}^{N_x/2} \sum_{q=-\frac{N_y}{2}+1}^{N_y/2} \imath\, \tilde{q}\, \hat{u}_{pq}\, e^{\imath(\tilde{p},\tilde{q})\cdot(x_i,y_j)}, \quad (2.2.2)$$

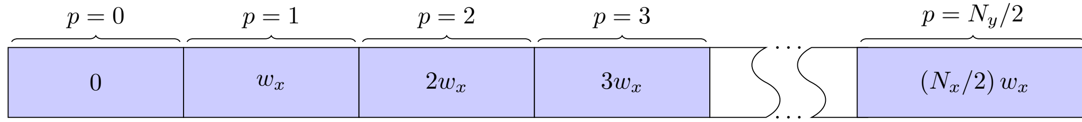
$$\begin{aligned}\partial^2 u_{ij} &= \partial_x\partial_x u + \partial_y\partial_y u \\ &= \partial_x \left(\sum_{p=-\frac{N_x}{2}+1}^{N_x/2} \sum_{q=-\frac{N_y}{2}+1}^{N_y/2} \imath\, \tilde{p}\, \hat{u}_{pq}\, e^{\imath(\tilde{p},\tilde{q})\cdot(x_i,y_j)} \right) + \partial_y \left(\sum_{p=-\frac{N_x}{2}+1}^{N_x/2} \sum_{q=-\frac{N_y}{2}+1}^{N_y/2} \imath\, \tilde{q}\, \hat{u}_{pq}\, e^{\imath(\tilde{p},\tilde{q})\cdot(x_i,y_j)} \right) \\ &= \sum_{p=-\frac{N_x}{2}+1}^{N_x/2} \sum_{q=-\frac{N_y}{2}+1}^{N_y/2} -\left(\tilde{p}^2 + \tilde{q}^2\right) \hat{u}_{pq} e^{\imath(\tilde{p},\tilde{q})\cdot(x_i,y_j)}, \end{aligned} \quad (2.2.3)$$

and

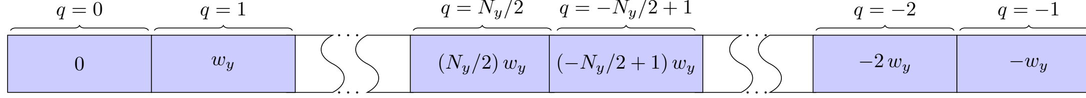
$$\begin{aligned}\partial^2\partial^2 u_{ij} &= \partial_x\partial_x\left(\partial^2 u\right) + \partial_y\partial_y\left(\partial^2 u\right) \\ &= \sum_{p=-\frac{N_x}{2}+1}^{N_x/2} \sum_{q=-\frac{N_y}{2}+1}^{N_y/2} \left(\tilde{p}^2 + \tilde{q}^2\right)^2 \hat{u}_{pq}\, e^{\imath(\tilde{p},\tilde{q})\cdot(x_i,y_j)}. \end{aligned} \quad (2.2.4)$$

2.2.2 my_derop_setup - subroutine that setups the differentiation environment

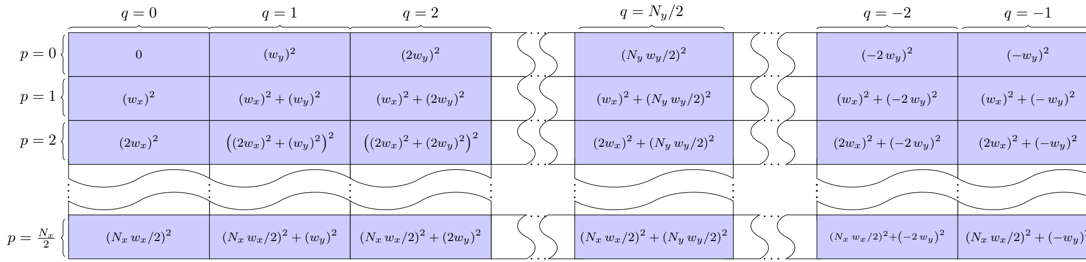
In all functions that will be presented in this section, it will be common to find the subroutine's arguments named as kx, ky, k2 and k4. These arguments are simply the differentiation arrays, in the case of kx and ky, or differentiation matrices, in the case of k2 and k4. In other words, kx corresponds to an array with the possible values of $\tilde{p}$ in expression (2.2.1) (see Figure 2.2.1a), ky to an array that contains all possible values of $\tilde{q}$ in expression (2.2.2) (see Figure 2.2.1b), k2 to a matrix with all possible values of $\left(\tilde{p}^2 + \tilde{q}^2\right)$ present in expression (2.2.3) (see Figure 2.2.1c) and k4 to a matrix that consists of all possible values of $\left(\tilde{p}^2 + \tilde{q}^2\right)^2$ in expression (2.2.4) which basically corresponds to the matrix k2 where each element is squared.



(a) Illustration on how $\tilde{p} = pw_x$ values are organized within the array kx where $w_x = 2\pi/L_x$.



(b) Illustration on how $\tilde{q} = qw_y$ values are organized within the array ky where $w_y = 2\pi/L_y$.



(c) Illustration on how $\tilde{p}^2 + \tilde{q}^2 = (pw_x)^2 + (qw_y)^2$ values are organized within the matrix k2 where $w_x = 2\pi/L_x$ and $w_y = 2\pi/L_y$.

Figure 2.2.1: Differentiation arrays and matrices.

The subroutine my_derop_setup is in charge of initializing all differentiation arrays and matrices, so before using any subroutine from this module, it is necessary to call my_derop_setup subroutine to setup the differentiation operators environment. Note that some of the subroutines require the usage of my_fft2d.f95 module so sometimes it is also necessary to setup the FFT environment beforehand. An example of a good usage of this module is given below in simple_example program.

```
program simple_example
    implicit none
    real(8), parameter :: PI = datan(1.d0) * 4.d0, TWO_PI = 2.d0 * PI
    integer, parameter :: Nx = 64 !Nx: power of 2 to maximize performance
    integer, parameter :: Ny = Nx !Ny: power of 2 to maximize performance
    real(8), parameter :: Lx = TWO_PI !function periodicity in x
    real(8), parameter :: Ly = TWO_PI !function periodicity in y
    integer, parameter :: pmax = Nx/2, qmax = ny/2

    integer(8) :: fft2d_dir_plan, fft2d_inv_plan
    real(8),     dimension(0:Nx/2, 0:Ny-1) :: k2, k4
    real(8),     dimension(0:Nx/2)          :: kx
    real(8),     dimension(0:Ny-1)          :: ky
    real(8),     dimension(0:Nx-1, 0:Ny-1) :: u
    complex(8), dimension(0:Nx/2, 0:Ny-1) :: u_hat

    call my_fft2d_setup(fft2d_dir_plan,fft2d_inv_plan, u, u_hat, Nx, Ny)
    call my_derop_setup(kx, ky, k2, k4, Lx, Ly, pmax, qmax)
    !<All my_derop subroutines calls goes here>
    call my_fft2d_close(fft2d_dir_plan,fft2d_inv_plan)
    stop 'end of main'
```

```
end program simple_example
```

The interface of the subroutine my_derop_setup is

```
subroutine my_derop_setup (kx, ky, k2, k4, Lx, Ly, pmax, qmax)
        integer, intent(in)   :: pmax, qmax
        real(8), intent(in)   :: Lx, Ly
        real(8), dimension(0:pmax,0:2*qmax-1), intent(out) :: k2, k4
        real(8), dimension(0:2*qmax-1),            intent(out) :: ky
        real(8), dimension(0:pmax),                intent(out) :: kx
end subroutine my_derop_setup
```

Arguments:

kx	:	Differentiation array to compute $\partial_x u$;
ky	:	Differentiation array to compute $\partial_y u$;
k2	:	Differentiation matrix to compute $\partial^2 u$;
k4	:	Differentiation matrix to compute $\partial^2\partial^2 u$;
Lx	:	Period of u with respect to the independent variable x;
Ly	:	Period of u with respect to the independent variable y;
pmax	:	$N_x/2$;
qmax	:	$N_y/2$.

2.2.3 der_in_x_hat - subroutine that computes the partial derivative with respect to x

This subroutine returns the partial derivative with respect to the independent variable x given a matrix uhat that contains the Fourier coefficients of u. In other words, this subroutine calculates the r.h.s. of expression (2.2.1) by multiplying each column of uhat by kx (see Appendix C.2 for the implementation code). As it can be seen on its interface, which is given below, the result is returned in the variable dudx_hat.

```
subroutine der_in_x_hat (dudx_hat, uhat, kx, pmax, qmax)
    integer, intent(in) :: pmax, qmax
    real(8), dimension(0:pmax), intent(in) :: kx
    complex(8), dimension(0:pmax, 0:2*qmax-1), intent(in)  :: uhat
    complex(8), dimension(0:pmax, 0:2*qmax-1), intent(out) :: dudx_hat
end subroutine der_in_x_hat
```

Arguments:

dudx_hat	:	Fourier coefficients of $\partial_x u$;
uhat	:	Fourier coefficients of u;
kx	:	Differentiation array to compute$\partial_x u$;
pmax	:	$N_x/2$;
qmax	:	$N_y/2$.

2.2.4 der_in_y_hat - subroutine that computes the partial derivative with respect to y

On the other hand, the subroutine der_in_y_hat computes the r.h.s. of expression (2.2.2), which corresponds to the partial derivative with respect to the independent variable y. The result is calculated by multiplying each line of uhat by ky and it is returned in the variable dudy_hat (see Appendix C.3). In order to know which arguments compose this subroutine see the following interface.

```
subroutine der_in_y_hat (dudy_hat, uhat, ky, pmax, qmax)
    integer, intent(in) :: pmax, qmax
    real(8),    dimension(0:2*qmax-1), intent(in)          :: ky
    complex(8), dimension(0:pmax, 0:2*qmax-1), intent(in)  :: uhat
    complex(8), dimension(0:pmax, 0:2*qmax-1), intent(out) :: dudy_hat
end subroutine der_in_y_hat
```

Arguments:

dudy_hat	:	Fourier coefficients of $\partial_y u$;
uhat	:	Fourier coefficients of u;
ky	:	Differentiation array to compute$\partial_y u$;
pmax	:	$N_x/2$;
qmax	:	$N_y/2$.

2.2.5 lap_hat - subroutine that computes the Laplacian

The subroutine lap_hat is responsible for computing the Laplacian given a matrix uhat that contains the Fourier coefficients of u and its result is given through the variable lap_uhat. In a formal way, this subroutine calculates the r.h.s. of expression (2.2.3) by multiplying uhat by k2 (see Appendix C.4). Furthermore, the arguments which compose this subroutines and their order is shown in the following interface:

```
subroutine lap_hat (lap_uhat, uhat, k2, pmax, qmax)
    integer, intent(in) :: pmax, qmax
    real(8),    dimension(0:pmax, 0:2*qmax-1), intent(in)  :: k2
    complex(8), dimension(0:pmax, 0:2*qmax-1), intent(in)  :: uhat
    complex(8), dimension(0:pmax, 0:2*qmax-1), intent(out) :: lap_uhat
end subroutine lap_hat
```

Arguments:

lap_uhat	:	Fourier coefficients of $\partial^2 u$;
uhat	:	Fourier coefficients of u;
k2	:	Differentiation matrix to compute $\partial^2 u$;
pmax	:	$N_x/2$;
qmax	:	$N_y/2$.

2.2.6 bilap_hat - subroutine that computes two times the Laplacian

The bi-Laplacian operator is defined as $\partial^2\partial^2$, which in Fourier domain corresponds to multiply uhat by k4 (see Appendix C.5). Formally, this subroutine computes the r.h.s. of expression (2.2.4) and its interface is given below.

```
subroutine bilap_hat (bilap_uhat, uhat, k4, pmax, qmax)
    integer, intent(in) :: pmax, qmax
    real(8),    dimension(0:pmax, 0:2*qmax-1), intent(in)  :: k4
    complex(8), dimension(0:pmax, 0:2*qmax-1), intent(in)  :: uhat
    complex(8), dimension(0:pmax, 0:2*qmax-1), intent(out) :: bilap_uhat
end subroutine bilap_hat
```

Arguments:

bilap_uhat	:	Fourier coefficients of $\partial^2\partial^2 u$;
uhat	:	Fourier coefficients of u;
k4	:	Differentiation matrix to compute $\partial^2\partial^2 u$;
pmax	:	$N_x/2$;
qmax	:	$N_y/2$.

2.2.7 J_hat - subroutine that computes the Jacobian between two functions $f(x, y)$ and $g(x, y)$

J_hat subroutine calculates the Jacobian between two functions, $f(x, y)$ and $g(x, y)$, given their Fourier coefficients. Formally, this subroutine computes:

$$\mathcal{J}(f, g) = \begin{vmatrix} \partial_x f & \partial_y f \\ \partial_x g & \partial_y g \end{vmatrix} = (\partial_x f)(\partial_y g) - (\partial_y f)(\partial_x g),$$

which in the conservative form reads

$$\mathcal{J}(f,g) = \partial_x (f(\partial_y g)) - \partial_y (f(\partial_x g)).$$

As to compute the Jacobian function using its conservative form see Figure 2.2.2 which depicts the algorithm that was implemented. First of all, it is necessary to set the function g in the Fourier domain by applying a forward Fourier transform. The following steps consist of computing the partial derivatives, $\partial_y g$ and $\partial_x g$, by calling der_in_y_hat and der_in_x_hat subroutines, respectively. Afterwards, it is applied two backward Fourier transforms so that the sub-results are in the physical domain. In the fifth step, the multiplications are calculated and then, they are followed by a new forward Fourier transform since it will be necessary to compute $\partial_x (f(\partial_y g))$ and $\partial_y (f(\partial_x g))$. Again, these partial derivatives are calculated using the proper subroutines that were developed. Now, it is only necessary to compute $\partial_x (f(\partial_y g)) - \partial_y (f(\partial_x g))$ which corresponds to the last step in Figure 2.2.2.

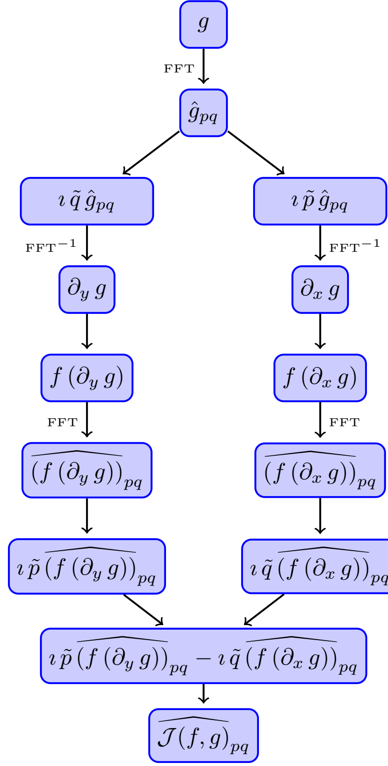


Figure 2.2.2: Computation of $\mathcal{J}(f,g)$ flowchart.

In the end, the result, $\widehat{\mathcal{J}(f,g)}_{pq}$, is returned in the output_hat variable as it can be seen by looking at the subroutine interface given bellow. Unlike previous subroutines, J_hat makes usage of my_fft2d.f95 module since it needs to compute some forward and backward FFTs. So, before calling this subroutine, make sure that the FFT environment is set up previously.

```
subroutine J_hat(output_hat, fhat, ghat, kx, ky, fft2d_dir_plan, &
    fft2d_inv_plan, Nx, Ny)
    integer,    intent(in) :: Nx, Ny
    integer(8), intent(in) :: fft2d_dir_plan, fft2d_inv_plan
    real(8),    dimension(0:Ny-1), intent(in) :: ky
    real(8),    dimension(0:Nx/2), intent(in) :: kx
    complex(8), dimension(0:Nx/2, 0:Ny-1), intent(in)  :: fhat, ghat
    complex(8), dimension(0:Nx/2, 0:Ny-1), intent(out) :: output_hat
subroutine J_hat
```

Arguments:

output_hat	:	Fourier coefficients of $\mathcal{J}\left(f\left(x,y\right),g\left(x,y\right)\right)$;
fhat	:	Fourier coefficients of f;
ghat	:	Fourier coefficients of g;
kx	:	Differentiation array to compute $\partial_x u$;
ky	:	Differentiation array to compute $\partial_y u$;
fft2d_dir_plan	:	An object that contains the data that FFTW needs to compute the forward FFT;
fft2d_inv_plan	:	An object that contains the data that FFTW needs to compute the backward FFT;
Nx	:	Number of rows present in matrices f and g;
Ny	:	Number of columns present in matrices f and g.

2.2.8 Verification and validation of my_derop.f95 subroutines

In order to test all subroutines within the module my_derop.f95, the following expression was solved numerically:

$$\tilde{\mathcal{A}}\psi = \mathcal{J}\left(\partial^2\psi,\Psi\right)+\mathcal{J}\left(\partial^2\Psi,\psi\right)\text{-}\nu\partial^2\partial^2\psi \tag{2.2.5}$$

where $\tilde{\mathcal{A}}$ is the linearized Navier-Stokes operator and ψ and Ψ were defined as follows:

$$\psi = \sum_{p=0}^{p_{max}}\sum_{q=0}^{q_{max}} \frac{1}{1+p+q}\ \cos\left(px+qy\right) + \frac{1}{1+pq}\ \sin\left(px+qy\right),$$

$$\Psi = \sum_{p=0}^{p_{max}}\sum_{q=0}^{q_{max}} \frac{1}{2+p+q}\ \cos\left(px+qy\right) + \frac{1}{2+pq}\ \sin\left(px+qy\right).$$

The analytical solution of the expression (2.2.5) is given by

$$\begin{aligned}
\tilde{\mathcal{A}}\psi = \; & \left(\sum_{p=0}^{p_{max}}\sum_{q=0}^{q_{max}} -\left(p^2+q^2\right)\left(-\frac{p}{1+p+q}\sin\left(px+qy\right)+\frac{p}{1+pq}\cos\left(px+qy\right)\right)\right) \\
& \left(\sum_{p=0}^{p_{max}}\sum_{q=0}^{q_{max}} -\frac{q}{2+p+q}\sin(px+qy)+\frac{q}{2+pq}\cos(px+qy)\right) \\
- & \left(\sum_{p=0}^{p_{max}}\sum_{q=0}^{q_{max}} -\left(p^2+q^2\right)\left(-\frac{q}{1+p+q}\sin\left(px+qy\right)+\frac{q}{1+pq}\cos\left(px+qy\right)\right)\right) \\
& \left(\sum_{p=0}^{p_{max}}\sum_{q=0}^{q_{max}} -\frac{p}{2+p+q}\sin(px+qy)+\frac{p}{2+pq}\cos(px+qy)\right) \\
+ & \left(\sum_{p=0}^{p_{max}}\sum_{q=0}^{q_{max}} -\left(p^2+q^2\right)\left(-\frac{p}{2+p+q}\sin(px+qy)+\frac{p}{2+pq}\cos(px+qy)\right)\right). \\
& \left(\sum_{p=0}^{p_{max}}\sum_{q=0}^{q_{max}} -\frac{q}{1+p+q}\sin(px+qy)+\frac{q}{1+pq}\cos(px+qy)\right) \\
- & \left(\sum_{p=0}^{p_{max}}\sum_{q=0}^{q_{max}} -\left(p^2+q^2\right)\left(-\frac{q}{2+p+q}\sin\left(px+qy\right)+\frac{q}{2+pq}\cos\left(px+qy\right)\right)\right) \\
& \left(\sum_{p=0}^{p_{max}}\sum_{q=0}^{q_{max}} -\frac{p}{1+p+q}\sin(px+qy)+\frac{p}{1+pq}\cos(px+qy)\right) \\
-\nu & \sum_{p=0}^{p_{max}}\sum_{q=0}^{q_{max}} \left(p^2+q^2\right)^2\left(\frac{1}{1+p+q}\cos\left(px+qy\right)+\frac{1}{1+pq}\sin\left(px+qy\right)\right)
\end{aligned} \tag{2.2.6}$$

Let u be the numerical solution of the expression (2.2.5) computed using the subroutines within the module my_derop.f95 and u_e the exact solution calculated using the expression (2.2.6). Table 2.4 presents the maximum absolute error for several tests by varying p_{max} and q_{max}.

p_{max}/q_{max}	N_x/N_y	$O\left(\|\|u-u_e\|\|_{max}\right)$
5	64	10^{-9}
8	128	10^{-8}
10	128	10^{-7}
20	256	10^{-6}

Table 2.4: Magnitude of the errors obtained from numerical experiences based on expression (2.2.5) where the numerical solution u is compared to the exact solution u_e, which was calculated using the expression (2.2.6).

Chapter 3

Numerical computation of the eddy viscosity and the large-scale transport coefficients

This chapter presents the steps that were taken to compute the eddy viscosity and the large-scale transport coefficients related to the two-dimensional incompressible Navier-Stokes equations which will be defined as ν_{eddy}, a and c from now on. Moreover, at the end of the chapter, the respective results are presented. In order to compute the eddy viscosity [3][11], ν_{eddy}, associated with the basic flow Ψ, the following equation must be solved:

$$\nu_{eddy}(\Psi,\nu) = \nu - < Q_1(\partial_y \Psi) > -2 < (\partial_x S)(\partial_y \Psi) > \tag{3.0.1}$$

where $< \cdot >$ denotes the average over the space periodicities and ν the molecular viscosity. The quantities Q_1 and S are the solutions of the auxiliary problems:

$$\tilde{\mathcal{A}} Q_1 = \partial_y \partial^2 \Psi, \tag{3.0.2}$$

$$\tilde{\mathcal{A}} S = \left(\partial_y \partial^2 \Psi\right) Q_1 + 2\,\mathcal{J}\left(\Psi, \partial_x Q_1\right) - \left(\partial_y \Psi\right)\left(\partial^2 \Psi\right) + 4\,\nu\,\partial_x\,\partial^2 Q_1. \tag{3.0.3}$$

Here $\tilde{\mathcal{A}}$ is the linearized Navier-Stokes operator:

$$\tilde{\mathcal{A}}\psi \equiv \mathcal{J}(\partial^2\psi, \Psi) + \mathcal{J}(\partial^2\Psi, \psi) - \nu\partial^2\partial^2\psi.$$

The other two coefficients, a and c [11], are calculated by solving the following equations, respectively:

$$a = 1 + 2\,A_1 + 4\,B_1 \quad \text{and} \quad c = -4\,A_2 - 4\,(B_2 + B_3)\,.$$

Where,

$$A_1 =< (\partial_x Q_1)\,(\partial_y Q_2) >, \quad B_1 =< (\partial_y Y_{12})\,(\partial_x \Psi) >,$$

$$A_2 = -\tfrac{1}{2} < (\partial_y Q_1)\,(\partial_y Q_2) >, \quad B_2 = \tfrac{1}{2} < (\partial_x Y_{21})\,(\partial_x \Psi) >, \quad B_3 = \tfrac{1}{2} < (\partial_x Y_{12})\,(\partial_x \Psi) >,$$

$$\tilde{\mathcal{A}} Y_{12} = \left(\partial_y \partial^2 Q_1\right)(\partial_x Q_2) - \left(\partial_x \partial^2 Q_1\right)(\partial_y Q_2) + \partial_y \partial^2 Q_2,$$

$$\tilde{\mathcal{A}} Y_{21} = \left(\partial_y \partial^2 Q_2\right)(\partial_x Q_1) - \left(\partial_x \partial^2 Q_2\right)(\partial_y Q_1) - \partial_x \partial^2 Q_1$$

and

$$\tilde{A}Q_2 = -\partial_x \partial^2 \Psi.$$

Although it is not said explicitly in the following sections, it is assumed that every partial derivatives, Laplacian or Fourier transforms are performed by taking advantage of my_fft2d and my_derop modules which were described in Chapter 2.

3.1 Computation of the eddy viscosity coefficient

In order to solve Equation (3.0.1), it is required to find first Q_1 and S which are the solutions of Equation (3.0.2) and Equation (3.0.3), respectively. As a means of solving these two auxiliary problems, numerical approaches were used such as the iterative method, second-order Runge-Kutta method and slaved leap-frog method. However, as it will be seen, only the last one is able to converge in all cases. These methods were chosen for their simplicity and also because of previous experience using them. After analyzing Equation (3.0.2) and Equation (3.0.3), it is possible to conclude that they follow the same structure which is

$$\tilde{A}\psi = b$$

and thus, by solving the first auxiliary problem (Equation (3.0.2)), the second one will be also automatically solved by just providing the appropriate r.h.s. member. Then, Equation (3.0.1) can be solved. To do so, Equation (3.0.1) was slightly modified so that the number of FFTs needed is minimized:

$$\begin{aligned} \nu_{eddy}(\Psi,\nu) &= \nu - < Q_1(\partial_y \Psi) > -2 < (\partial_x S)(\partial_y \Psi) > \\ &= \nu - < Q_1(\partial_y \Psi) + 2(\partial_x S)(\partial_y \Psi) > \\ &= \nu - < (Q_1 + 2(\partial_x S))\, \partial_y \Psi > \end{aligned} \tag{3.1.1}$$

Now, by recalling that in the Fourier domain the first wave number corresponds to the mean of the function, i.e.

$$\hat{f}_{00} = \frac{1}{L_x L_y} \int_0^{L_x} \int_0^{L_y} f(x,y)\, dy\, dx =< f(x,y) >,$$

Equation 3.1.1 can be easily solved by writing it in the Fourier domain:

$$\nu_{eddy}(\Psi,\nu) \quad = \quad \nu - \widehat{((Q_1 + 2(\partial_x S))\, \partial_y \Psi)}_{00}.$$

3.1.1 Simple iterative method - algorithm

The first attempt to solve Equation (3.0.2) was by using the simple iterative method. First, one has to write the former equation in the Fourier domain which leads

$$\mathcal{J}(\partial^2 Q_1, \Psi) + \mathcal{J}(\partial^2 \Psi, Q_1) - \nu \partial^2 \partial^2 Q_1 = \partial_y \partial^2 \Psi,$$

into

$$\hat{J}_{pq} + \hat{G}_{pq} - \nu k^4 \hat{Q}_{pq} = \hat{b}_{pq}$$

where $\hat{J}_{pq} = \widehat{\mathcal{J}(\partial^2 Q_1, \Psi)}_{pq}$, $\hat{G}_{pq} = \widehat{\mathcal{J}(\partial^2 \Psi, Q_1)}_{pq}$, $k^2 = \tilde{p}^2 + \tilde{q}^2$, $k^4 = k^2 k^2$ and $\hat{b}_{pq} = -\imath\, \tilde{q}\, k^2\, \hat{\Psi}_{pq}$. By keeping in mind that the first wave number corresponds to the mean, it is possible to assume that $\hat{Q}_{00} = 0$ and rearrange the previous equation in the following way

$$\hat{Q}_{pq} = \frac{1}{\nu k^4} \left[\hat{J}_{pq} + \hat{G}_{pq} - \hat{b}_{pq} \right] \text{ if } (p,q) \neq (0,0).$$

Finally, in order to find the numeric solution using the simple iterative method, the system of equations (3.1.2) must be solved.

$$\begin{cases} \hat{Q}_{pq}^{n+1} = \frac{1}{\nu k^4}\left[\hat{J}_{pq}^n + \hat{G}_{pq}^n - \hat{b}_{pq}\right] & \text{if } (p,q) \neq (0,0) \\ \hat{Q}_{00}^{n+1} = 0 \\ \hat{Q}_{pq}^{0} \quad = \hat{\Psi}_{pq} \end{cases} \tag{3.1.2}$$

Figure 3.1.1 shows an illustration on how the simple iterative method works step by step to solve the system of equations (3.1.2) . Firstly, it is necessary to give as input an initial condition, $\hat{Q}_{pq}^0$, $\hat{b}_{pq}$ and $\hat{\Psi}_{pq}$. Then, all terms that will not vary on each iteration are computed so they can be stored in memory and used as needed in the loop. To avoid the problem of $k^4 = 0$ in the second step, it is assumed that the result is zero for $p = q = 0$ since it is known that $\hat{Q}_{00}^n = 0$ for all n. The following steps in the flowchart will compute the required terms to compute $\hat{Q}_{pq}^{n+1}$ which are $\hat{J}_{pq}^n$ and $\hat{G}_{pq}^n$. These terms can be easily calculated by calling the subroutine presented in Section 2.2.7 with the appropriate arguments. After computing $\hat{Q}_{pq}^{n+1}$, the steps are repeated until the absolute value of the difference between $\hat{Q}_{pq}^{n+1}$ and $\hat{Q}_{pq}^n$ is smaller than a given tolerance value TOL. At the end, and if the numerical method converges, the solution for the system of equations (3.1.2) is found and given by $\hat{Q}_{pq}^*$.

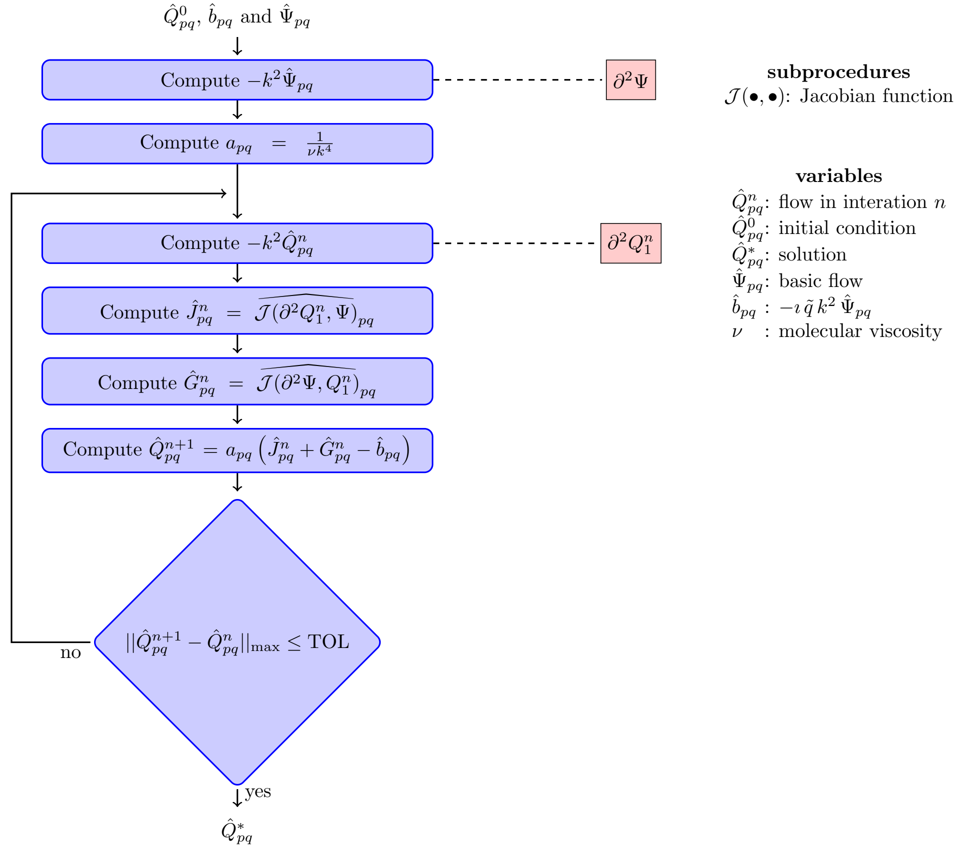


Figure 3.1.1: Simple iterative method flowchart.

After finding Q_1, it is possible to compute S. The only change that needs to be done to the previous algorithm is to set

$$b = \left(\partial_y \partial^2 \Psi\right) Q_1 + 2\,\mathcal{J}\left(\Psi, \partial_x Q_1\right) - \left(\partial_y \Psi\right)\left(\partial^2 \Psi\right) + 4\,\nu\,\partial_x\,\partial^2 Q_1$$

which in the Fourier domain corresponds to

$$\hat{b}_{pq} = \widehat{\left(\left(\partial_y \partial^2 \Psi\right) Q_1\right)}_{pq} + 2\, \widehat{\mathcal{J}\left(\Psi, \partial_x Q_1\right)}_{pq} - \widehat{\left(\left(\partial_y \Psi\right)\left(\partial^2 \Psi\right)\right)}_{pq} - 4\, \nu\, \imath\, \tilde{p}\, k^2\, \widehat{Q}_{pq},$$

and give an initial condition $\hat{S}^0_{pq} = 0$ instead of $\hat{Q}^0_{pq}$.

In Figure 3.1.2 is presented a flowchart of the algorithm that was used to compute this new $\hat{b}_{pq}$ that corresponds to the r.h.s of the Equation 3.0.3 in the Fourier domain. Two forward Fourier transforms are computed in the first step, one applied to Ψ and the other to Q_1. Afterwards, all the partial derivatives and Laplacian of Ψ and Q_1 are computed in the Fourier domain. Then, it is required to go back to the physical domain in order to compute the multiplications that appear in the expression of $\hat{b}_{pq}$. At the end, it is applied a forward Fourier transform to each sub-result and then they are all summed up as shown in the last step in Figure 3.1.2.

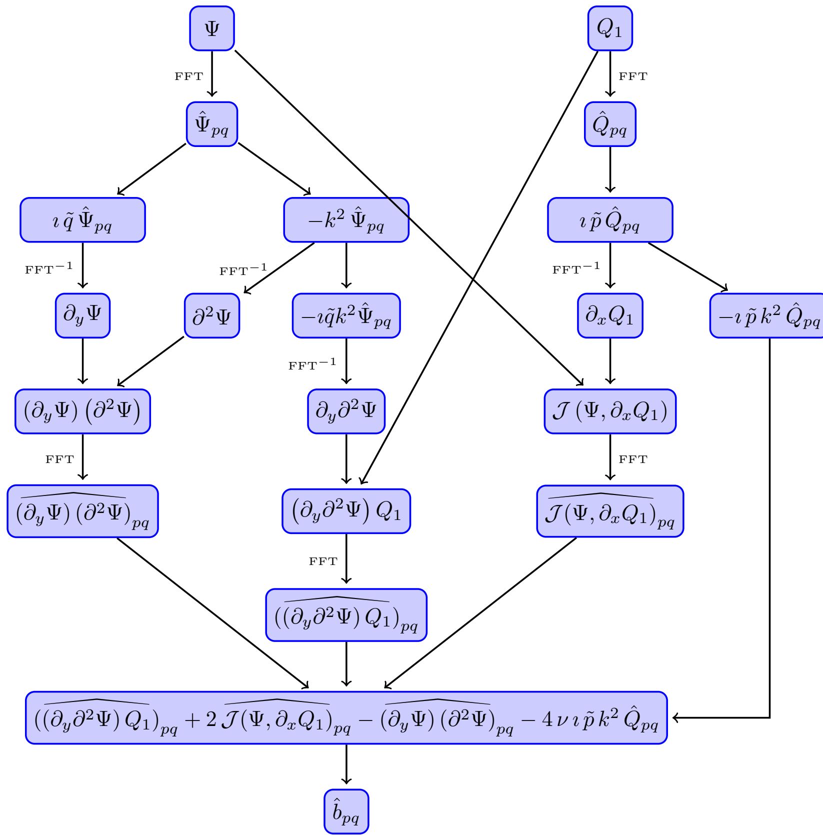


Figure 3.1.2: Flowchart to compute the r.h.s. of Equation 3.0.3.

3.1.2 Simple iterative method - modified algorithm

Aiming at a more robust method, some modifications were made to the previous algorithm. Due to the presence of non-linear terms, aliases are originated and so they need to be removed. Thus, the first improvement made was by dealiasing the result at the end of each iteration. Several methods exist to prevent aliasing such as zero-padding (3/2-rule), truncating (2/3-rule) and phase shift. In this case, the truncating method was used, since it is simple to apply and it has a low cost in computing time. The basic idea is to define a set of wavenumbers which will be forced to be zero every time a non-linear term is computed. Firstly, let

$$k_{\text{trunc}} = \left\lfloor \frac{1}{3} \min\left\{N_x, N_y\right\} \right\rfloor$$

be the truncation radius in which every wavenumber outside it will be forced to be zero after a non-linear computation is made. A wavenumber, $\hat{u}_{pq}$, is said to be outside if $\left(p^2+q^2\right) > k_{\text{trunc}}^2$.

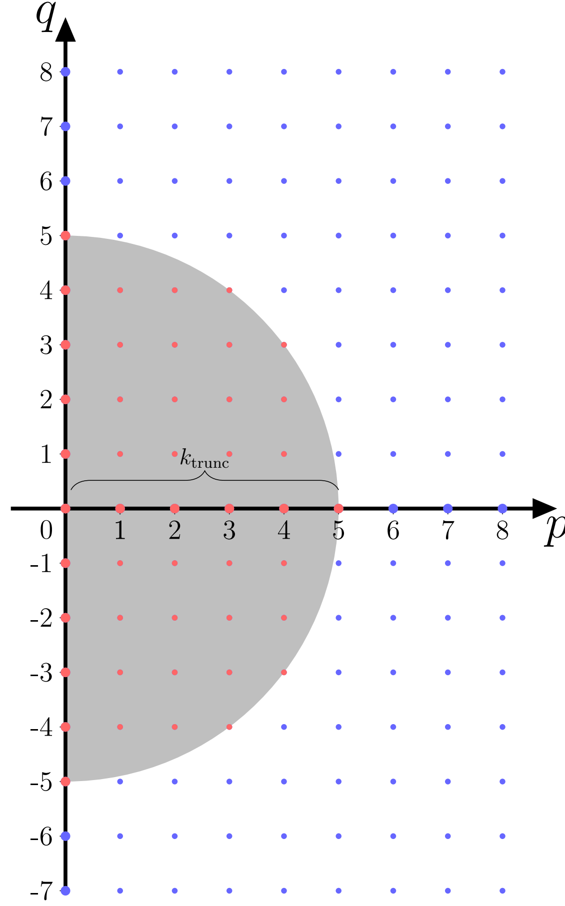


Figure 3.1.3: Dealising illustration assuming $N_x = N_y = 16$ where the blue dots represent all wavenumbers that will be contaminated by aliases whenever a non-linear computation is performed.

In Figure 3.1.3 an example is presented by assuming $N_x = N_y = 16$. All wavenumbers in blue, which are outside the semicircle, are going to be forced to be zero whenever a non-linear computation is done. On the other hand, the wavenumbers in red are those which will never be contaminated by the aliases. Although the numerical result is improved, this method has a big disadvantage because it forces more than half of the matrix to be zero, i.e., it cannot be used. An efficient strategy to apply the dealising is to take advantage of the term $1/\left(\nu k^4\right)$ by setting to zero every value associated with a wavenumber that is not within k_{trunc}. In this way, the dealiasing is implicitly applied while $\hat{Q}_{pq}^{n+1}$ is computed. Another improvement made was to introduce mixing times to the algorithm, which will make the solution more precise. The main idea is to compute $\hat{Q}_{pq}^{\prime n}$ by using the following expression

$$\hat{Q}_{pq}^{\prime n} = \frac{1}{2} Q_{pq}^{n} + \frac{1}{4}\left(\hat{Q}_{pq}^{n-1} + \hat{Q}_{pq}^{n+1}\right),$$

at every t_N iterations in order to make the result more precise, i.e., at iterations $t_N, 2\,t_N, 3\,t_N, \ldots$, a correction is made using $\hat{Q}_{pq}^{\prime n}$. Figure 3.1.4 presents the flowchart of the modified algorithm.

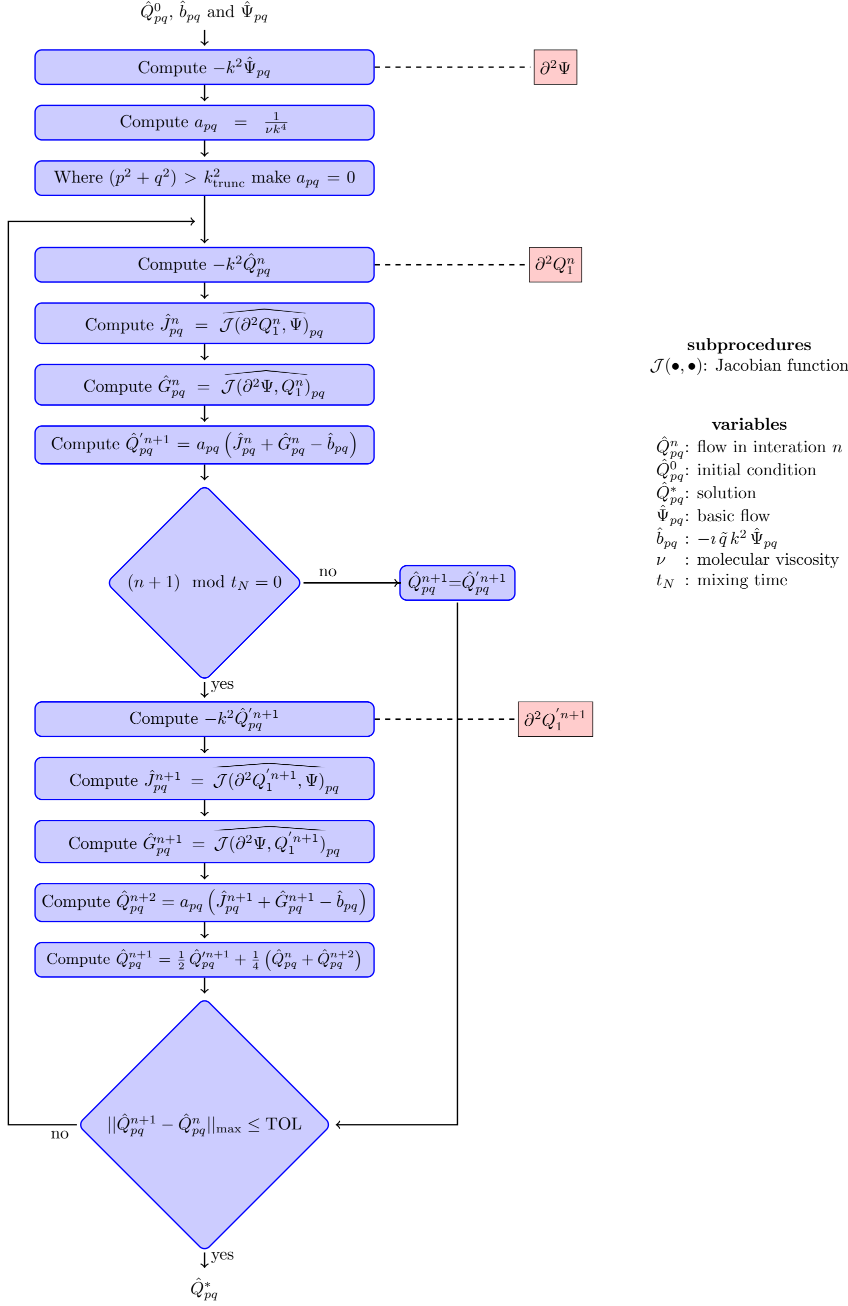


Figure 3.1.4: Modified simple iterative method flowchart.

The last refinement made takes into consideration the time step, Δt, that should be chosen in order to make the program efficient without making it diverge. The Courant-Friedrichs-Lewy or CFL condition, is a necessary condition for convergence while solving partial differential equations. It states that

$$C = \frac{u_{\max}\,\Delta t}{\Delta x} \leq C_{\max}$$

where C is known as the Courant number, Δx is the space step used to discretize the physical domain and $u_{\max} = \max\left(|u|\right)$. By keeping in mind that the space discretization assumes $\Delta x = L_x/N_x$ and by knowing that $k_{\max} = N_x/2$ then it is possible to conclude that $1/\Delta x = 2\,k_{\max}/L_x$. As a result, it is possible to rewrite the previous expression in the following way

$$\Delta t \leq \frac{L_x\, C_{\max}}{2\, u_{\max} k_{\max}}.$$

However, the order of magnitude is what it matters here and thus, it is possible to simplify the expression as follows:

$$\Delta t \leq \frac{C_{\max}}{u_{\max}\, k_{\max}}.$$

Despite $C_{\max}$ being assumed to be equal to 1 when explicit time integration schemes are used, $C_{\max} = 0.8$ was used instead, due to the computer precision. This way there is no risk that the CFL condition is not respected due to the finite precision of the computer. See Appendix D.1 for the modified iterative method implementation code.

3.1.3 Second-order Runge-Kutta method - algorithm

In this new approach, the second-order Runge-Kutta (RK2) method was used in order to find Q_1 and S (Equation (3.0.2) and Equation (3.0.3), respectively). Let an initial value problem be specified as follows

$$\begin{cases} \dot{y} = f(t, y) \\ y(t_0) = y_0 \end{cases}.$$

Here, y is an unknown function of time, function f and constants t_0 and y_0 are given. To solve this ordinary differential equation, use the RK2 method, which can be summed up through the set of equations presented in (3.1.3).

$$\begin{aligned} k_{rk1} &= \Delta t\, f\,(t,\, y_n) \\ k_{rk2} &= \Delta t\, f\left(t + \frac{1}{2}\,\Delta t,\, y_n + \frac{1}{2} k_{rk1}\right) \\ y_{n+1} &= y_n + k_{rk2} + O\left(h^3\right) \end{aligned} \tag{3.1.3}$$

Before using the RK2 method to solve Equation (3.0.2), it is required to write the previous equation in an equivalent form such as

$$\dot{Q}_1 + \mathcal{J}(\partial^2 Q_1, \Psi) + \mathcal{J}(\partial^2 \Psi, Q_1) - \nu \partial^2 \partial^2 Q_1 = \partial_y \partial^2 \Psi$$

which has the following structure

$$\dot{\psi} + \tilde{A}\psi = b.$$

Now, as in Section 3.1.1, one has to write the equation in the Fourier domain

$$\dot{\hat{Q}}_{pq} + \hat{J}_{pq} + \hat{G}_{pq} - \nu k^4 \hat{Q}_{pq} = \hat{b}_{pq},$$

which is equivalent to

$$\dot{\hat{Q}}_{pq} = \hat{b}_{pq} - \hat{J}_{pq} - \hat{G}_{pq} + \nu\, k^4\, \hat{Q}_{pq}, \tag{3.1.4}$$

where $\hat{J}_{pq} = \widehat{\mathcal{J}(\partial^2 Q_1, \Psi)}_{pq}$, $\hat{G}_{pq} = \widehat{\mathcal{J}(\partial^2 \Psi, Q_1)}_{pq}$, $k^2 = \tilde{p}^2 + \tilde{q}^2$, $k^4 = k^2 k^2$ and $\hat{b}_{pq} = -\imath\, \tilde{q}\, k^2\, \hat{\Psi}_{pq}$. Finally, so that Q_1 is found, it is required to solve the initial value problem

$$\begin{cases} \dot{\hat{Q}}_{pq} = \hat{b}_{pq} - \hat{J}_{pq} - \hat{G}_{pq} + \nu\, k^4\, \hat{Q}_{pq} \\ \hat{Q}^0_{pq} = 0 \end{cases}. \tag{3.1.5}$$

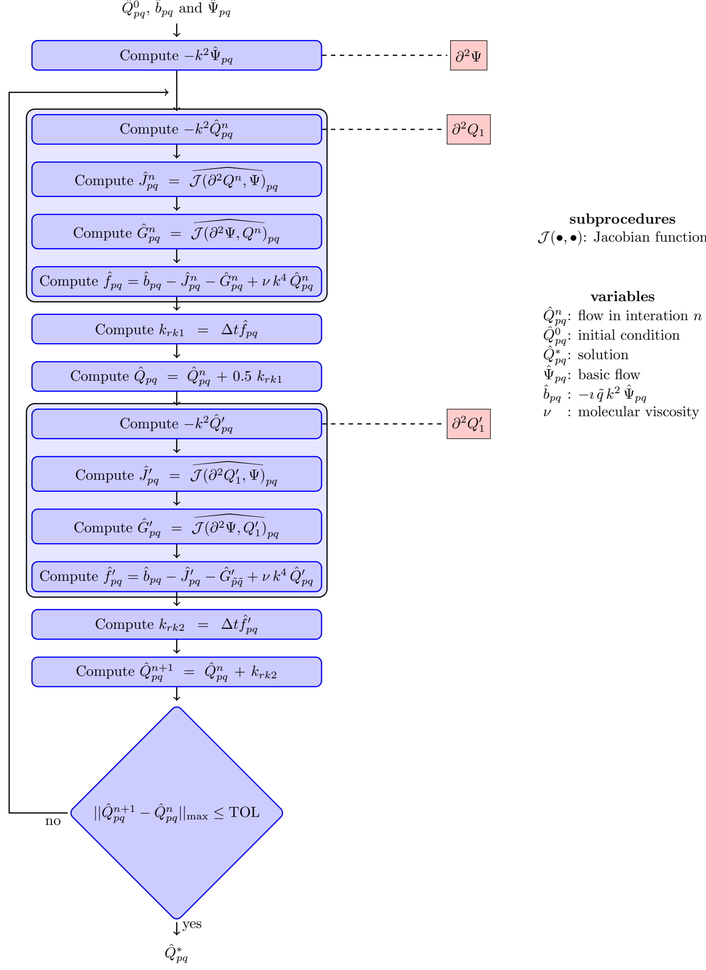


Figure 3.1.5: RK2 method flowchart.

By taking into consideration Equation (3.1.4) and the RK2 method the following equations appear:

$$
\begin{aligned}
k_{rk1} &= \Delta t\, \left(\hat{b}_{pq} - \hat{J}^n_{pq} - \hat{G}^n_{pq} + \nu\, k^4\, \hat{Q}^n_{pq}\right), \\
k_{rk2} &= \Delta t\, \left(\hat{b}_{pq} - \widehat{\mathcal{J}(\partial^2 Q_1'^n, \Psi)}_{pq} - \widehat{\mathcal{J}(\partial^2 \Psi, Q_1'^n)}_{pq} + \nu\, k^4\, \left(\hat{Q}^n_{pq} + 0.5\, k_{rk1}\right)\right), \\
\hat{Q}^{n+1}_{pq} &= \hat{Q}^n_{pq} + k_{rk2} + O(\Delta t^3),
\end{aligned}
$$

where $Q_1'^n$ corresponds to $\hat{Q}^n_{pq} + 0.5\, k_{rk1}$ in the physical domain.

Figure 3.1.5 illustrates the resulting algorithm using RK2 method to solve the system of equations (3.1.5) and its implementation can be found in Appendix D.2. If the RK2 method converges, it will converge to the solution of the Equation (3.0.2), i.e., if $\lim_{t\to\infty} Q_1(t, x, y)$ exists, it will converge to the solution of $\tilde{A}Q_1 = \partial_y \partial^2 \Psi$. As in the simple iterative method, it is required to provide to the algorithm as input the initial condition $\hat{Q}^0_{pq}$, $\hat{b}_{pq}$ and $\hat{\Psi}_{pq}$. Then, it is computed $\partial^2\Psi$ and stored in memory since it is constant over time. Inside the first block, all terms necessary to compute k_{rk1} are computed. Subsequently, all terms needed to compute k_{rk2} are calculated and finally $\hat{Q}^{n+1}_{pq}$ is updated. These steps are repeated until the absolute value of the difference between $\hat{Q}^{n+1}_{pq}$ and $\hat{Q}^n_{pq}$ is smaller then a given tolerance value TOL. At the end, and if the numerical method converges, the numerical solution for the system of equations (3.1.5) is found and given by $\hat{Q}^*_{pq}$.

Again, after finding Q_1 it is possible to compute S, and the only change that needs to be made on the algorithm is to set

$$\hat{b}_{pq} = \widehat{\left(\left(\partial_y \partial^2 \Psi\right) Q_1\right)}_{pq} + 2\,\widehat{\mathcal{J}\left(\Psi, \partial_x Q_1\right)}_{pq} - \widehat{\left(\left(\partial_y \Psi\right)\left(\partial^2 \Psi\right)\right)}_{pq} - 4\,\nu\,\imath\,\tilde{p}\,k^{\mathbf{2}}\,\widehat{Q}_{pq}.$$

and give an initial condition $\hat{S}^0_{pq} = 0$ instead of $\hat{Q}^0_{pq}$. Unfortunately, this method does not converge for any of the tested cases, but it will be important for the slaved leap-frog method as it will be shown in the next section.

3.1.4 Slaved leap-frog method - algorithm

The last method that was taken into consideration was the slaved leap-frog method which works properly in all cases that were tested. Its scheme is

$$q_{n+1} = e^{-2\alpha\Delta t} q_{n-1} + \frac{1 - e^{-2\alpha\Delta t}}{\alpha} h_n \tag{3.1.6}$$

by assuming the ordinary differential equation

$$\dot{q}(t) = -\alpha\, q(t) + h(t).$$

The first thought would be to use the same equivalent equation as in Section (3.1.3),

$$\begin{aligned} \dot{\hat{Q}}_{pq} &= \quad \nu\,k^4\,\hat{Q}_{pq} + \hat{b}_{pq} - \hat{J}_{pq} - \hat{G}_{pq} \\ &= -\left(-\nu\,k^4\right)\hat{Q}_{pq} + \hat{b}_{pq} - \hat{J}_{pq} - \hat{G}_{pq} \end{aligned}$$

However, a further analysis shows that it is not a good choice because it will not allow the method to converge. In this case $\alpha = -\nu\,k^4$ and it is known that both ν and k^4 are always positive integers, thus the exponential in Equation (3.1.6) will always be positive, which means that $\lim_{t\to\infty} Q(t,x,y) = +\infty$. So as to overcome this problem, a slightly different equivalent equation was taken into consideration

$$\partial^2 \dot{Q}_1 + \mathcal{J}(\partial^2 Q_1, \Psi) + \mathcal{J}(\partial^2 \Psi, Q_1) - \nu \partial^2 \partial^2 Q_1 = \partial_y \partial^2 \Psi \tag{3.1.7}$$

which in the Fourier domain corresponds to

$$-k^2 \dot{\hat{Q}}_{pq} + \hat{J}_{pq} + \hat{G}_{pq} - \nu\,k^4\,\hat{Q}_{pq} = \hat{b}_{pq},$$

where $\hat{J}_{pq} = \widehat{\mathcal{J}(\partial^2 Q_1, \Psi)}_{pq}$, $\hat{G}_{pq} = \widehat{\mathcal{J}(\partial^2 \Psi, Q_1)}_{pq}$, $k^2 = \tilde{p}^2 + \tilde{q}^2$, $k^4 = k^2 k^2$ and $\hat{b}_{pq} = -\imath\,\tilde{q}\,k^{\mathbf{2}}\,\hat{\Psi}_{pq}$. As before, by knowing that $\hat{Q}_{00} = 0$, it is possible to rewrite the previous equation as

$$\dot{\hat{Q}}_{pq} = -\nu\,k^2\,\hat{Q}_{pq} + \frac{1}{k^{\mathbf{2}}}\left(\hat{J}_{pq} + \hat{G}_{pq} - \hat{b}_{pq}\right).$$

Now, the resulting slaved leap-frog scheme is

$$\begin{aligned} \hat{Q}^{n+1}_{pq} &= e^{-2\nu k^{\mathbf{2}}\Delta t} q_{n-1} + \frac{1 - e^{-2\nu k^{\mathbf{2}}\Delta t}}{\nu\,k^{\mathbf{2}}}\left(\frac{1}{k^{\mathbf{2}}}\left(\hat{J}_{pq} + \hat{G}_{pq} - \hat{b}_{pq}\right)\right) \\ &= e^{-2\nu k^{\mathbf{2}}\Delta t} q_{n-1} + \frac{1 - e^{-2\nu k^{\mathbf{2}}\Delta t}}{\nu\,k^4}\left(\hat{J}_{pq} + \hat{G}_{pq} - \hat{b}_{pq}\right) \end{aligned}$$

and since the exponential is negative, in the long run the method will converge to the solution of $\tilde{\mathcal{A}} Q_1 = \partial_y \partial^2 \Psi$. Note that the initial value problem that is being solved is

$$\begin{cases} \dot{\hat{Q}}_{pq} = -\nu\,k^2\,\hat{Q}_{pq} + \frac{1}{k^{\mathbf{2}}}\left(\hat{J}_{pq} + \hat{G}_{pq} - \hat{b}_{pq}\right) \\ \hat{Q}^0_{pq} = 0 \end{cases}$$

and that the slaved leap-frog method needs $\hat{Q}^0_{pq}$ and $\hat{Q}^1_{pq}$ to start. As a result this method will always need another numerical method in order to compute $\hat{Q}^1_{pq}$. This is in fact the main disadvantage of using this method. Since the RK2 was already implemented previously, this was the numerical method used to assist the slaved leap-frog method.

Figure 3.1.6 shows the flowchart of the resulting algorithm. Proving that an initial condition is given, $\hat{b}_{pq}$ and $\hat{\Psi}_{pq}$, the first three steps consist of computing all terms that are constant over time. Afterwards, $\hat{Q}^1_{pq}$ is computed, by using RK2 and then the slaved leap-frog comes to action. In each iteration, $\partial^2 Q_1^n$ is computed first, then the non-linear terms and finally Q_1^{n+1}. Note that the non-linear terms can be easily computed by using the subroutine presented in Section 2.2.7. Until the absolute value of the difference between $\hat{Q}^{n+1}_{pq}$ and $\hat{Q}^n_{pq}$ is smaller than a given tolerance value TOL, the process is repeated and if the method converges, it will converge to the solution of $\tilde{\mathcal{A}} Q_1 = \partial_y \partial^2 \Psi$ which is given by $\hat{Q}^*_{pq}$.

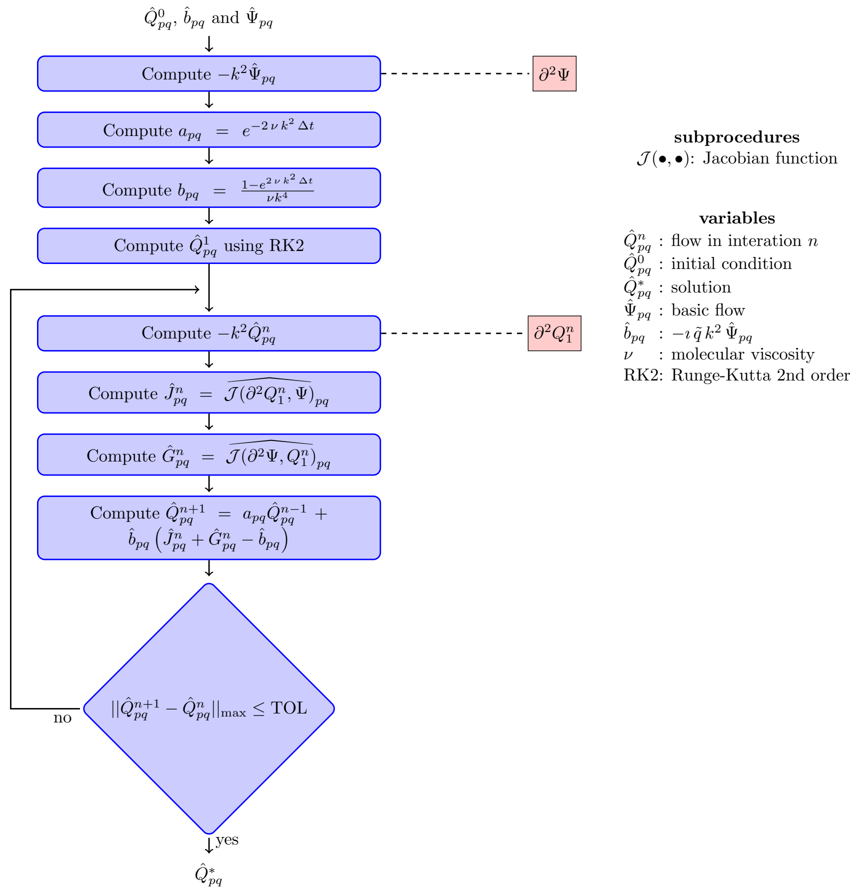


Figure 3.1.6: Slaved leap-frog method flowchart.

Like in the previous methods that were presented, after finding Q_1, it is possible to compute S without much effort. The only change that needs to be made to the previous algorithm is to set

$$\hat{b}_{pq} = \widehat{\left(\left(\partial_y \partial^2 \Psi\right) Q_1\right)}_{pq} + 2\,\widehat{\mathcal{J}\left(\Psi, \partial_x Q_1\right)}_{pq} - \widehat{\left(\left(\partial_y \Psi\right)\left(\partial^2 \Psi\right)\right)}_{pq} - 4\,\nu\,\imath\,\tilde{p}\,k^{\mathbf{2}}\,\widehat{Q}_{pq},$$

and give an initial condition $\hat{S}^0_{pq} = 0$ instead of $\hat{Q}^0_{pq}$.

3.1.5 Slaved leap-frog method - modified algorithm

In this section, a modified version of the slaved leap-frog method is presented in order to improve the solution. The ideas are the same to those that were applied to the simple iterative method, Section 3.1.2, i.e., implementation of dealiasing, mixing times and CFL condition. The resulting algorithm is shown in Figure 3.1.7 and its implementation can be seen in Appendix D.3.

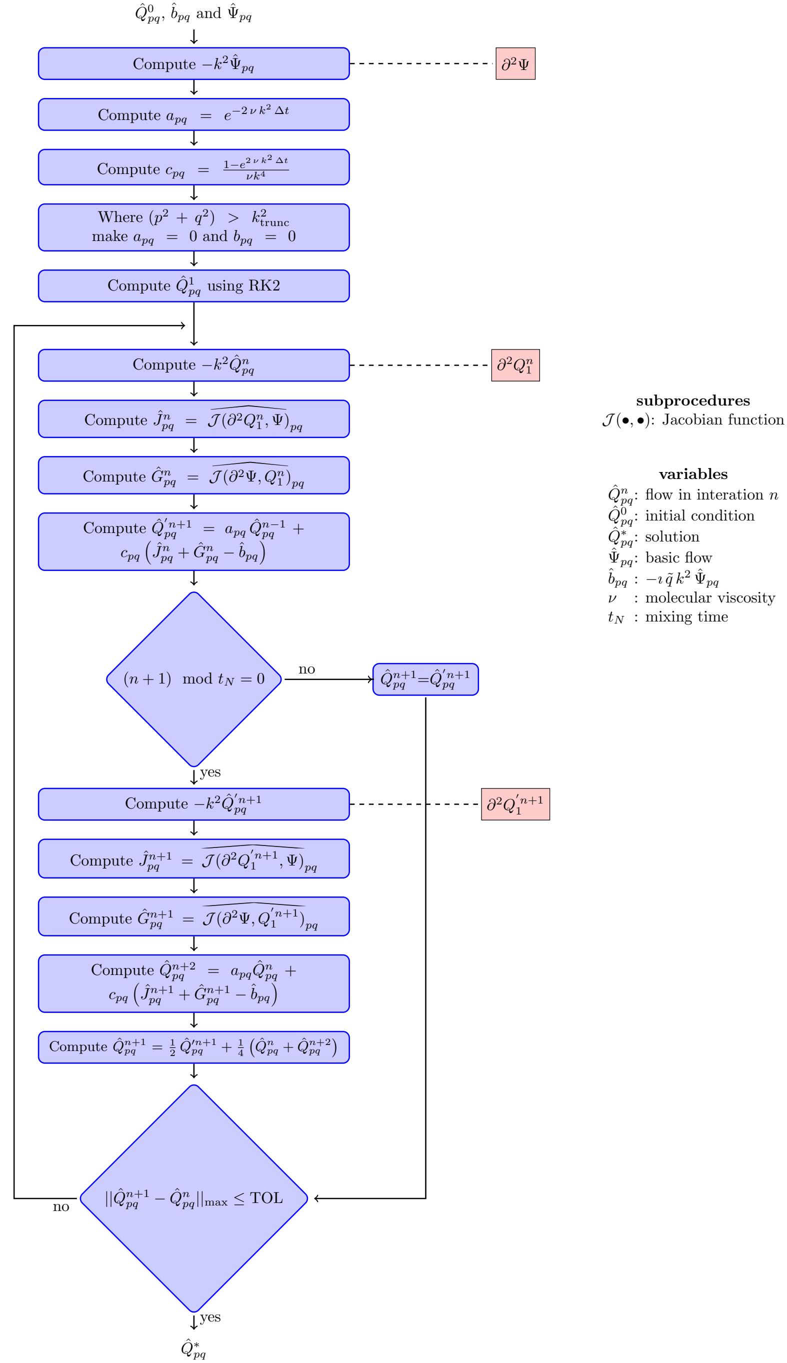


Figure 3.1.7: Modified slaved leap-frog method flowchart.

3.2 Computation of the large-scale transport coefficient a

Most part of the work has already been done while solving ν_{eddy}, so now computing the coefficient a will be quite straightforward. First, it is important to recall that the equation associated with the coefficient a is

$$a = 1 + 2\,A_1 + 4\,B_1, \tag{3.2.1}$$

where

$$A_1 =< (\partial_x Q_1)\,(\partial_y Q_2) >, \quad B_1 =< (\partial_y Y_{12})\,(\partial_x \Psi) >,$$

$$\tilde{\mathcal{A}} Q_2 = -\partial_x \partial^2 \Psi. \tag{3.2.2}$$

and

$$\tilde{\mathcal{A}} Y_{12} = \left(\partial_y \partial^2 Q_1\right)(\partial_x Q_2) - \left(\partial_x \partial^2 Q_1\right)(\partial_y Q_2) + \partial_y \partial^2 Q_2. \tag{3.2.3}$$

Just like for the coefficient ν_{eddy}, it is required to solve the auxiliary problem which has Q_1 as its solution. In order to find Q_2 and Y_{12}, Equation (3.2.2) and Equation (3.2.3) must be solved, respectively. Note that the equations related to Q_1, Q_2 and Y_{12} have the same structure,

$$\tilde{\mathcal{A}} \psi = b,$$

and so the numeric methods used to compute Q_1, see Section 3.1, are also useful to find Q_2 and Y_{12} providing that the correct b is given. In case of Q_2, the r.h.s. of the equation is easily computed. By assuming that $b = -\partial_x \partial^2 \Psi$, then in the Fourier domain the result is $\hat{b}_{pq} = \imath\, \tilde{p}\, k^2\, \hat{\Psi}_{pq}$. On the other hand, to compute the r.h.s. of the equation associated with Y_{12} requires a bit more work. Nonetheless, it is still quite simple to compute if the subroutines that were presented in Section 2.2 are used. Figure 3.2.1 shows the steps that can be followed to compute $\hat{b}_{pq} = \widehat{\left(\partial_y \partial^2 Q_1\right)(\partial_x Q_2) - \left(\partial_x \partial^2 Q_1\right)(\partial_y Q_2) + \partial_y \partial^2 Q_2}_{pq}$. The first step in the flowchart corresponds to the computation of the Fourier transforms of Q_1 and Q_2. Afterwards, the respective derivatives in the Fourier domain are computed and then backward Fourier transform are applied so that the multiplication can be performed. Finally, the terms are added up and a Fourier transform is applied.

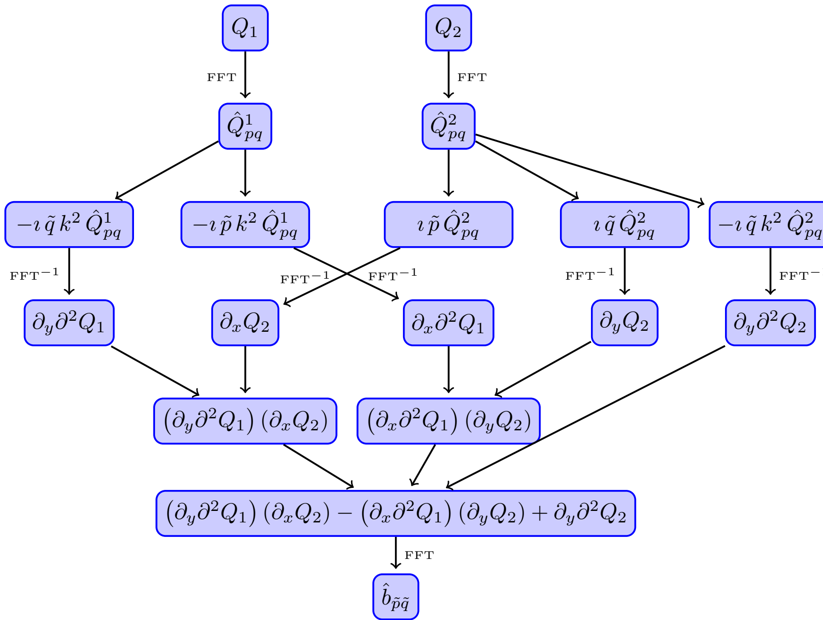


Figure 3.2.1: Flowchart to compute the r.h.s. of Equation (3.2.3).

As soon as Q_1, Q_2 and Y_{12} are found, it is possible to compute the terms A_1 and B_1. Since these two terms are just averages, it is correct to state that

$$A_1 = \widehat{\left(\left(\partial_x Q_1\right)\left(\partial_y Q_2\right)\right)}_{00} \text{ and } B_1 = \widehat{\left(\partial_y Y_{12}\right)\left(\partial_x \Psi\right)}_{00}$$

which means that

$$a = 1 + 2\,\widehat{\left(\left(\partial_x Q_1\right)\left(\partial_y Q_2\right)\right)}_{00} + 4\,\widehat{\left(\partial_y Y_{12}\right)\left(\partial_x \Psi\right)}_{00}.$$

3.3 Computation of the large-scale transport coefficient c

The equation associated with the coefficient c is

$$c = -4\,A_2 - 4\,\left(B_2 + B_3\right),$$

where

$$A_2 = -\tfrac{1}{2} < \left(\partial_y Q_1\right)\left(\partial_y Q_2\right) >, \quad B_2 = \tfrac{1}{2} < \left(\partial_x Y_{21}\right)\left(\partial_x \Psi\right) >, \quad B_3 = \tfrac{1}{2} < \left(\partial_x Y_{12}\right)\left(\partial_x \Psi\right) >,$$

and

$$\tilde{\mathcal{A}} Y_{21} = \left(\partial_y \partial^2 Q_2\right)\left(\partial_x Q_1\right) - \left(\partial_x \partial^2 Q_2\right)\left(\partial_y Q_1\right) - \partial_x \partial^2 Q_1. \tag{3.3.1}$$

By comparing the set of equations of coefficient c with the set of equations of a, it is possible to see that they are very similar. Hence, the same strategy applied to compute a will be applied here. Firstly, the solutions Q_1, Q_2 and Y_{21} must be found using a numerical method. The auxiliary problem related to Y_{21} follows the same structure as all others that have been seen until now, and so the only effort that must be made in order to calculate it is to compute the equation r.h.s. . This is depicted in Figure 3.3.1 and it follows the basic approach that was seen in Section 3.2.

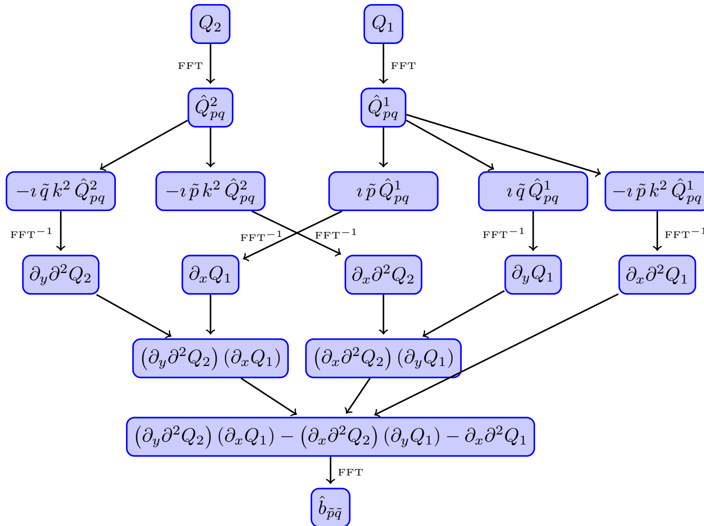


Figure 3.3.1: Flowchart to compute the r.h.s. of Equation (3.3.1).

3.4 Results

This section will present the results obtained while trying to compute the large transport coefficients associated with a hexagonal decorated basic flow and by varying ν. The hexagonal decorated basic flow is defined as

$$\begin{aligned}\Psi(x,y) = \; & -0.5\left[\cos(2.0x) + \cos\left(x+\sqrt{3}y\right) + \cos\left(x-\sqrt{3}y\right)\right] \\ & +0.5\left[\cos\left(4x+2\sqrt{3}y\right) + \cos\left(5x-\sqrt{3}y\right) + \cos\left(x-3\sqrt{3}y\right)\right] \\ & -0.5\left[\cos(4x) + \cos\left(2x+2\sqrt{3}y\right) + \cos\left(2x-2\sqrt{3}y\right)\right] \\ & +0.5\left[\cos\left(4x-2\sqrt{3}y\right) + \cos\left(5x+\sqrt{3}y\right) + \cos\left(x+3\sqrt{3}y\right)\right]\end{aligned}, \tag{3.4.1}$$

and in Figure 3.4.1 the counter contour plot is shown .

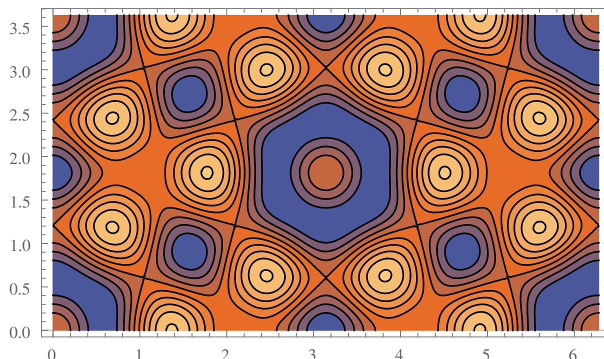


Figure 3.4.1: Hexagonal decorated flow contour plot.

In order for the work that has been developed to be validated to a certain degree, Table 3.1 presents the values for each large-scale transport coefficient and eddy viscosity given a ν, which were taken from [7]. In this way, there is a reference which can be used to compare and check if our methods are working properly.

ν	ν_e	a	c
1	1.348	1.351	0.173
0.8	0.961	-	-
0.7	0.642	1.640	0.357
0.55	-0.259	1.982	0.597

Table 3.1: Correct transport coefficients values related to different ν and associated with the hexagonal decorated basic flow.

3.4.1 Simple iterative method

The results associated with the simple iterative method are presented here. Except from when $\nu = 1$, the method diverges when the first auxiliary problem is being solved (Equation 3.0.2) and thus, there is nothing else that can be computed afterwards. The values in Table 3.2 show that independently of the problem resolution the same values are returned when $\nu = 1$. Comparing them with those presented in Table 3.1, it is also possible to conclude that the method converges to the correct solutions.

N_x	N_y	ν_{eddy}	a	c
64	64	1.348	1.352	0.173
256	256	1.348	1.352	0.173

Table 3.2: Transport coefficients computed using the simple iterative method with $\nu = 1$ and TOL $= 10^{-5}$.

The following three tables show the evolution of the maximum absolute difference between $\hat{Q}_{pq}^{n+1}$ and $\hat{Q}_{pq}^{n}$ while computing Q_1. In the first case, i.e. when $\nu = 0.8$, it seems that the method starts to converge and then diverges due to numerical instability. Whereas, in the other two, as it can be seen by analyzing Table 3.3b and Table 3.3c, the method simply diverges from the beginning. Other initial conditions, such as $\hat{Q}_{pq}^{0} = \hat{b}_{pq}$ or $\hat{Q}_{pq}^{0} = 0$, were taken into consideration but the method still did not converge.

Iteration number	$\|\|\hat{Q}_{pq}^{n+1} - \hat{Q}_{pq}^{n}\|\|_{max}$
1	0.322
2	0.207
3	0.193
4	0.124
5	0.134
10	7.254×10^{-2}
15	0.11
20	4.029×10^{-2}
40	0.213
60	1.577
80	6.063
100	13356.538

(a) $\nu = 0.8$.

Iteration number	$\|\|\hat{Q}_{pq}^{n+1} - \hat{Q}_{pq}^{n}\|\|_{max}$
1	0.332
2	0.270
3	0.255
4	0.212
5	0.244
10	0.276
15	0.819
20	0.582
40	45.464
60	210.099
80	827246.916
100	9471886348.787

(b) $\nu = 0.7$.

Iteration number	$\|\|\hat{Q}_{pq}^{n+1} - \hat{Q}_{pq}^{n}\|\|_{max}$
1	0.357
2	0.437
3	0.428
4	0.557
5	0.733
10	3.075
15	30.507
20	72.404
40	1.012×10^{6}
60	1.297×10^{10}
80	3.516×10^{14}
100	6.485×10^{20}

(c) $\nu = 0.55$.

Table 3.3: Maximum absolute difference between $\hat{Q}_{pq}^{n+1}$ and $\hat{Q}_{pq}^{n}$ evolution while computing Q_1 using the simple iterative method with $N_x = N_y = 64$ and $\text{TOL} = 10^{-5}$.

3.4.2 Second-order Runge-Kutta method

Although there is not much to point out about this method since it simply diverges in every situation, the following three tables show the behavior of the maximum absolute difference between $\hat{Q}_{pq}^{n+1}$ and $\hat{Q}_{pq}^{n}$ while computing Q_1. In all tested cases, the maximum absolute difference between $\hat{Q}_{pq}^{n+1}$ and $\hat{Q}_{pq}^{n}$ grows rapidly from the beginning, which indicates that the method is not suitable for the problem being solved (Equation (3.0.2)). The equivalent equation (3.1.7), which was used in the slaved leap-frog method, was also taken into consideration for the RK2. However, it continues to diverge from the very first iteration but in a slower pace.

Iteration number	$\|\|\hat{Q}_{pq}^{n+1} - \hat{Q}_{pq}^{n}\|\|_{max}$
1	0.379×10^{-2}
2	0.413×10^{-2}
3	0.449×10^{-2}
4	0.489×10^{-2}
5	0.402
6	0.816×10^{3}
7	0.552×10^{7}
8	0.424×10^{11}

(a) $\nu = 1$.

Iteration number	$\|\|\hat{Q}_{pq}^{n+1} - \hat{Q}_{pq}^{n}\|\|_{max}$
1	0.376×10^{-2}
2	0.403×10^{-2}
3	0.432×10^{-2}
4	0.463×10^{-2}
5	0.171
6	0.293×10^{3}
7	0.129×10^{7}
8	0.727×10^{10}

(b) $\nu = 0.8$.

Iteration number	$\|\|\hat{Q}_{pq}^{n+1} - \hat{Q}_{pq}^{n}\|\|_{max}$
1	0.375×10^{-2}
2	0.399×10^{-2}
3	0.424×10^{-2}
4	0.450×10^{-2}
5	0.103
6	0.163×10^{3}
7	0.544×10^{6}
8	0.263×10^{10}

(c) $\nu = 0.7$.

Iteration number	$\|\|\hat{Q}_{pq}^{n+1} - \hat{Q}_{pq}^{n}\|\|_{max}$
1	0.373×10^{-2}
2	0.392×10^{-2}
3	0.412×10^{-2}
4	0.432×10^{-2}
5	0.480×10^{-1}
6	0.575×10^{2}
7	0.115×10^{6}
8	0.425×10^{9}

(d) $\nu = 0.55$.

Table 3.4: Maximum absolute difference between $\hat{Q}_{pq}^{n+1}$ and $\hat{Q}_{pq}^{n}$ evolution while computing Q_1 using the RK2 method with $N_x = N_y = 64$ and $\text{TOL} = 10^{-5}$.

3.4.3 Slaved leap-frog method

The slaved leap-frog method was the only numerical method to work properly for all tested cases. The results are shown in Table 3.5a for $N_x = N_y = 64$ and in Table 3.5b for $N_x = N_y = 256$. Comparing these values with those in the reference table (Table 3.1), it is possible to conclude that the method converges to the correct solutions for each auxiliary problem that must be solved.

ν	ν_e	a	c
1	1.348	1.352	0.173
0.8	0.961	1.514	0.276
0.7	0.642	1.641	0.357
0.55	-0.259	1.982	0.596

(a) $N_x = N_y = 64$.

ν	ν_e	a	c
1	1.348	1.351	0.173
0.8	0.961	1.514	0.276
0.7	0.642	1.641	0.357
0.55	-0.259	1.982	0.596

(b) $N_x = N_y = 256$.

Table 3.5: Transport coefficients computed using the slaved leap-frog method with different resolutions and $\text{TOL} = 10^{-11}$.

3.4.4 Comparison between simple iterative method and the slaved leap-frog method

The first big difference between the simple iterative method and the slaved leap-frog method is related to their complexity. While the implementation of the simple iterative method is quite straightforward, the slaved leap-frog method requires another numerical method to be able to start. This is indeed one of the biggest drawbacks, since one has all the work of implementing a second numerical method just to compute the first time step before using the slaved leap-frog scheme. Additionally, the slaved leap-frog method for thinner resolutions, i.e. for bigger N_x and N_y, takes much more time to converge than the simple iterative method. In order to have a clearer idea, Table 3.6a and Table 3.6b present the elapsed time in seconds and the result

obtained while computing all three transport coefficients by considering different N_x and N_y. By comparing the figures of the elapsed time between the two tables it is clear that the simple iterative method increases its time approximately by a factor of 5 when the resolution is doubled, while the slaved leap-frog the factor is approximately 9. This difference is even more noticeable by looking at Figure 3.4.2 which compares the elapsed time of both methods through a plot.

Unfortunately, the simple iterative method does not work for all cases and even for $\nu = 1$ it is not very stable and for this reason TOL cannot be bigger than 10^{-5} otherwise the method will diverge. Thereby the slaved leap-frog must be used. Note that for the slaved leap-frog method to give always the same result independently of the resolution, TOL must be smaller than 10^{-5}. For instance, if TOL $= 10^{-10}$ the results for ν_{eddy}, a and c will always be the correct one even if the resolution is changed.

N_x	N_y	ν_{eddy}	a	c	Time (s)
64	64	1.348	1.352	0.173	0.177
128	128	1.348	1.352	0.173	0.867
256	256	1.348	1.352	0.173	4.534
512	512	1.348	1.352	0.173	23.352
1024	1024	1.348	1.352	0.173	119.516

(a) Simple iterative method.

N_x	N_y	ν_{eddy}	a	c	Time (s)
64	64	1.348	1.351	0.173	0.494
128	128	1.347	1.352	0.173	4.257
256	256	1.346	1.353	0.173	39.087
512	512	1.345	1.357	0.169	360.381
1024	1024	1.341	1.359	0.152	2793.566

(b) Slaved leap-frog method.

Table 3.6: Numerical solutions of the transport coefficients for $\nu = 1$ and TOL $= 10^{-5}$.

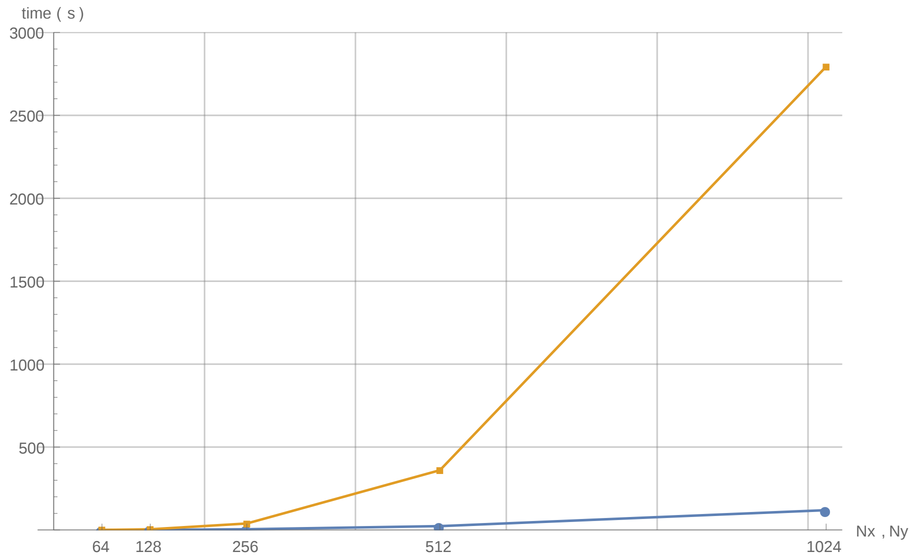


Figure 3.4.2: Simple iterative method and slaved leap-frog method execution time assuming $\nu = 1$, and the hexagonal decorated basic flow.

Chapter 4

Numerical computation of the time evolution of the vorticity of a flow

This chapter is going to present the steps that were taken to compute the time evolution of the vorticity of a flow and in the end an example is shown. The equation is the following

$$\partial_T \partial^2 \psi + a\,\mathcal{J}(\partial^2 \psi, \psi) + c\,\nabla \cdot \left((\nabla \psi) \left(\partial^2 \psi \right) \right) = -\mu_4 \partial^2 \partial^2 \psi - \mu_6 \partial^2 \partial^2 \partial^2 \psi \tag{4.0.1}$$

where the constants a, c, μ_4 and μ_6 are given. In order to read through the deduction of the previous equation see [11].

4.1 Computation of the time evolution of the vorticity of a flow

The reasoning applied in the previous chapter will be also used here in order to solve Equation (4.0.1). So firstly, the equation to be solved is written in the Fourier domain

$$-\partial_T k^2 \hat{\psi}_{pq} + \widehat{NLT}_{pq} = -\mu_4 \left(-k^2\right) \left(-k^2\right) \hat{\psi}_{pq} - \mu_6 \left(-k^2\right) \left(-k^2\right) \left(-k^2\right) \hat{\psi}_{pq}$$

where $\widehat{NLT}_{pq} = \widehat{\left(a\,\mathcal{J}(\partial^2 \psi, \psi) + c\,\nabla \cdot \left((\nabla \psi) \left(\partial^2 \psi \right) \right) \right)}_{pq}$. After some simplifications the previous equation can be written as

$$\partial_T \hat{\psi}_{pq} = \left(\mu_4 k^2 - \mu_6 k^4 \right) \hat{\psi}_{pq} + \frac{1}{k^2} \widehat{NLT}_{pq}, \tag{4.1.1}$$

by taking into account that the first Fourier coefficient corresponds to the average and thus, assuming that $\hat{\psi}_{00}(t) = 0,\ \forall t > 0$. Since the slaved leap-frog method was the only method to work properly in any case in the previous chapter, it was the chosen one to be used for this problem. As a result, the following expression appears

$$\hat{\psi}_{pq}^{n+1} = e^{2\left(\mu_4 k^2 - \mu_6 k^4\right)\Delta t} \hat{\psi}_{pq}^{n-1} + \frac{1 - e^{2\left(\mu_4 k^2 - \mu_6 k^4\right)\Delta t}}{\left(\mu_4 k^2 - \mu_6 k^4\right) k^2} \widehat{NLT}_{pq}^{\,n} \tag{4.1.2}$$

where

$$
\begin{aligned}
\widehat{NLT}^n_{pq} &= \widehat{\left(a\,\mathcal{J}(\partial^2\psi^n,\psi^n) + c\,\nabla\cdot\left((\nabla\psi^n)\left(\partial^2\psi^n\right)\right)\right)}_{pq} \\
&= a\left[\imath\tilde{p}\widehat{\left(\left(\partial^2\psi^n\right)\left(\partial_y\psi^n\right)\right)}_{pq} - \imath\tilde{q}\widehat{\left(\left(\partial^2\psi^n\right)\left(\partial_x\psi^n\right)\right)}_{pq}\right] + c\left[\imath\tilde{p}\widehat{\left(\left(\partial^2\psi^n\right)\left(\partial_x\psi^n\right)\right)}_{pq} + \imath\tilde{q}\widehat{\left(\partial^2\psi^n\right)\left(\partial_y\psi^n\right)}_{pq}\right] \\
&= \left(a\,\imath\,\tilde{p} + c\,\imath\,\tilde{q}\right)\widehat{\left(\left(\partial^2\psi^n\right)\left(\partial_y\psi^n\right)\right)}_{pq} + \left(c\,\imath\,\tilde{q} - a\,\imath\,\tilde{p}\right)\widehat{\left(\left(\partial^2\psi^n\right)\left(\partial_x\psi^n\right)\right)}_{pq}.
\end{aligned}
$$

A thorough analysis applied in Equation (4.1.2) shows that one has to be careful when $f(k) = \left(\mu_4 k^{\mathbf{2}} - \mu_6 k^4\right) k^2 = 0$. From Figure 4.1.1, it is shown that the previous equation has three roots which are $k = \pm\sqrt{\mu_4/\mu_6}$ and $k = 0$. Since $k >= 0$, only $k = 0$ and $k = \sqrt{\mu_4/\mu_6}$ must be taken into consideration. After a further analysis, it is possible to conclude that $k = \sqrt{\mu_4/\mu_6}$ is not a common situation since k only assumes a specific set of values which will unlikely contain $\sqrt{\mu_4/\mu_6}$. Therefore, only the situation $k = 0$ requires attention and it can be easily dealt with by assuming that

$$
\frac{1 - e^{2\gamma_k \Delta t}}{\gamma_k} = -2\Delta t
$$

when $k = 0$. It is easy to prove the previous statement by keeping in mind the exponential expansion:

$$
\frac{1 - e^{2\gamma_k \Delta t}}{\gamma_k} = \frac{1 - (1 + 2\gamma_k \Delta t + O(\gamma_k^2))}{\gamma_k} = -2\Delta t,
$$

where $\gamma_k = \mu_4 k^{\mathbf{2}} - \mu_6 k^4$.

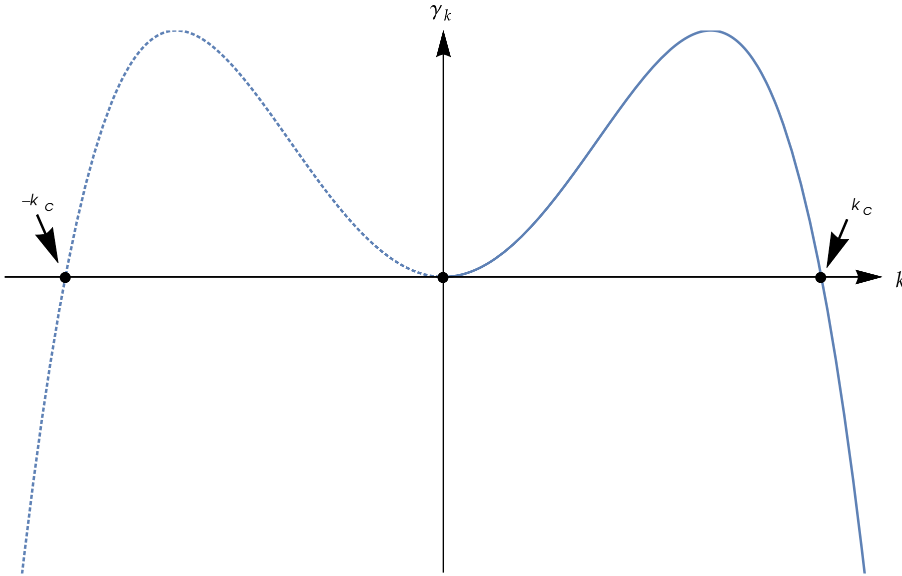


Figure 4.1.1: Function $f(k)$ plot with its roots where $k_c = \sqrt{\mu_4/\mu_6}$.

The resulting algorithm is shown in Figure 4.1.2. As input, it is necessary to give to the algorithm a random periodic flow in the Fourier domains, $\hat{\psi}^0_{pq}$, and the constants a, c, μ_4 and μ_6. Then, the first two steps are responsible to compute the constant of the linear term and the constant of the non-linear term associated with the (p, q) Fourier coefficient. Afterwards, all constants whose Fourier coefficient belong to the truncation area are set to zero. This way whenever ψ^{n+1} is computed the dealiasing is applied implicitly. In the fourth step, ψ^1 is computed by calling the RK2 method so that the slaved leap-frog method has everything it needs

to start. Thereupon, the modified salved leap-frog method comes into action to compute the time evolution of the vorticity of the flow.

The algorithm used to compute $\widehat{NLT}_{pq}$ is illustrated in Figure 4.1.3. Firstly, it is applied a forward Fourier transform to ψ^n so that the partial derivates and the Laplacian can be calculated in the third step. Like it was seen in other problems, it is required to go back to the physical domain in order to compute the multiplications. After computing the multiplications a forward Fourier is applied to each sub-result, and at the end, they are added up as shown in the last step.

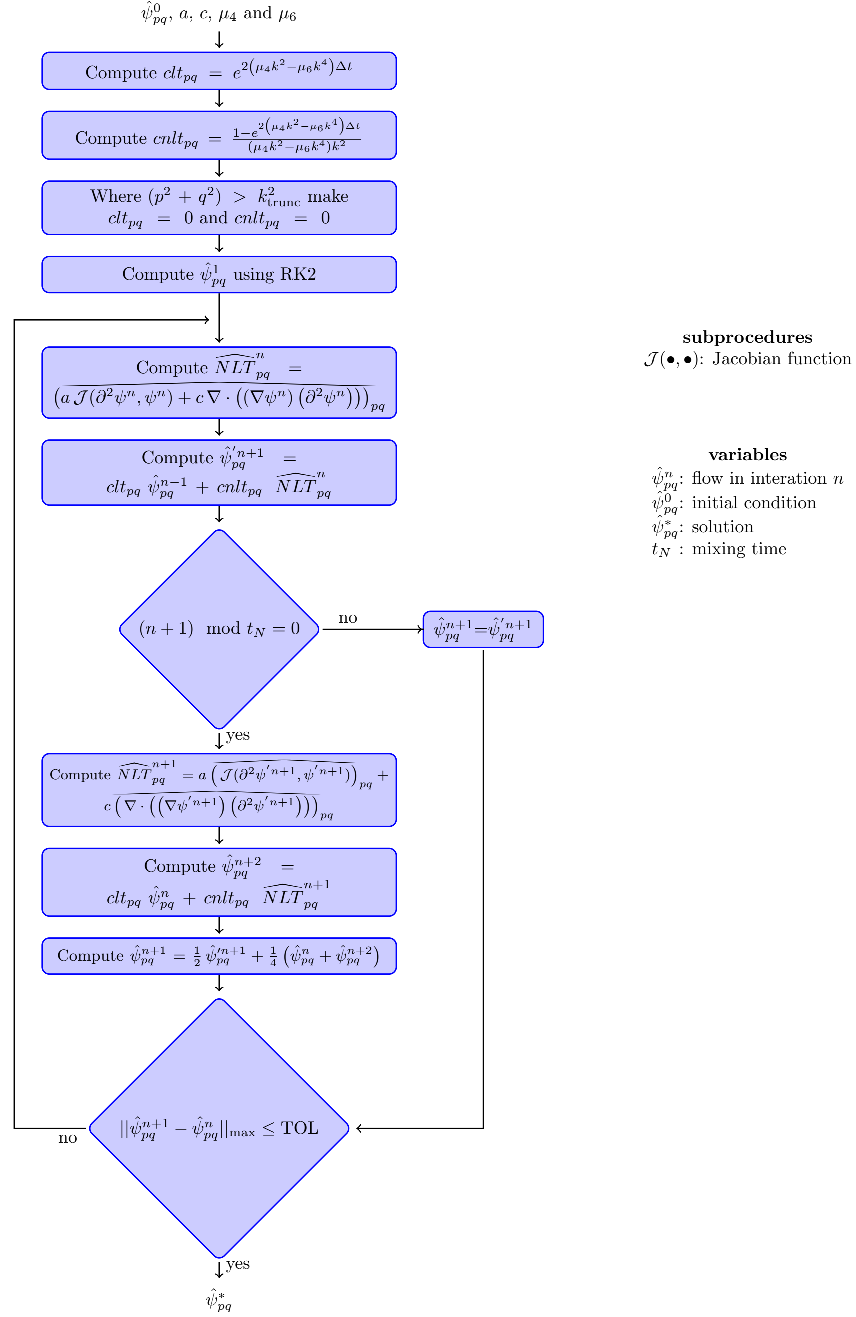


Figure 4.1.2: Modified slaved leap-frog method flowchart for the computation of the time evolution of the vorticity of a flow.

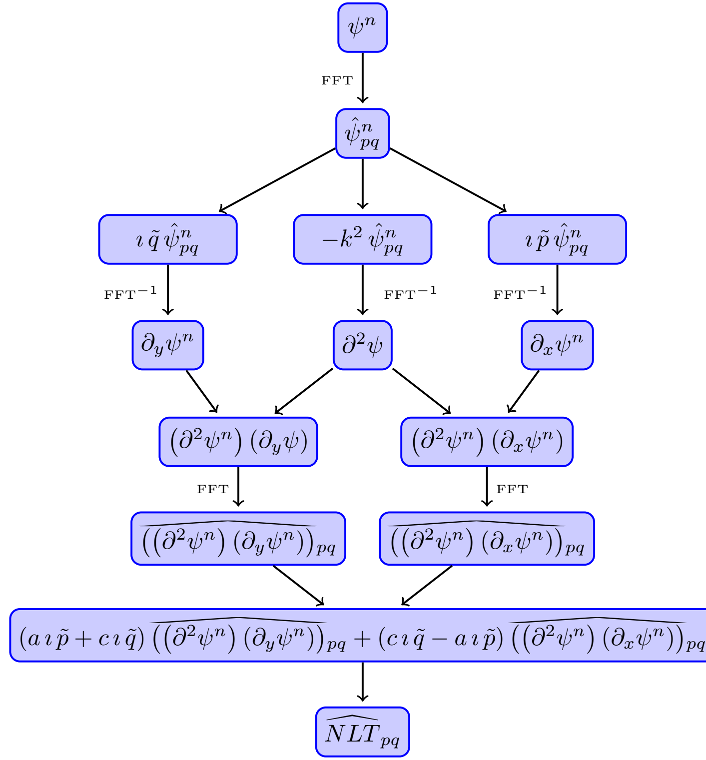


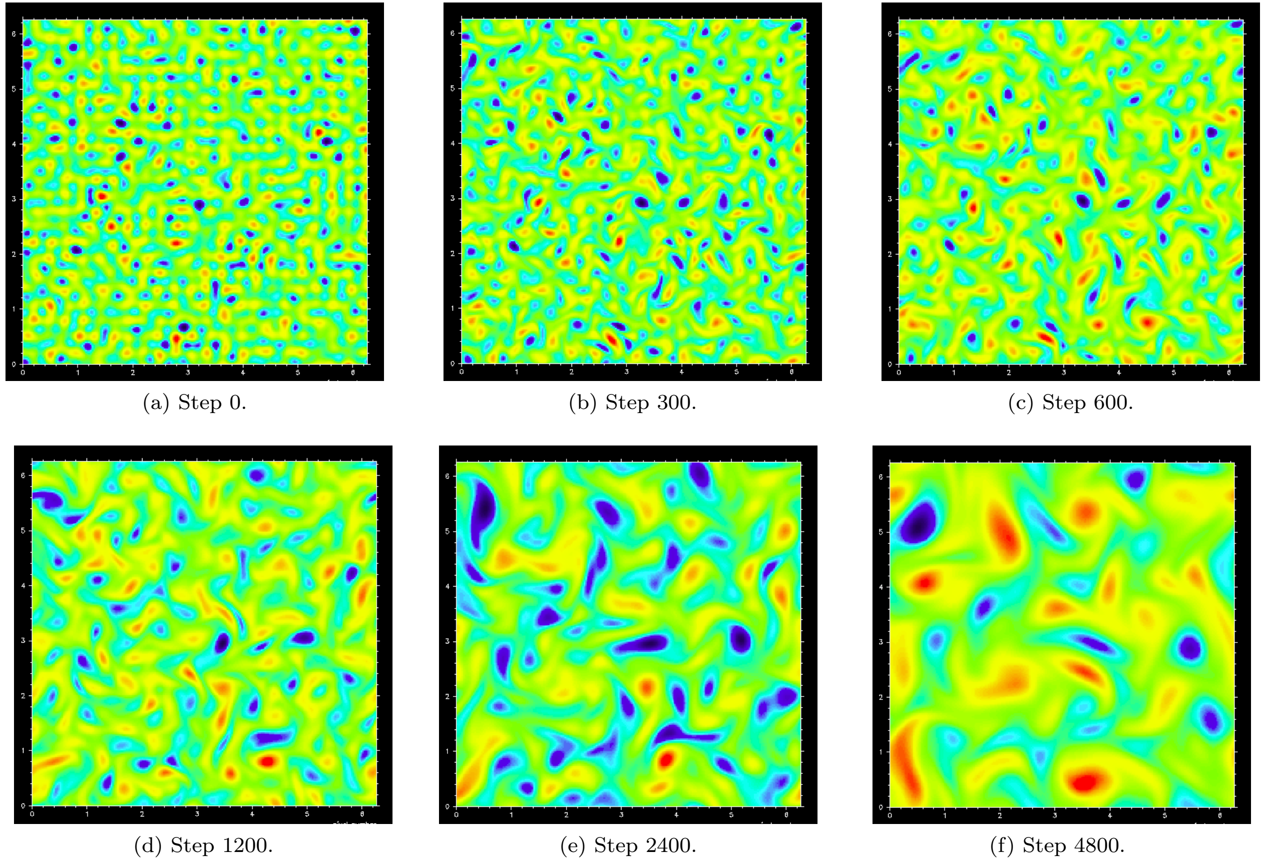
Figure 4.1.3: Algorithm flowchart to compute the NLT.

4.2 Results

This section presents a small example of the time evolution of the vorticity associated with ψ. The equation that was taken into consideration was

$$\partial_T\partial^2\psi+\mathcal{J}(\partial^2\psi,\psi)=0.05\,\partial^2\partial^2\psi$$

with $N_x=N_y=256$ and where ψ^0 is a random periodic flow. Figure 4.2.1 shows a sequence of images related to the time evolution of the vorticity. The first image presents the initial vorticity. The second and third one are the vorticity after 300 and 600 iterations, respectively. The last three are the results obtained after 1200, 2400 and 4800 iterations.



(a) Step 0. (b) Step 300. (c) Step 600.

(d) Step 1200. (e) Step 2400. (f) Step 4800.

Figure 4.2.1: Plot of the evolution over time of $\partial^2\psi$ for $N_x = N_y = 256$, $a = 1$, $c = 0$, $\mu_4 = -0.05$ and $\mu_6 = 0$ with a random periodic initial condition ψ^0.

Chapter 5

Conclusion and future work

As a result of the work that has been undertaken, a software package was developed that computes the forward and backward Fourier transform, partial derivatives and the Laplacian in the Fourier domain, eddy viscosities, large-scale coefficients related to the two-dimensional incompressible Navier-Stokes equations and the time evolution of the vorticity of a flow. Three different approaches were studied as a means of computing the transport coefficients, namely the simple iterative method, the 2nd-order Runge-Kutta method and the slaved leap-frog method. The advantages of the former method is its simplicity and its converging speed which is considerably faster compared to the slaved leap-frog method for big N_x and N_y. However, it fails to converge when $\nu < 1$. The second method diverges for any value of ν, but it is still useful since it fulfills an important role before the slaved leap-frog method is able to come into action. Although the last method is the most complex one, it was the only one that worked properly in all tested cases. Thus, it was also the method chosen to compute the time evolution of a two-dimensional Navier-Stokes flow in the large scales.

It is planned, as future work, to keep working on the development of this computational tool by adding more features to the package in order to turn this computational tool more complete.

Bibliography

[1] Downloading FFTW. Website: http://www.fftw.org/download.html.

[2] M. Frigo and S. Johnson. FFTW (version 3.3.4). Website: http://www.fftw.org/fftw3.pdf.

[3] M. Chaves and S. Gama. *Time Evolution of the Eddy Viscosity in Two-Dimensional Navier-Stokes Flow*, published in Phys. Rev ., 2000, No. 61 , pp. 2118–2120.

[4] W. Press, S. Teukolsky, W. Vetterling and B. Flannery. *Numerical Recipes in C - The Art of Scientific Computing.* Cambridge University Press, 2nd edition, 1992.

[5] U. Frisch, Z. S. She and O. Thual. *Elasticity of Flame Fronts and Numerical Experiments,* published in Numerical Simulation of Combustion Phenomena Lecture Notes in Physiscs, 1985, No. 241, pp. 234-23.

[6] R. Peyret. *Spectral Methods for Incompressible Viscous Flow.* Springer, 2002.

[7] S. Gama. Personal communication.

[8] W. E. Boyce and R. C. DiPrima. *Elementary Differential Equations and Boundary Problems.* John Willey & Sons, 10th edition, 2012.

[9] D. Gottlieb, and S. A. Orszag. *Numerical Analysis of Spectral Methods* (SIAM, Philadelphia, PA, 1977).

[10] L. Prandlt, Z. Angew. Math Mech. 5, 136 (1925).

[11] S. Gama, M. Vergassola, and U. Frisch. *Negative eddy-viscosity in isotropically forced two-dimensional flows: linear and nonlinear dynamics*, published in Journal of Fluid Mechanics, 1994, No. 260, pp. 95–126.

Appendix A

Appendix - Discrete Orthogonality Relation

Consider the following discrete orthogonality relation:

$$\sum_{\alpha=1}^{N} e^{\imath(k-l)\,2\pi\alpha/N} = \begin{cases} N & \text{if } k-l = mN,\ m \in \mathbb{Z} \\ 0 & \text{if } k-l \neq mN,\ m \in \mathbb{Z} \end{cases}.$$

In order to prove its correctness, it is necessary to take into consideration two cases which are when $k-l$ is a multiple of N and when it is not. By assuming that $k-l = m\,N$, $m \in \mathbb{Z}$, then

$$\sum_{\alpha=1}^{N} e^{\imath\, m\, N\, 2\pi\alpha/N} = \sum_{\alpha=1}^{N} \left(e^{\imath\, m\, 2\pi}\right)^{\alpha} = \sum_{\alpha=1}^{N} \left(\cos\left(m\, 2\pi\right) + \imath \sin\left(m\, 2\pi\right)\right)^{\alpha} = N.$$

On the other hand, if $k-l \neq m\,N$, $m \in \mathbb{Z}$ then

$$\sum_{\alpha=1}^{N} \left(e^{\imath(k-l)\,2\pi/N}\right)^{\alpha} = \frac{e^{\imath(k-l)\,2\pi/N} - e^{\imath(k-l)\,2\pi(N+1)/N}}{1 - e^{\imath(k-l)\,2\pi/N}},$$

by recalling the geometric series:

$$\sum_{\alpha=1}^{N} r^{\alpha} = \begin{cases} \frac{r-r^{N+1}}{1-r} & \text{if } r \neq 1 \\ N & \text{if } r = 1 \end{cases}.$$

Now, it is only necessary to do the arithmetics and remember that $(k-l) \in \mathbb{Z}$:

$$\frac{e^{\imath(k-l)\,2\pi/N} - e^{\imath(k-l)\,2\pi(N+1)/N}}{1 - e^{\imath(k-l)\,2\pi/N}} = \frac{e^{\imath(k-l)\,2\pi/N} - e^{\imath(k-l)\,2\pi} e^{\imath(k-l)\,2\pi/N}}{1 - e^{\imath(k-l)\,2\pi/N}} = 0.$$

Appendix B

Appendix - FFT

B.1 Subroutine my_fft2d_setup

```
subroutine my_fft2d_setup(fft2d_dir_plan,fft2d_inv_plan, u, uhat, Nx, Ny)
  implicit none

  integer, intent(in) :: Nx, Ny
  real(8), dimension(0:Nx-1, 0:Ny-1), intent(in) :: u
  complex(8), dimension(0:Nx/2, 0:Ny-1), intent(in) :: uhat
  integer(8), intent(out) :: fft2d_dir_plan, fft2d_inv_plan

  call dfftw_plan_dft_r2c_2d(fft2d_dir_plan, Nx, Ny, u, uhat, 64) !fftw_estimate = 64
  call dfftw_plan_dft_c2r_2d(fft2d_inv_plan, Nx, Ny, uhat, u, 64) !fftw_estimate = 64

  return
end subroutine my_fft2d_setup
```

B.2 Subroutine my_fft2d_close

```
subroutine my_fft2d_close(fft2d_dir_plan,fft2d_inv_plan)
  implicit none

  integer(8), intent(inout) :: fft2d_dir_plan, fft2d_inv_plan

  call dfftw_destroy_plan(fft2d_dir_plan)
  call dfftw_destroy_plan(fft2d_inv_plan)

  return
end subroutine my_fft2d_close
```

B.3 Subroutine my_fft2d_dir

```
subroutine my_fft2d_dir(uhat, u, plan, Nx, Ny)
  implicit none

  complex(8), dimension(0:Nx/2, 0:Ny-1), intent(out) :: uhat
```

```
  real(8),    dimension(0:Nx-1, 0:Ny-1), intent(in)  :: u
  integer(8), intent(in) :: plan
  integer,    intent(in) :: Nx, Ny

  real(8),    dimension(0:Nx-1, 0:Ny-1)  :: u_bk   real(8) :: dfloatNxNy

  u_bk = u
  call dfftw_execute_dft_r2c(plan, u_bk, uhat)
  dfloatNxNy = dfloat(Nx*Ny)
  uhat = uhat/dfloatNxNy !just to normalize our values

  return
end subroutine my_fft2d_dir
```

B.4 Subroutine my_fft2d_inv

```
subroutine my_fft2d_inv(u, uhat, plan, Nx, Ny)
  implicit none

  real(8),    dimension(0:Nx-1, 0:Ny-1), intent(out) :: u
  complex(8), dimension(0:Nx/2, 0:Ny-1), intent(in)  :: uhat
  integer(8), intent(in) :: plan
  integer,    intent(in) :: Nx, Ny

  complex(8), dimension(0:Nx/2, 0:Ny-1) :: uhat_bk
  uhat_bk = uhat
  call dfftw_execute_dft_c2r(plan, uhat_bk, u)

  return
end subroutine my_fft2d_inv
```

Appendix C

Appendix - Differential Operators in Fourier domain

C.1 Subroutine my_derop_setup

```
subroutine my_derop_setup(kx, ky, k2, k4, Lx, Ly, pmax, qmax)
  implicit none

  real(8), parameter :: TWO_PI = 8.d0 * datan(1.d0)
  integer, intent(in)  :: pmax, qmax
  real(8), intent(in) :: Lx, Ly
  real(8), dimension(0:pmax,0:2*qmax-1), intent(out) :: k2, k4
  real(8), dimension(0:2*qmax-1),        intent(out) :: ky
  real(8), dimension(0:pmax),            intent(out) :: kx

  integer :: p, q   integer :: Ny
  real(8) :: wx, wy

  Ny = 2*qmax
  wx = TWO_PI / Lx
  wy = TWO_PI / Ly

  forall(p=0:pmax) kx(p) = wx*dfloat(p)
  forall(q=0:qmax) ky(q) = wy*dfloat(q)
  forall(q=qmax+1:2*qmax-1) ky(q) = wy*dfloat(q-Ny)

  do p = 0, pmax
    do q = 0, 2*qmax-1
      k2(p,q) = kx(p)**2 + ky(q)**2
    end do
  end do
  k4 = k2**2

  return
end subroutine my_derop_setup
```

C.2 Subroutine der_in_x_hat

```
subroutine der_in_x_hat(dudx_hat, uhat, kx, pmax, qmax)
```

```
  implicit none

  integer, intent(in) :: pmax, qmax
  real(8), dimension(0:pmax), intent(in) :: kx
  complex(8), dimension(0:pmax, 0:2*qmax-1), intent(in)  :: uhat
  complex(8), dimension(0:pmax, 0:2*qmax-1), intent(out) :: dudx_hat
  integer :: q

  forall(q=0:2*qmax-1)
    dudx_hat(:,q) = (0.d0,1.d0) * kx(:) * uhat(:,q)

  return
end subroutine der_in_x_hat
```

C.3 Subroutine der_in_y_hat

```
subroutine der_in_y_hat(dudy_hat, uhat, ky, pmax, qmax)
  implicit none

  integer, intent(in) :: pmax, qmax
  real(8), dimension(0:2*qmax-1), intent(in) :: ky
  complex(8), dimension(0:pmax, 0:2*qmax-1), intent(in)  :: uhat
  complex(8), dimension(0:pmax, 0:2*qmax-1), intent(out) :: dudy_hat

  integer :: p

  forall(p=0:pmax)
    dudy_hat(p,:) = (0.d0,1.d0) * ky(:) * uhat(p,:)

  return
end subroutine der_in_y_hat
```

C.4 Subroutine lap_hat

```
subroutine lap_hat(lap_uhat, uhat, k2, pmax, qmax)
  implicit none

  integer, intent(in) :: pmax, qmax
  real(8),    dimension(0:pmax, 0:2*qmax-1), intent(in)  :: k2
  complex(8), dimension(0:pmax, 0:2*qmax-1), intent(in)  :: uhat
  complex(8), dimension(0:pmax, 0:2*qmax-1), intent(out) :: lap_uhat

  lap_uhat =  - k2 * uhat

  return
end subroutine lap_hat
```

C.5 Subroutine bilap_hat

```
subroutine bilap_hat(bilap_uhat, uhat, k4, pmax, qmax)
  implicit none

  integer, intent(in) :: pmax, qmax
  real(8),     dimension(0:pmax, 0:2*qmax-1), intent(in)  :: k4
  complex(8), dimension(0:pmax, 0:2*qmax-1), intent(in)  :: uhat
  complex(8), dimension(0:pmax, 0:2*qmax-1), intent(out) :: bilap_uhat

  bilap_uhat = k4 * uhat

  return
end subroutine bilap_hat
```

C.6 Subroutine J_hat

```
subroutine J_hat(output_hat, fhat, ghat, kx, ky, fft2d_dir_plan, fft2d_inv_plan,
  Nx, Ny)
  implicit none

  integer,     intent(in) :: Nx, Ny
  integer(8), intent(in) :: fft2d_dir_plan, fft2d_inv_plan
  real(8), dimension(0:Ny-1), intent(in) :: ky
  real(8), dimension(0:Nx/2), intent(in) :: kx
  complex(8), dimension(0:Nx/2, 0:Ny-1), intent(in)  :: fhat, ghat
  complex(8), dimension(0:Nx/2, 0:Ny-1), intent(out) :: output_hat

  integer :: pmax, qmax
  real(8),     dimension(0:Nx-1, 0:Ny-1) :: aux1, aux2, f, dgdx, dgdy
  complex(8), dimension(0:Nx/2, 0:Ny-1) :: aux1_hat, aux2_hat, aux3_hat, aux4_hat
  complex(8), dimension(0:Nx/2, 0:Ny-1) :: dgdx_hat, dgdy_hat

  pmax = Nx/2
  qmax = Ny/2

  call der_in_y_hat(dgdy_hat, ghat, ky, pmax, qmax)
  call der_in_x_hat(dgdx_hat, ghat, kx, pmax, qmax)

  call my_fft2d_inv(dgdy, dgdy_hat, fft2d_inv_plan, Nx, Ny)
  call my_fft2d_inv(dgdx, dgdx_hat, fft2d_inv_plan, Nx, Ny)

  call my_fft2d_inv(f, fhat, fft2d_inv_plan, Nx, Ny)

  aux1 = f * dgdy
  aux2 = f * dgdx

  call my_fft2d_dir(aux1_hat, aux1, fft2d_dir_plan, Nx, Ny)
  call my_fft2d_dir(aux2_hat, aux2, fft2d_dir_plan, Nx, Ny)

  call der_in_x_hat(aux3_hat, aux1_hat, kx, pmax, qmax)
  call der_in_y_hat(aux4_hat, aux2_hat, ky, pmax, qmax)

  output_hat = aux3_hat - aux4_hat
```

```
   return
end subroutine J_hat
```

Appendix D

Appendix - Numerical methods

D.1 Subroutine iterative_method_hat

```
subroutine iterative_method_hat(fhat, bhat, BF_hat, kx, ky, k2, k4, visc,
  fft2d_dir_plan, fft2d_inv_plan, Nx, Ny, TOL_iter)
  implicit none

  include 'interfaces_my_derop.f95'

  real(8),     intent(in) :: TOL_iter
  integer,     intent(in) :: Nx, Ny
  integer(8),  intent(in) :: fft2d_dir_plan, fft2d_inv_plan
  real(8),     intent(in) :: visc
  real(8),     dimension(0:Nx/2, 0:Ny-1), intent(in)  :: k2, k4
  real(8),     dimension(0:Ny-1), intent(in)          :: ky
  real(8),     dimension(0:Nx/2), intent(in)     :: kx
  complex(8),  dimension(0:Nx/2, 0:Ny-1), intent(in)  :: BF_hat, bhat
  complex(8),  dimension(0:Nx/2, 0:Ny-1), intent(out) :: fhat

  integer, parameter  :: mixing_time = 19
  integer :: p, q, pmax, qmax
  integer :: counter
  integer :: ktrunc2, k2_linha
  real(8) :: max_error, maxvalue
  real(8),     dimension(0:Nx/2, 0:Ny-1) :: one_over_visc_k4
  complex(8),  dimension(0:Nx/2, 0:Ny-1) :: fhat_old, lap_fhat
  complex(8),  dimension(0:Nx/2, 0:Ny-1) :: aux1_hat, aux2_hat, lap_BF_hat

  pmax = Nx/2    qmax = Ny/2

  call lap_hat(lap_BF_hat, BF_hat, k2, pmax, qmax)

  one_over_visc_k4 = visc * k4
  ! avoiding division by zero
  one_over_visc_k4(0,0) = 1.d0
  one_over_visc_k4      = 1.d0 / one_over_visc_k4
  one_over_visc_k4(0,0) = 0.d0
  !set everything that is outside the truncation limit to zero
  !for apply dealiasing
  ktrunc2 =  (min(Nx, Ny)/3.d0)**2
  do p = 0, pmax
```

```
    do q = 0, 2*qmax-1
          if(q <= qmax) then
        k2_linha = p**2 + q**2
      else
        k2_linha = p**2 + (q-Ny)**2
          end if
      if (k2_linha > ktrunc2) one_over_visc_k4(p,q) = 0.d0
    end do
  end do

  ! initial condition
  fhat = BF_hat

  max_error = 1.D10
  ! count number of iterations
  counter   = 0
  do while (max_error > TOL_iter)
    fhat_old = fhat
    call lap_hat(lap_fhat, fhat, k2, pmax, qmax)
    call J_hat(aux1_hat, lap_fhat, BF_hat , kx, ky, fft2d_dir_plan,&
      fft2d_inv_plan, Nx, Ny)
      call J_hat(aux2_hat, lap_BF_hat , fhat, kx, ky, fft2d_dir_plan,&
      fft2d_inv_plan, Nx, Ny)

    ! since we set to zero all values that are outside truncation
    ! limit in one_over_visc_k4 by doing this multiplication we
    ! are also applying dealiasing
      fhat     = one_over_visc_k4 * (aux1_hat + aux2_hat - bhat)

    counter = counter+1
      if(mod(counter, mixing_time)==0) then
      call lap_hat(lap_fhat, fhat, k2, pmax, qmax)
      ! aux1_hat: J_hat (from sheet)
      ! aux2_hat: G_hat (from sheet)
      call J_hat(aux1_hat, lap_fhat, BF_hat , kx, ky, fft2d_dir_plan,&
        fft2d_inv_plan, Nx, Ny)
      call J_hat(aux2_hat, lap_BF_hat , fhat, kx, ky, fft2d_dir_plan,&
        fft2d_inv_plan, Nx, Ny)

      ! since we set to zero all values that are outside truncation limit in
          ! a1 and a2 by doing this multiplication we are also applying dealiasing
        aux1_hat = one_over_visc_k4 * (aux1_hat + aux2_hat - bhat)
      fhat  = 0.5d0 * fhat + 0.25 * (fhat_old + aux1_hat)
    end if

    !compute max difference between the updated value and the previous one
    max_error = maxval(cdabs(fhat-fhat_old))
    if ((mod(counter,1)==0 .or. counter==1)) print*,"Iter number: ",counter, &
      "Max error:",max_error,maxval(cdabs(fhat)
  end do
end subroutine iterative_method_hat
```

D.2 Subroutine rk2_hat

```
subroutine rk2_hat(psi_hat, bhat, BF_hat, lap_BF_hat, kx, ky, k2, k4, visc,
```

```
  fft2d_dir_plan, fft2d_inv_plan, Nx, Ny, dt)
  implicit none

  integer,    intent(in) :: Nx, Ny
  integer(8), intent(in) :: fft2d_dir_plan, fft2d_inv_plan
  real(8),    intent(in) :: visc, dt
  real(8),    dimension(0:Nx/2, 0:Ny-1), intent(in)    :: k2, k4
  real(8),    dimension(0:Ny-1), intent(in)            :: ky
  real(8),    dimension(0:Nx/2), intent(in)            :: kx
  complex(8), dimension(0:Nx/2, 0:Ny-1), intent(in)    :: BF_hat, lap_BF_hat, bhat
  complex(8), dimension(0:Nx/2, 0:Ny-1), intent(inout) :: psi_hat

  complex(8), dimension(0:Nx/2, 0:Ny-1) :: kapaRK_hat, Fhat

  ! compute F(psi_hat)
  call F_function_hat(Fhat, psi_hat, bhat, BF_hat, lap_BF_hat, kx, ky, k2, k4,
    visc, fft2d_dir_plan, fft2d_inv_plan, Nx, Ny)

  ! compute K1 for RK2
  kapaRK_hat = dt*Fhat

  ! compute F(psi_linha_hat)
  call F_function_hat(Fhat, psi_hat+0.5d0*kapaRK_hat, bhat, BF_hat, lap_BF_hat,
    kx, ky, k2, k4, visc, fft2d_dir_plan, fft2d_inv_plan, Nx, Ny)

  ! compute K2 for RK2
  kapaRK_hat = dt*Fhat

  psi_hat    = psi_hat + kapaRK_hat

  return
end subroutine rk2_hat
```

D.3 Subroutine slaved_leap_frog

```
subroutine slaved_leap_frog (psi_hat, bhat, BF_hat, kx, ky, k2, k4, visc, &
  fft2d_dir_plan, fft2d_inv_plan, Nx, Ny, TOL_iter, dt)
  implicit none
  include 'interfaces_my_derop.f95'

  real(8),    intent(in) :: TOL_iter, dt
  integer,    intent(in) :: Nx, Ny
  integer(8), intent(in) :: fft2d_dir_plan, fft2d_inv_plan
  real(8),    intent(in) :: visc
  real(8),    dimension(0:Nx/2, 0:Ny-1), intent(in)  :: k2, k4
  real(8),    dimension(0:Ny-1),         intent(in)  :: ky
  real(8),    dimension(0:Nx/2),         intent(in)  :: kx
  complex(8), dimension(0:Nx/2, 0:Ny-1), intent(in)  :: BF_hat, bhat
  complex(8), dimension(0:Nx/2, 0:Ny-1), intent(out) :: psi_hat

  integer, parameter :: mixing_time = 19
  integer :: pmax, qmax
  integer :: counter
  real(8) :: max_error
  real(8),    dimension(0:Nx/2, 0:Ny-1) :: a1, a2 !(see notebook for their meaning)
```

```
complex(8), dimension(0:Nx/2, 0:Ny-1) :: lap_BF_hat, psi_hat_old
complex(8), dimension(0:Nx/2, 0:Ny-1) :: aux1_hat, aux2_hat, lap_psi_hat
integer :: ktrunc2, p, q, k2_linha

pmax = Nx/2    qmax = Ny/2

! initial condition
psi_hat = dcmplx(0.d0,0.d0)

! compute all constants
call lap_hat(lap_BF_hat, BF_hat, k2, pmax, qmax)
a1      = dexp(-2.d0 * visc * dt * k2)    a1(0,0) = 0.d0
a2      = visc * k4
! avoiding division by zero
a2(0,0) = 1.d0
a2      = (1.d0 - a1) / a2
a2(0,0) = 0.d0

! set everything that is outside the truncation limit to zero for apply dealiasing
ktrunc2 =  (min(Nx, Ny)/3.d0)**2
do p = 0, pmax
  do q = 0, 2*qmax-1
        if(q <= qmax) then
      k2_linha = p**2 + q**2
        else
      k2_linha = p**2 + (q-Ny)**2
        end if

       if (k2_linha > ktrunc2) then
           a1(p,q) = 0.d0
           a2(p,q) = 0.d0
    end if
  end do
end do

! compute one time step
psi_hat_old = psi_hat
call rk2_hat(psi_hat, bhat, BF_hat, lap_BF_hat, kx, ky, k2, k4, visc,
  fft2d_dir_plan, fft2d_inv_plan, Nx, Ny, dt)

max_error = 1.D10
! count number of iterations
counter   = 0
do while (max_error > TOL_iter)
  ! slaved leapfrog scheme
  call lap_hat(lap_psi_hat, psi_hat, k2, pmax, qmax)

  ! aux1_hat: J_hat (from sheet)
  ! aux2_hat: G_hat (from sheet)
  call J_hat(aux1_hat, lap_psi_hat, BF_hat , kx, ky, fft2d_dir_plan,
    fft2d_inv_plan, Nx, Ny)
  call J_hat(aux2_hat, lap_BF_hat , psi_hat, kx, ky, fft2d_dir_plan,
    fft2d_inv_plan, Nx, Ny)

  ! since we set to zero all values that are outside truncation limit in
```

```
    ! a1 and a2 by doing this multiplication we are also applying dealiasing
    aux1_hat = a1 * psi_hat_old + a2 * (aux1_hat + aux2_hat - bhat)

    psi_hat_old = psi_hat
    psi_hat     = aux1_hat

    counter  = counter+1

    if(mod(counter, mixing_time)==0) then
      call lap_hat(lap_psi_hat, psi_hat, k2, pmax, qmax)

      ! aux1_hat: J_hat (from sheet)
      ! aux2_hat: G_hat (from sheet)
      call J_hat(aux1_hat, lap_psi_hat, BF_hat , kx, ky, fft2d_dir_plan,
        fft2d_inv_plan, Nx, Ny)
      call J_hat(aux2_hat, lap_BF_hat , psi_hat, kx, ky, fft2d_dir_plan,
        fft2d_inv_plan, Nx, Ny)

      ! since we set to zero all values that are outside truncation limit in
      ! a1 and a2 by doing this multiplication we are also applying dealiasing
      aux1_hat = a1 * psi_hat_old + a2 * (aux1_hat + aux2_hat - bhat)
          psi_hat  = 0.5d0 * psi_hat + 0.25 * (psi_hat_old + aux1_hat)
    end if

    ! compute max difference between the updated value and the previous one
    max_error = maxval(cdabs(psi_hat-psi_hat_old))
    if(mod(counter,500) == 0 .or. counter==1)
      print "(a,i6,a,e8.3,a,f20.16,f20.16)","Iter: ",counter," Max error: ",
        max_error,"    ",maxval(cdabs(psi_hat))
  end do

  return
end subroutine slaved_leap_frog
```