Trym Haddal

# Modelling Near Surface Turbulence with Helmholtz-Onsager Vortices

Master's thesis in Mechanical Engineering
Supervisor: Simen Andreas Ådnøy Ellingsen
June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Energy and Process Engineering

NTNU
Kunnskap for en bedre verden

Trym Haddal

# Modelling Near Surface Turbulence with Helmholtz-Onsager Vortices

Master's thesis in Mechanical Engineering
Supervisor: Simen Andreas Ådnøy Ellingsen
June 2021

Norwegian University of Science and Technology
Faculty of Engineering
Department of Energy and Process Engineering

NTNU
Norwegian University of
Science and Technology

# Abstract

The vorticity in near-surface turbulent currents constantly deforms the ocean surface. This phenomenon is important to understand because the surface roughness determines the amount of gas transfer from the ocean to the atmosphere, and is therefore vital to global warming. The sub-surface turbulence is represented as a set of discrete eddies using the Helmholtz-Onsager vortex turbulence model, in order to further understand the interaction, and to investigate how the system scales with eddy parameters.

The joint Probability Distribution Function (PDF) for the depth and impulse vector of an eddy interacting with a free surface was found using a Markov Chain Monte Carlo (MCMC) algorithm. Eddies were subsequently drawn from this distribution and the surface deformation was ensemble averaged to investigate the surface statistics. Nine different simulations were run with different combination of Bond numbers and turbulent Froude numbers.

It was found that the PDF of an eddy is affected by two competing effects for minimizing the system energy. On the one hand, the surface repels the eddies due to the energy required to deform it. On the other hand, the eddy is attracted to the surface because it can cancel more of its velocity field in closer proximity, given that its impulse vector is oriented normal to the surface. This resulted in a zone of zero probability near the surface, followed by sharp peaks in the PDFs where the two effects are in equilibrium. The strength and depth of the peaks depend on the Froude- and Bond numbers, which control the relative strength of the two effects.

Convergence issues were encountered during the surface statistics analysis because of the steep gradients above the aforementioned peaks in the PDFs. The result of this analysis is therefore inconclusive. However, it was speculated that the surface elevation distribution was negatively skewed because of the vertical eddy impulse preference found for weak turbulence, and that the higher corresponding excess kurtosis signify a less random surface due to the increased anisotropy of the eddy impulse vectors.

# Sammendrag

Vortisiteten i turbulente strømmer under en fri havoverflate fornyer konstant overflaten. Dette fenomenet er viktig å forstå fordi ruheten til overflaten bestemmer fluks av gasser fra havet til atmosfæren, og er derfor viktig for global oppvarming. Turbulensen er modellert som et sett med diskrete virvler ved hjelp av Helmholtz-Onsager vortex modellen, for å forstå interaksjonen ytterligere, og for å undersøke hvordan systemet skalerer med virvelparametere.

Simultanfordelingen for dybde- og impulsvektoren til en virvel som vekselvirker med en fri overflate ble funnet ved hjelp av en MCMC algoritme. Virvler ble deretter hentet fra denne fordelingen, og overflatedeformasjonen ble beregnet for å undersøke overflatestatistikken. Ni forskjellige simuleringer ble kjørt med forskjellige kombinasjoner av Bond tall og turbulente Froude tall.

Det ble funnet at sannsynlighetsfordelingen for en virvel påvirkes av to konkurrerende effekter for å minimere systemets energi. På den ene siden dytter overflaten virvler vekk fra seg selv på grunn av energien som kreves for deformasjon. På den andre siden tiltrekkes virvelen mot overflaten fordi den kan kansellere mer av hastighetsfeltet sitt der, gitt at impulsvektoren er orientert normalt mot overflaten. Dette resulterte i en sone med null sannsynlighet nær overflaten, etterfulgt av skarpe topper i sannsynlighetsfordelingene der de to effektene er i likevekt. Styrken og dybden av toppene avhenger av Froude- og Bond tallene som styrer den relative styrken til de to effektene.

Konvergensproblemer oppsto under overflatestatistikkanalysen på grunn av de bratte gradientene over toppene i sansynlighetsfordelingene. Resultatene av denne analysen er derfor inkonklusiv. Likevel ble det spekulert i at fordelingen av overflatehøyde var negativt grunnet den vertikale virvelimpuls preferansen som ble funnet for svak turbulens, og at den høyere tilsvarende overflødige kurtosen betegner en mindre tilfeldig overflate på grunn av den økte anisotropien til virvelimpulsvektorene.

# Acknowledgements

I would like to thank my supervisor, *Simen Ellingsen* for excellent guidance throughout the year by means of engaging meetings and e-mail correspondence. Additionally I would like to thank the wave-current interaction research group at the department of Energy and Process Engineering for helpful feedback during the bi-weekly meetings.

# Contents

**Abstract** . . . iii
**Sammendrag** . . . iv
**Acknowledgements** . . . v
**Contents** . . . vi
**Figures** . . . viii
**Tables** . . . x
**Nomenclature** . . . xi
**Acronyms** . . . xiv
**1 Introduction** . . . **1**
1.1 Motivation . . . 1
1.2 Literary Review . . . 2
1.3 Scope . . . 2
1.4 Outline . . . 3
**2 Theory** . . . **4**
2.1 The Helmholtz-Onsager Vortex Model . . . 4
2.2 Kernel smoothing . . . 7
2.3 Dynamic Vortex Pressure . . . 9
2.4 Surface Imprint . . . 11
2.5 Instantaneous Surface Deformation . . . 13
2.6 Surface Deformation Energy . . . 14
2.7 Prior Distribution . . . 15
**3 Numerical Model** . . . **17**
3.1 Non-dimensional Form . . . 17
3.2 Test Matrix . . . 18
3.3 Setup . . . 20
3.3.1 Boundary Conditions . . . 20
3.3.2 Statistical Surface Imprint Approach . . . 21
3.4 Eddy Separation . . . 22
3.5 Interaction Length . . . 23
3.6 Domain Proportions . . . 24
3.7 Grid Dependence . . . 27
**4 Markov Chain Monte Carlo** . . . **30**
4.1 Random Walk Metropolis . . . 31
4.2 Constraining Interaction Energy . . . 31

4.3 Parameter Tuning . . . 33
4.4 Convergence . . . 34
4.5 A Note on Hamiltonian Monte Carlo . . . 36
**5 Numerical Implementation . . . 38**
5.1 Class Structure . . . 38
5.2 Code Implementation . . . 39
5.2.1 Computing Dynamic Vortex Pressure . . . 39
5.2.2 Computing Surface Elevation . . . 40
5.2.3 Generating Turbulence From a Posterior Distribution . . . 40
5.2.4 Updating Interaction Properties . . . 41
5.3 Computational Efficiency . . . 41
5.3.1 Parallelization . . . 41
5.3.2 Vectorization . . . 41
5.3.3 Performance . . . 42
**6 Results . . . 43**
6.1 Posterior Distribution . . . 43
6.2 Surface Statistics . . . 48
**7 Discussion . . . 53**
7.1 Posterior Distribution . . . 53
7.1.1 Limitations . . . 54
7.2 Surface Statistics . . . 54
7.3 Suggested Model Improvements . . . 55
**8 Conclusion . . . 57**
**Bibliography . . . 59**
**A Matlab Code . . . 62**
A.1 Classes . . . 62
A.2 RWM script . . . 81
A.3 SDS script . . . 84
A.4 Supporting functions . . . 87

# Figures


2.1 Showing a system of turbulent eddies modelled as blobs of vorticity with linear impulse $L_e$. Figure from *Davidson* [20] . . . . . 4
2.2 Axisymmetric velocity profile with different core smoothing functions compared to the $r^{-3}$ dependence of no smoothing. Eddy centre in origin and impulse perpendicular to r-axis with $L = 0.1$ and $R = 0.1$ in non-dimensional units. . . . . 8
2.3 Comparing the axisymmetric velocity field of the smoothed kernel to Hill's spherical vortex, with $L = 0.1$ and $R = 0.2$ in non-dimensional units. The dotted circle has radius $R$. . . . . 9
2.4 An eddy mirrored about $z = 0$ . . . . . 10

3.1 Length-scale velocity-scale phasespace plot with shaded regions indicating region of marginal breaking for FST as per *Brocchini & Peregrine* [30]. Dashed lines show constant Froude numbers and dotted lines show constant Weber numbers. Experiments from from *Savelsberg & Van de Water* [17] in red with range from Taylors microscale to integral scale. The simulations of this paper are represented as blue dots. . . . . 19
3.2 Setup as seen from above, demonstrating periodic boundary conditions and interaction. Eddies are depicted as dots and arrows, representing $\mathbf{x}^n$ and $\mathbf{L}^n$ respectively, superimposed on the surface. The contour plot shows the surface deformation from the eddy in red, and the sphere of minimum separation is shown as dashed red circle. The interaction sphere of influence for the eddy in red is shown as the circle in cyan and the interacting eddies within the sphere of influence are highlighted in cyan. . . 21
3.3 Domain width required to capture the 95% highest surface elevations as a function of eddy submergence s for different combinations $Bo$. The dashed line represent the linear relationship with the slope found from a similar analysis witht the DVP . . . . . 26
3.4 Surface elevation from an eddy and its mirror normalized by the maximum value for different impulse angles $\alpha$. Blue contour shows where 5% of maximum surface elevation is, and black dotted circle bounds this for all $\alpha$. . . . . 27
3.5 Grid dependence of surface deformation energy with the given domain width for four different eddy depths and the Bond numbers from the test matrix. . . . . 28

4.1 Demonstrating effect of bounding interaction energy for a typical simulation. Top figure showing impulse components over a segment of iterations. Bottom figure showing terms of the Hamiltonian for the same iterations. . . . . 33

4.2 Monitoring change in the maximum change over $z$ in sampled parameters of interest $p(z)$ and $\sigma_{|\mathbf{L}|}(z)$ for an increasing number of iterations for all 9 simulations in the test matrix. . . . . 34
4.3 Autocorrelation function of the Hamiltonian for a typical simuation. Here, the first 300 iterations are shown to demonstrate the burn-in period. . . . . 35
4.4 Evolution of absolute value of Spearman correlation between $L_x$ and $L_y$ for an increasing number of iterations. Subfigure shows a zoomed in version of the final 300,000 iterations . . . . . 36

6.1 Normalized frequency distribution histograms of sampled $p(z)$ for the different simulations parameters in the test matrix. The dotted line is the uniform distribution for the spacial prior distribution, and the dashed line is the equilibrium depth discussed in section 3.6 . . . . . 44
6.2 Two-dimensional normalized histograms of the $p(L_\perp, z)$ distributions for the different simulations. The red dashed lines show the 99.7% confidence interval of the prior distribution. . . . . 45
6.3 Two-dimensional normalized histograms of the $p(L_\parallel, z)$ distributions for the different simulations. The red dashed lines show the 99.7% confidence interval of the prior distribution. . . . . 46
6.4 Standard deviation of impulse components as a function of depth for the different simulations in the test matrix. The dotted line represents the standard deviation of the prior distribution. . . . . 47
6.5 Normalized histograms of sampled surface elevation distributions away from mean. The different simulations are grouped by Bond number. . . . . 50
6.6 Statistical central moments for an increasing number of ensembles . . . . . 51

# Tables

4.1 Acceptance ratio at final iteration for RWM simulations with different parameters . . 34
4.2 Spearman correlation between $L_x$ & $L_y$ at final iteration for RWM simulations . . . . 36
4.3 p-value of Spearman correlation between $L_x$ and $L_y$ at final iteration for RWM simulations with 95% confidence interval . . . . . 36

6.1 Variance of sampled surface deformation distribution . . . . . 52
6.2 Skewness of sampled surface deformation distribution . . . . . 52
6.3 Kurtosis of sampled surface deformation distribution . . . . . 52
6.4 Maximum surface gradient . . . . . 52

# Nomenclature

### Accents

| | | |
|---|---|---|
| $\breve{a}$ | Smoothed variable | |
| $\dot{a}$ | Time derivative | |
| $\hat{a}$ | Non dimensional variable | |
| $\overline{a}$ | Mirror | |
| $\widetilde{a}$ | Fourier transformed variable | |
| $a_{\parallel}$ | Horizontal vector component | |
| $a_{\perp}$ | Vertical vector component | |

### Constants

| | | |
|---|---|---|
| $\gamma$ | Kinematic surface tension coefficient | $76 \times 10^{-6}\, m^3/s^2$ |
| $g$ | Gravitational acceleration | $9.81\, m/s^2$ |

### Physical parameters

| | | |
|---|---|---|
| $\beta$ | Inverse eddy temperature | $s^2/m^5$ |
| $\lambda$ | Length scale | $m$ |
| $\mathbf{u}$ | Velocity vector | $m/s$ |
| $\mathbf{x}$ | Position vector | $m$ |
| $\omega$ | Frequency | $s^{-1}$ |
| $\phi$ | Potential function | $m^2/s$ |
| $\boldsymbol{\xi}$ | Vorticity vector | $s^{-1}$ |
| $\tau$ | Time scale | $s$ |
| $\zeta$ | Surface elevation | $m$ |

| | | |
|---|---|---|
| $Bo$ | Bond number | - |
| $E$ | Kinematic energy | $m^5/s^2$ |
| $Fr$ | Turbulent Froude number | - |
| $H$ | Hamiltonian | $m^5/s^2$ |
| $K$ | Kinematic kinetic energy | $m^5/s^2$ |
| $k$ | Wavenumber | $m^{-1}$ |
| $L$ | Kinematic linear impulse | $m^4/s$ |
| $M$ | Kinematic mass inertia | $m^3$ |
| $P$ | Kinematic pressure | $m^2/s^2$ |
| $R$ | Eddy radius | m |
| $t$ | Time | $s$ |
| $We$ | Turbulent Weber number | - |

### Statistical parameters

| | |
|---|---|
| $\mathcal{N}$ | Normal distribution |
| $\mathcal{U}$ | Uniform distribution |
| $\mu$ | Mean |
| $\sigma$ | Standard deviation |
| $\theta$ | System state |

### Model parameters

| | |
|---|---|
| $\alpha$ | Impulse vector angle with surface |
| $\delta$ | Minimum eddy separation distance |
| $GP$ | Gridpoints |
| $h$ | Domain height |
| $r_c$ | Maximum eddy interaction distance |
| $V$ | Domain volume |
| $w$ | Domain width |

### Other symbols

| | |
|---|---|
| $\delta_{ij}$ | Kroenecker delta |
| $\epsilon_{ijk}$ | Levi-Civita symbol |
| $\mathbb{B}, \mathbb{C}, \mathbb{D}$ | Propagation tensors |
| $\mathbf{A}$ | Vector potential |

# Acronyms

**ACF** AutoCorrelation Function. 35

**BC** Boundary Condition. 9, 53, 57

**DNS** Direct Numerical Simulation. 2

**DVP** Dynamic Vortex Pressure. 9–14, 22–25, 28, 30, 39–41, 53, 54, 57, 58

**FFT** Fast Fourier Transform. 13, 40

**FST** Free Surface Turbulence. 1, 2, 19

**HMC** Hamiltonian Monte Carlo. 36, 56

**IC** Initial Condition. 6, 13

**MCMC** Markov Chain Monte Carlo. iii, iv, 2, 3, 7, 18, 19, 21, 28, 30–32, 34, 39, 41, 42, 53–55

**NST** Near Surface Turbulence. 1, 2, 9, 19

**PBC** Periodic Boundary Condition. 20, 24, 26, 38

**PDF** Probability Distribution Function. iii, 2, 15, 17, 22, 43

**RWM** Random Walk Metropolis. 31, 33, 36

**SD** Standard Deviation. 34, 44, 48

**SDS** Surface Deformation Statistics. 3, 22, 26, 39, 41, 42, 54, 55, 57

**SOI** Sphere Of Influence. 20, 38, 41

# Chapter 1

# Introduction

## 1.1 Motivation

The shape of the ocean surface controls the flux of heat and gas from the ocean to the atmosphere. The amount of transport of these properties can affect entire ecosystems [1], and more importantly still, global warming [2]. A key aspect affecting the surface shape is Near Surface Turbulence (NST), i.e. turbulent currents in the uppermost layer of the ocean [3]. These currents occur most commonly due to wind shearing the surface layer and transferring turbulent energy to the ocean, where the NST then continuously deforms the surface above it [4]. This interaction is not well understood however, as the literature is lacking in quantifying the types of eddies that constitute the turbulence in this layer.

In this report, turbulence will be modelled as a system of discrete eddies instead of the turbulent field quantities normally associated with turbulence modelling, in order to investigate the Free Surface Turbulence (FST). This approach might provide a direct correlation between the eddy properties and surface imprint. It also reduces the complexity of the system significantly. For this purpose, the Helmholtz-Onsager vortex model is used, a classic turbulence model that approximate each turbulent eddy as a "blob" of vorticity in an elsewhere inviscid fluid. The vortices constitute a Hamiltonian system that exchange velocity and impulse with each other.

Obviously, real turbulence does not consist of a system of discrete vortices, as described above. However, the approximation is not that far-fetched, as vorticity in turbulence is well known to be highly concentrated in small vorticial structures [5]. Using a simple model such as Helmholtz-Onsager vortices might therefore approximate the turbulence well enough to gain insights into the dynamics of the system. A major advantage of this approach is that the theoretical framework is analytical, reducing the need for complex numerical schemes. The model is also well established due to the dipole analogy extending to electrodynamics where it has had many applications.

This report then aims to answer the question: *By modelling near surface turbulence as a set of Helmholtz-Onsager vortices, what qualitative correlations could be drawn between the eddies and the surface imprint?*

## 1.2 Literary Review

Studying fluid vortex dynamics dates all the way back to the vortex theorems of *Helmholtz* [6]. Already in *1858*, he drew the peculiar analogy between a fluid vortex and an electromagnetic dipole. *Onsager* [7] then extended the idea to statistical mechanics. He formulated the evolution of a system of vortices by means of a conserved quantity from classical mechanics, known as the *Hamiltonian* [8]. This meant that turbulence could be heuristically modelled as a finite set of the vortices found by Helmholtz. Motivated by the innovation in computers, the Hamiltonian formalism was later extended to numerics by *Roberts* [9], with notable contributions by *Buttke & Chorin* [10]. However, this was largely overshadowed by classic reynold stress turbulence models and Direct Numerical Simulation (DNS), as computers became more efficient. It should be noted that the Helmholtz-Onsager vortex model is only one of many attempts to approximate turbulence as a finite set of vortices. For an extended review, see *Pullin & Saffmann* [11].

NST has so far never been modelled with Helmholtz-Onsager vortices to the extent of my knowledge. The literature of single vortex structures interacting with a free surface seems to be sparse in general. There are some studies investigating the vortex ring near a free surface, most notably *Bernal & Kwon*, *Song et al.* and *Gharib & Weigand* [12–14]. They find that vortex filaments near a free surface break up into tubes ending perpendicular to the surface, in accordance to Helmholtz's second theorem. These in turn cause downwellings. Albeit interesting, this mechanism for surface deformation is different from what is regarded in the present work. There are also some analytical studies on surface deformation using line vortices, e.g *Telste* [15] or *Tyvand* [16]. While these are useful in investigating for example the vortex shedding of a hydrofoil, they cannot represent the three dimensional nature of turbulent eddies.

The literature on general FST is considerably richer. There are several studies relating turbulence to surface deformation both experimentally and using DNS. *Savelsberg & Van de Water* [17] found experimentally a strong correlation between a single sub surface vortex and surface deformation. This correlation was dramatically reduced for fully developed turbulence, because of the turbulent eddies exciting gravity-capillary wave motion on the surface. *Guo & Shen* [18] found using DNS that the surface induced roughness consist of propagating waves at all wavenumbers from FST. Furthermore, they found that the surface elevation is sensitive to gravitational and surface tension effects. These studies find correlations between the average turbulent field quantities and surface crispations. However, measurable parameters of the eddies themselves are harder to extrapolate from the data. These parameters they conclude are the main contributors to surface renewal, which emphasizes a need to quantify eddy properties to further understand surface imprints from NST.

## 1.3 Scope

The report will look at the turbulence-surface interaction in two parts. First, it is investigated how a free surface affects the turbulent eddies. This will be formulated in a manner of how a free surface affects the PDF of a turbulent eddy, numerically in terms of a MCMC algorithm. Then the eddies are drawn from this PDF in order to see how they in turn affect the surface imprint. The goal during these processes is not to validate the model, as it is excessively simple to realistically accomplish that. Rather, the aim is to see how the system scales with the turbulence parameters, with an ambition to find some defining features of the interaction worth looking more into.

Some key simplifications are made in the theoretical model. The flow is modelled as inviscid, and as a consequence all eddies must be well separated. The surface is approximated to linear order, restricting the analysis to non-steep surface deformations. Additionally some simplifications are made in the numerical model for pragmatic reasons. During MCMC simulations, only a single eddy is considered in order to save computation time, meaning eddy-eddy interaction is ignored. Also a kernel smoothing function is used to obtain convergence of the MCMC simulations. Furthermore, it is assumed without proof that the surface deformation energy enters into the Hamiltonian structure as to conserve the total energy of the system. When evaluating interaction between eddies for the surface statistics analysis, a maximum interaction length is set as a cutoff criterion, again to save computation time.

## 1.4 Outline

Chapter 2 firstly gives a short review of the Helmholtz-Onsager vortex model, followed by a kernel smoothing method in order to address a blow-up workaround for MCMC simulations. Then the model is extended to surface waves, which *Simen Ellingsen* has developed the theoretical framework for. This is unpublished work, and is thus included in the report as the subject of sections 2.3, 2.4 and 2.5. It should be noted that this is entirely his work and it is merely reworded here to fit the context of this report. Thereafter is a brief review of surface deformation energy and some basic statistical mechanics for the purpose of implementing the MCMC algorithm. Chapter 3 gives an overview of the numerical model used to implement the theory along with the decision of some important model parameters such as domain size, grid resolution and eddy interaction length. Then follows chapter 4, an entirely separate chapter dedicated to how the model is tailored to implement the MCMC algorithm. Chapter 5 concerns some details as to how the numerical model is implemented in *Matlab*. The results of the MCMC simulations and the following Surface Deformation Statistics (SDS) simulations are presented in chapter chapter 6 and discussed in chapter 7, alongside some ideas for improving the model.

# Chapter 2

# Theory

## 2.1 The Helmholtz-Onsager Vortex Model

This is a well established model and thus only the essentials are covered here. Similar, more extensive reviews can be found in *Lamb* [19] or *Davidson* [20].

As a preliminary remark it should be said that throughout the report, a superscript variable represents the index of an eddy, and must not be confused with an exponent. Consider an incompressible fluid that is irrotational everywhere except for $N$ "blobs" of vorticity with finite volumes $\mathcal{V}^n$, corresponding to turbulent eddies, like shown in Figure 2.1.

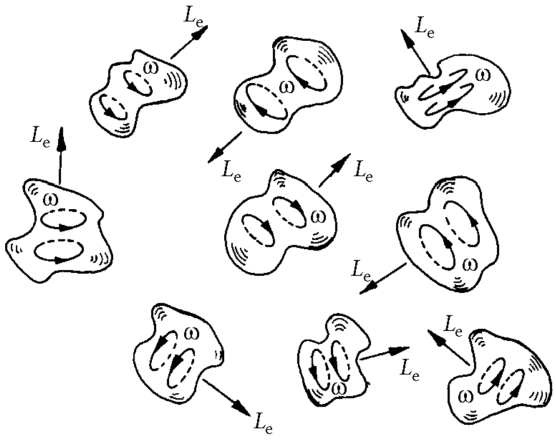



**Figure 2.1:** Showing a system of turbulent eddies modelled as blobs of vorticity with linear impulse $L_e$. Figure from *Davidson* [20]

Outside the eddies where there is no vorticity, the Euler equation can be used. Ignoring gravitational effects, it reads

$$\dot{\mathbf{u}} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p. \tag{2.1}$$

Taking the curl of this, and using the definition of voriticity as $\boldsymbol{\xi} = \nabla \times \mathbf{u}$, we are left with the well known vorticity equation

$$\dot{\boldsymbol{\xi}} + (\mathbf{u} \cdot \nabla)\boldsymbol{\xi} = (\boldsymbol{\xi} \cdot \nabla)\mathbf{u}. \tag{2.2}$$

The relationship can be inverted to derive the velocity field in terms of the vorticity distributon, resulting in the Biot-Savarts law

$$\mathbf{u}(\mathbf{x}) = \frac{1}{4\pi} \int \frac{\boldsymbol{\xi}(\mathbf{x}') \times \left(\mathbf{x} - \mathbf{x}'\right)}{|\mathbf{x} - \mathbf{x}'|^3} \mathrm{d}^3 x', \tag{2.3}$$

where $\mathbf{x}'$ denotes a position vector inside the eddy. Because the vorticity is zero except for inside the eddies, Equation 2.3 can be rewritten in terms of each eddy's vector potential $\mathbf{A}^n$. This will be defined as

$$\mathbf{A}^n(\mathbf{x}) = \frac{1}{4\pi} \int_{\mathcal{V}^n} \frac{\boldsymbol{\xi}(\mathbf{x}')}{|\mathbf{x} - \mathbf{x}'|} \mathrm{d}^3 x', \tag{2.4}$$

such that

$$\mathbf{u}(\mathbf{x}) = \nabla \times \sum_{n=1}^{N} \mathbf{A}^n(\mathbf{x}). \tag{2.5}$$

We then define a vector from anywhere in the fluid to the eddy centre $\mathbf{x}^n$ as $\mathbf{r}^n = \mathbf{x} - \mathbf{x}^n$, and from a point within the eddy to the centre as $\mathbf{x}'' = \mathbf{x}' - \mathbf{x}^n$. Then, the denominator in Equation 2.4 could be Taylor expanded to give

$$\frac{1}{|\mathbf{x} - \mathbf{x}'|} = \frac{1}{|\mathbf{r}^n - \mathbf{x}''|} = \frac{1}{r^n} + \frac{\mathbf{x}'' \cdot \mathbf{r}^n}{r^{n^3}} + 4\pi \mathbf{x}'' \cdot \mathbb{C}\left(\mathbf{r}^n\right) \cdot \mathbf{x}'' + \ldots. \tag{2.6}$$

These terms are called the *monopole*, *dipole* and *quadrupole* moments. $\mathbb{C}$ is the Hessian of $(4\pi r)^{-1}$, a rank 2 tensor. This, along with two other tensors that occur from the Taylor expansion are defined for later use in index notation as

$$\begin{aligned} B_i &= \partial_i \frac{1}{4\pi r} = -\frac{r_i}{4\pi r^3} \\ C_{ij} &= \partial_i B_j = \frac{1}{4\pi r^5}(3 r_i r_j - r^2 \delta_{ij}) \\ D_{ijk} &= \partial_i C_{jk} = \frac{3}{4\pi r^7}\left[\left(r_i \delta_{jk} + r_j \delta_{ki} + r_k \delta_{ij}\right) r^2 - 5 r_i r_j r_k\right]. \end{aligned} \tag{2.7}$$

Truncating the third and all higher order terms of Equation 2.6, this could be substituted back into Equation 2.4. The higher order moments we drop are dominant when eddies are close. We have thus limited ourselves to well separated eddies. Note that the first term, i.e. the monopole moment is simply zero by using the divergence theorem, because there cannot be any flux of vorticity through the boundary of the eddy $\partial \mathcal{V}^n$. The only term left is the dipole moment. In index notation, where Einstein summation convention is to be understood, the vector potential is then

$$A_i^n(\mathbf{x}) = \frac{r_j^n}{r^{n^3}} \int_{\mathcal{V}^n} \omega_i' x_j' \mathrm{d}^3 x'. \tag{2.8}$$

The linear *dipole impulse* is defined as

$$\mathbf{L}^n = \frac{1}{2}\int_{\mathcal{V}n} \mathbf{x}\times\boldsymbol{\omega}\mathrm{d}^3x. \tag{2.9}$$

The relation $\partial_k x_i x_j a_k = a_i x_j + a_j x_i$ holds for divergence free vector fields [21]. Combining this relation with the diverge theorem, it is possible to show that Equation 2.8 can be rewritten as

$$\mathbf{A}^n = -\frac{\mathbf{r}^n\times\mathbf{L}^n}{4\pi(r^n)^3}. \tag{2.10}$$

Finally, curling and summing over the eddies according to Equation 2.5 gives the far-field velocity field, i.e. far from any particular eddy as

$$\mathbf{u}_{\mathrm{ff}}(\mathbf{x}) = \sum_{n=1}^{N}\mathbb{C}(\mathbf{r}^n)\cdot\mathbf{L}^n. \tag{2.11}$$

The evolution of the system is then given by the Hamiltonian. This is a conserved quantity from classical mechanics that describes the total energy of the system in terms of coordinates in phase space, called *canonical coordinates*. The Hamiltonian formalism here will be that of *Oseledets* [22]. He showed that the Euler equations can be expressed as a Hamiltonian system with $x_i^n$ and $L_i^n$ as the canonical coordinates as follows

$$\begin{aligned}\dot{x}_i^n &= \frac{\partial H}{\partial L_i^n}\\ \dot{L}_i^n &= -\frac{\partial H}{\partial x_i^n}.\end{aligned} \tag{2.12}$$

*Buttke & Chorin* [10] then showed that this Hamiltonian could be approximated by the kinetic energy of the system, which is

$$H \approx K = \sum_n \frac{L_i^n L_i^n}{2M^n} + \frac{1}{2}\sum_n\sum_{m\neq n} L_i^n C_{ij}^{nm} L_j^m. \tag{2.13}$$

Here, the first term is the self induced energy of the eddies, whilst the second term is the interaction energy between the eddies. $M^n$ is called the "inertial mass" of the eddy, and is approximated to be constant. It is not actually an inertia in the true physical meaning of the word, as it was derived by an analogy to electrodynamics. Nonetheless it inherits the same properties, such that self motion is slow with a large $M^n$, and vice versa. Differentiating Equation 2.13 according to Equation 2.12 yields the following time derivatives describing the time evolution of the system

$$\dot{x}_i^n = \frac{L_i^n}{M^n} + \sum_{m\neq n} C_{ij}^{nm} L_j^m \tag{2.14}$$

$$\dot{L}_i^n = -\frac{1}{2} L_j^n \sum_{m\neq n} D_{ijk}^{nm} L_k^m. \tag{2.15}$$

It is clear that the system is in full described by the properties $\mathbf{x}^n$, $\mathbf{L}^n$ and $M^n$. The time evolution is then given for any time $t$, provided with the Initial Condition (IC) for these properties of all relevant eddies. This of course only holds as long as the eddies remain well separated.

## 2.2 Kernel smoothing

Under the dipole approximation that was made, we have limited ourselves to well-separated eddies. However, it will prove impossible to retain this far-field approximation in full during MCMC simulations, as we will see in chapter 4. In order to retain the approximation as well as possible, *Qi* [23] proposes a smoothed kernel method in which the vector potential of Equation 2.10 is smoothed by some core function $f(\frac{r}{R})$, i.e. $\mathbf{A}^n \rightarrow \mathbf{A}^n f(\frac{r}{R})$, where $R$ is the eddy size. Then evaluating Equation 2.5 gives the following expression in index notation

$$\breve{\mathbf{u}}(\mathbf{r}^n) = \nabla \times \left( -\frac{f(r)}{4\pi r^3} \mathbf{r} \times \mathbf{L}^n \right) = -\epsilon_{ijk} \frac{\partial}{\partial x_j} \left[ \frac{f(r)}{4\pi r^3} \epsilon_{klm} r_l L_m^n \right], \tag{2.16}$$

where $\epsilon$ denotes the *Levi-Civita* symbol, and $\breve{\mathbf{u}}$ denotes the smoothed velocity field. Both $\mathbf{L}^n$ and $\epsilon_{klm}$ are independent of spacial coordinates, and can be drawn out of the derivative. Furthermore we utilize the well known property $\epsilon_{ijk}\epsilon_{klm} = \delta_{il}\delta_{jm} - \delta_{im}\delta_{jl}$ [24] such that

$$\breve{\mathbf{u}}(\mathbf{r}^n) = -\frac{L_m}{4\pi} \left( \delta_{il}\delta_{jm} \frac{\partial}{\partial x_j} \left( \frac{f(r)}{r^3} r_l \right) - \delta_{im}\delta_{jl} \frac{\partial}{\partial x_j} \left( \frac{f(r)}{r^3} r_l \right) \right]. \tag{2.17}$$

The *Kroenicker delta* is defined to be non-zero for equal indices, such that $i = l$ and $j = m$ for the first term and $j = l$ for the second. Expanding according to the chain rule then gives

$$\breve{\mathbf{u}}(\mathbf{r}^n) = -\frac{L_m^n}{4\pi} \left[ \frac{\partial}{\partial x_m} \left( \frac{f(r)}{r^3} \right) r_i r_m + \frac{f(r)}{r^3} \frac{\partial r_i}{\partial x_m} - \delta_{im} \left( r_j \frac{\partial}{\partial x_j} \left( \frac{f(r)}{r^3} \right) + \frac{f(r)}{r^3} \frac{\partial r_j}{\partial x_j} \right) \right]. \tag{2.18}$$

The tensor $\frac{\partial r_i}{\partial x_m}$ is simply the identity matrix $\delta_{im}$, and $\frac{\partial r_j}{\partial x_j}$ is obviously 3. The derivative $\partial_j(f/r^3)$ is easily evaluated to be

$$\frac{\partial}{\partial x_j} = \frac{r\frac{\partial f(r)}{\partial r} - 3f(r)}{r^5} r_j. \tag{2.19}$$

Because $r_j r_j = r^2$, the last term of Equation 2.18 cancels with the last term of the contribution from Equation 2.19. After some reshuffeling, Equation 2.18 can be rewritten in the following manner

$$\breve{\mathbf{u}}(\mathbf{r}^n) = -\left[ -f(r) \left( \frac{3r_i r_m - \delta_{im} r^2}{4\pi r^5} \right) + r f'(r) \left( \frac{1}{3} \left( \frac{3r_i r_m - \delta_{im} r^2}{4\pi r^5} \right) - \frac{1}{6\pi} \frac{\delta_{im}}{r^3} \right) \right] L_m^n, \tag{2.20}$$

with $f'$ denoting the derivative with respect to $r$. It is now obvious that the parenthesis terms are the propagator $C_{im}$ defined in Equation 2.7, such that

$$\breve{\mathbf{u}}(\mathbf{r}^n) = \left[ \left( f(r) - \frac{r}{3} f'(r) \right) \mathbb{C} + \frac{1}{6\pi r^2} f'(r) \mathbb{I} \right] \cdot \mathbf{L}^n, \tag{2.21}$$

where $\mathbb{I}$ is the identity matrix. Now all that is left is to evaluate in $r^* = r/R$. Then $f'(r) = \frac{1}{R} f'(r^*)$ and $\mathbb{C}(r) = \frac{1}{R^3}\mathbb{C}(r^*)$ such that the final expression reads

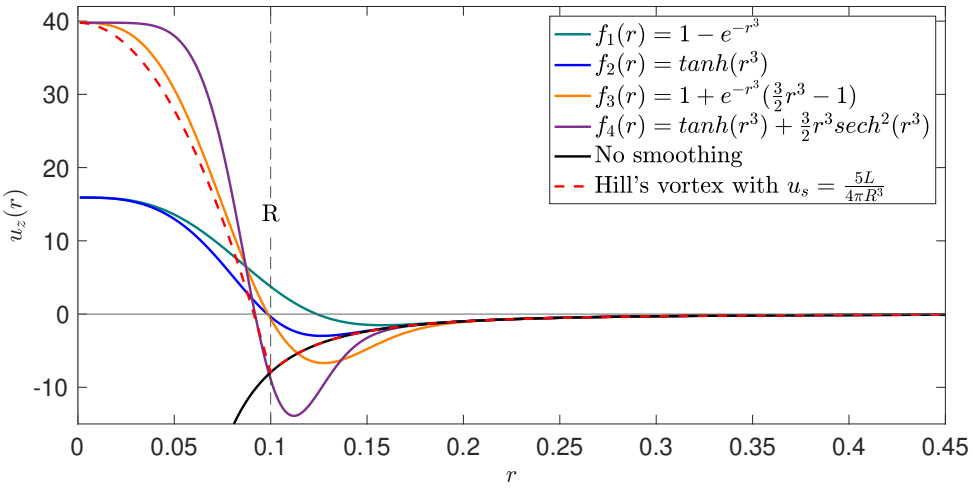



**Figure 2.2:** Axisymmetric velocity profile with different core smoothing functions compared to the $r^{-3}$ dependence of no smoothing. Eddy centre in origin and impulse perpendicular to r-axis with $L = 0.1$ and $R = 0.1$ in non-dimensional units.

$$\breve{\mathbb{C}}(\mathbf{r}_*^n) = \frac{1}{R^3}\left[\left(f(r_*) - \frac{r_*}{3}f'(r_*)\right)\mathbb{C}(\mathbf{r}_*) + \frac{1}{6\pi r_*^2}f'(r_*)\mathbb{I}\right]. \tag{2.22}$$

The velocity magnitude from the four different smoothing functions from *Qi* is plotted in Figure 2.2 along the perpendicular direction to the impulse vector, for a particular impulse magnitude and eddy radius. Note that the smoothed velocity profile goes to a finite value for $r = 0$, in the opposite direction of the velocity profile without smoothing. This is a result from giving the eddy a finite size, essentially making it a current loop. In the process above, a spherical shape is assumed. It is therefore interesting to compare to *Hill's* spherical vortex [25], which bears a close resemblance for the smoothed velocity field. The radial velocity profile is included in Figure 2.2, while the whole $u_z, u_r$ velocity profiles are compared in Figure 2.3.

For consistency, the propagator $\mathbb{D}^{nm}$ should also be smoothed whenever the induced impulse is calculated when using smoothing functions. Like in Equation 2.7, the smoothed propagator is $\breve{D}_{ijk} = \partial_j \breve{C}_{jk}$. Some tedious derivation of Equation 2.22 gives

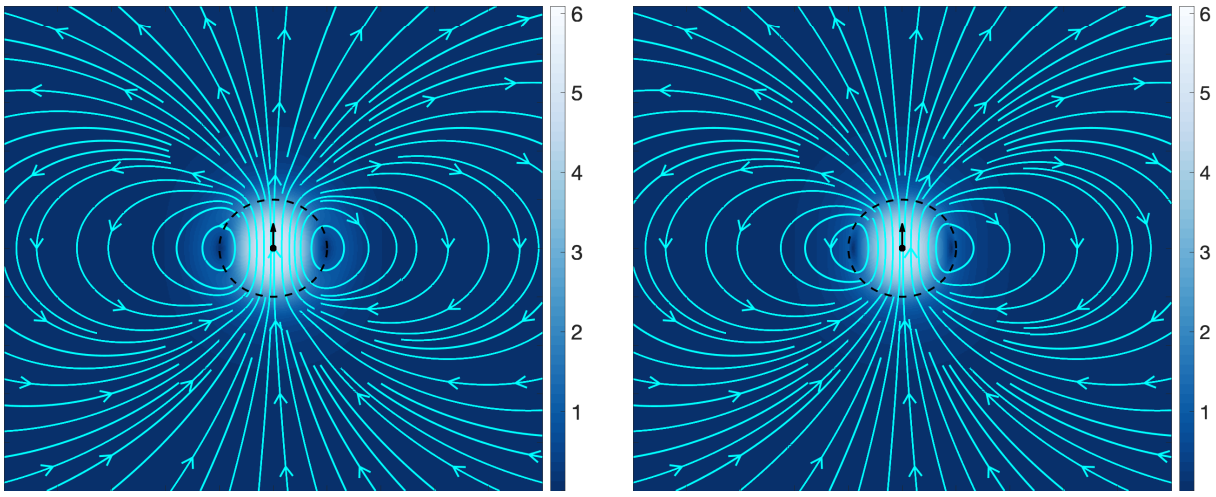
$$\breve{D}_{ijk} = D_{ijk}\left(g(r) - g'(r)\cdot r\right) + g'(r)\cdot r\cdot F_{ijk}, \tag{2.23}$$

where

$$F_{ijk} = \frac{3}{4\pi r^7}\left[\left(r_k\delta_{ij} + r_j\delta_{ki}\right)r^2 - 4r_i r_j r_k\right], \tag{2.24}$$

and

$$g(r) = f(r) - \frac{r}{3}f'(r). \tag{2.25}$$




**(a)** Smoothing with $f_3(r)$ **(b)** Hill's spherical vortex

**Figure 2.3:** Comparing the axisymmetric velocity field of the smoothed kernel to Hill's spherical vortex, with $L = 0.1$ and $R = 0.2$ in non-dimensional units. The dotted circle has radius $R$.

## 2.3 Dynamic Vortex Pressure

So far, the eddies have been considered as an isolated system. Introducing them beneath a free surface, two Boundary Condition (BC) needs to be fulfilled; the *kinematic*- and *dynamic* BCs [26]. The kinematic BC states that any particle on the free surface will stay on the surface, meaning that the surface moves with the flow. The dynamic BC states that the pressure is independent of the surface elevation, such that the pressure should be continuous on the interphase between air and water.

Surface tension plays an important role in NST however, and therefore sharp discontinuities will be present across the interphase. The procedure will therefore be to satisfy the BCs in two steps. First, only the dynamic BC is fulfilled by ignoring surface tension and wave motion, while solving for the pressure distribution on the surface. This distribution will from now on be referred to as the Dynamic Vortex Pressure (DVP). Then in section 2.4, the DVP will be applied to the surface as an externally applied pressure while letting the surface move freely, thus fulfilling the kinematic BC.

Ignoring wave motion is done by mirroring each eddy about the surface, which forces the vertical velocity at the surface to zero. The surface which we try to mirror about is itself unknown. However, a low wave steepness is assumed, i.e. $\nabla\zeta \ll 1$, such that the surface can be evaluated to be approximately $z = 0$. This is a linearization to the first order, and is common in wave theory [26]. The mirroring is illustrated in Figure 2.4.

First we consider the unsteady potential Bernoulli equation at the interphase with a suitable constant

$$g\zeta_d + \frac{1}{2}|\mathbf{u}_p|^2 + \left.\frac{\partial\phi}{\partial t}\right|_{z=0} = 0. \tag{2.26}$$

Here $\zeta_d(\mathbf{x}_\parallel, t)$ denotes the surface elevation corresponding to the dynamic BC. It is dependent on time $t$ and the planar coordinates $\mathbf{x}_\parallel = [x, y]$. $\mathbf{u}_p$ is the velocity at a point $p$ on the surface and

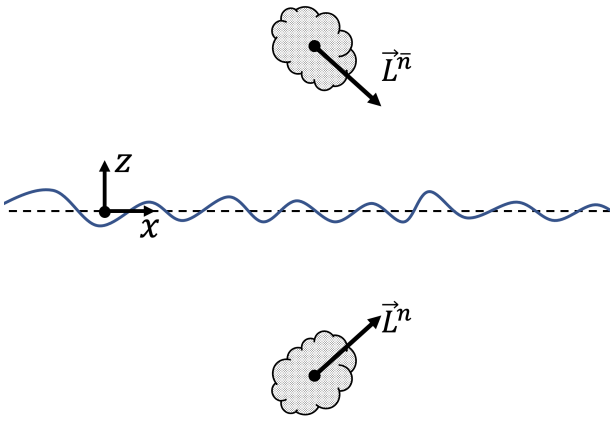



**Figure 2.4:** An eddy mirrored about $z = 0$

$\phi$ is the potential function. Note that all parameters are kinematic, i.e. divided by the density of the water. The DVP is then defined as the non-$\zeta_d$ terms as such:

$$P_d(\mathbf{x}_\parallel, t) := \frac{1}{2}|\mathbf{u}_p|^2 + \left.\frac{\partial \phi}{\partial t}\right|_{z=0}. \tag{2.27}$$

Note that this is a gauge pressure, i.e. relative to the atmosphere, and can therefore be negative. The first term of Equation 2.27 can be found directly by using Equation 2.11

$$\frac{1}{2}|\mathbf{u}_p|^2 = \frac{1}{2}\left(\sum_{n=1}^{N} \mathbb{C}(\mathbf{r}^{np}) \cdot \mathbf{L}^n\right)^2 = \frac{1}{2}\sum_{n,m} L_i^n C_{ij}^{np} C_{jk}^{mn} L_k^m. \tag{2.28}$$

The second term however requires some more work. As is customary, the potential function $\phi$ is defined as $\partial_i \phi = u_i$, and the contribution from each eddy likewise $\partial_i \phi^n = u_i^n$. It follows from Equation 2.11 and the definition of the propagators in Equation 2.7 that

$$u_i^n = C_{ij}^{np} L_j^n = \partial_i (B_j^{np} L_j^n). \tag{2.29}$$

This makes it obvious that the potential function associated with each eddy is

$$\phi^n = B_j^{np} L_j^n. \tag{2.30}$$

We see that $\phi$ is a function of $\mathbf{x}^n$ and $\mathbf{L}^n$. Using the product rule we can then find the time derivative of $\phi$ as the following expression

$$\frac{\partial \phi^n}{\partial t} = \frac{\partial \phi^n}{\partial x_i^n}\dot{x}_i^n + \frac{\partial \phi^n}{\partial L_i^n}\dot{L}_i^n. \tag{2.31}$$

The time derivatives of this expression has already been established in Equation 2.14 and Equation 2.15. The remaining terms are found by differentiating Equation 2.30 to be

$$\frac{\partial \phi}{\partial x_i^n} = u_i^n = C_{ij}^n L_j^n, \tag{2.32}$$

and

$$\frac{\partial \phi}{\partial L_i^n} = B_i^n. \tag{2.33}$$

Inserting these into Equation 2.31 we get the full time derivative of the potential function from each eddy $n$

$$\frac{\partial \phi^n}{\partial t} = \frac{1}{M^n} L_i^n C_{ij}^{np} L_j^n + C_{ij}^{np} L_j^n \sum_{m \neq n} C_{jk}^{mn} L_k^m - B_j^{np} L_j^n \sum_{m \neq n} D_{ijk}^{nm} L_k^m. \tag{2.34}$$

Finally summing Equation 2.34 over $n$ and evaluating at $z = 0$, the DVP becomes

$$P_d(\mathbf{x}_\parallel, t) = \frac{1}{2} \sum_{n,m} L_i C_{ij}^{np} C_{jk}^{mn} L_k^m + \sum_n \frac{L_i^n C_{ij}^{np} L_j^n}{M^n} + \sum_{n,m \neq n} \left( C_{ij}^{np} L_j^n C_{jk}^{mn} L_k^m - B_i^{np} L_j^n D_{ijk}^{nm} L_k^m \right). \tag{2.35}$$

The first term is caused by velocity induced at the surface. It will henceforth be referred to as the *kinetic pressure*. The second term is due to self motion of the eddy. The two last terms are the pressure due to interaction between eddies, from induced velocity and induced impulse respectively.

The terms $\sum_{m \neq n} \mathbb{C}^{mn} \cdot \mathbf{L}^m$ and $\mathbb{C}^{np} \cdot \mathbf{L}^n$ are the total induced velocity of eddy $n$ and surface velocity from eddy $n$ respectively, and will be denoted $\dot{\mathbf{x}}^n$ and $\mathbf{u}_p^n$. Equation 2.35 will prove to be more convenient decomposed as a superposition of all the eddies $P_d = \sum_n P_d^n$, in vector form

$$P_d^n(\mathbf{x}_\parallel, t) = \underbrace{\frac{1}{2} \mathbf{u}_p^n \cdot \mathbf{u}_p^{tot}}_{P_{Kin}^n} + \underbrace{\frac{1}{M^n} \mathbf{u}_p^n \cdot \mathbf{L}^n}_{P_{Self}^n} + \underbrace{\mathbf{u}_p^n \cdot \dot{\mathbf{x}}^n}_{P_{Int,Vel}^n} + \underbrace{\mathbb{B}_p^n \cdot \dot{\mathbf{L}}^n}_{P_{Int,Imp}^n}. \tag{2.36}$$

## 2.4 Surface Imprint

Now, we demand that the kinematic BC is fulfilled by means of introducing a velocity potential due to wave motion $\phi_w$. Consider the Bernoulli equation with this wave potential evaluated just beneath the surface

$$g\zeta + P + \frac{\partial \phi_w}{\partial t} = 0. \tag{2.37}$$

Now $\zeta(\mathbf{x}_\parallel, t)$ denotes the true surface elevation. The pressure P just beneath the surface can be related to the pressure right above it $P_d$ with surface tension by $P = P_d - \gamma \nabla_\parallel^2 \zeta$ [26], such that

$$g\zeta + P_d + \frac{\partial \phi_w}{\partial t} = \gamma \nabla_\parallel^2 \zeta. \tag{2.38}$$

$\gamma$ is the kinematic surface tension coefficient, and $\nabla_\parallel^2$ is the Laplace operator with respect to the planar coordinates. The only term unrelated to the desired surface distribution $\zeta$ is $\dot{\phi}_w$. The kinematic BC takes care of this as it can be formulated as

$$\frac{\partial \phi_w}{\partial z} = \left. \frac{\partial \zeta}{\partial t} \right|_{z=0}. \tag{2.39}$$

Now, two-dimensional spacial Fourier transforms of Equation 2.38 and Equation 2.39 can be done in the planar coordinates. The Fourier transformed variables will be the wave number components $\mathbf{k}_\parallel = [k_x, k_y]$, which together span what is called *k-space*. The length of the vector $\mathbf{k}_\parallel$ will be denoted $k$. The Fourier transformed DVP and surface elevation become respectively

$$\widetilde{P}_d(\mathbf{k}_\parallel, t) = \int P_d\left(\mathbf{x}_\parallel, t\right) e^{-i\mathbf{k}_\parallel \cdot \mathbf{x}_\parallel} d x_\parallel^2 \tag{2.40}$$

and

$$\widetilde{\zeta}(\mathbf{k}_\parallel, t) = \int \zeta\left(\mathbf{x}_\parallel, t\right) e^{-i\mathbf{k}_\parallel \cdot \mathbf{x}_\parallel} d x_\parallel^2, \tag{2.41}$$

where a tilde denotes a variable in Fourier space. The wave potential is in general three-dimensional in space, with the known form $\phi_w(\mathbf{x}, t) = A(t)e^{kz}e^{i\mathbf{k}_\parallel \cdot \mathbf{x}_\parallel}$. From this we can rewrite it as a spacial Fourier transform in the planar coordinates

$$\widetilde{\phi_w}(\mathbf{x}, t) = \frac{1}{(2\pi)^2} \int \phi_w\left(\mathbf{x}_\parallel, t\right) e^{kz} e^{i\mathbf{k}_\parallel \cdot \mathbf{x}_\parallel} d k_\parallel^2, \tag{2.42}$$

which satisfies $\phi_w = \mathcal{F}^{-1}\{\widetilde{\phi_w}\}$. The Fourier transforms of equations 2.38 and 2.39 become

$$\widetilde{P}_d + g\widetilde{\zeta} + \dot{\widetilde{\phi}}_w = -\gamma k^2 \widetilde{\zeta}, \tag{2.43}$$

and

$$k\widetilde{\phi_w} = \dot{\widetilde{\zeta}}. \tag{2.44}$$

Note that the $k$ components come from spacial differentiation of the exponents within the integrals from Equation 2.41 and Equation 2.42. Differentiating Equation 2.44 with respect to time and combining with Equation 2.43 gives a second order PDE for $\widetilde{\zeta}$

$$\ddot{\widetilde{\zeta}} + \omega^2(k)\widetilde{\zeta} = -k\widetilde{P}_d(\mathbf{k}_\parallel, t). \tag{2.45}$$

Here,

$$\omega(k) = \sqrt{gk + \gamma k^3} \tag{2.46}$$

is the well known deep water dispersion relation for gravity-capillary waves [19]. The dispersion relation is an important equation in wave dynamics, as it governs a direct relationship between the wavenumber and frequency of a wave. Waves are typically superposed of several *monochromatic* waves, i.e. waves of a single wavelength. Each of these waves can move independently with the frequency given by the dispersion relation. Note that the surface tension term dominate for high wavenumbers, corresponding to short wavelengths. These types of waves are called *capillary waves*. At lower wavenumbers, in what is called the *gravity wave regime*, gravity effects dominate.

Assuming all waves are caused by the DVP, one can keep the non-homogeneous solution of Equation 2.45, meaning it is sufficient to find a particular solution. For this we will use a Green function approach. For the unfamiliar, see [27]. This in principle means we will treat $P_d$ as an infinite set of Dirac delta functions $\delta(t)$ and compute the surface elevation due to the impulses at a later

time. The linear differential operator $\hat{\mathcal{D}} = (\partial_t^2 + \omega^2)$ is introduced, such that Equation 2.45 and the yet unknown Green function propagator G satisfy respectively

$$\hat{\mathcal{D}}\widetilde{\zeta} = -k\widetilde{P}_d(\mathbf{k}_\parallel, t), \tag{2.47}$$

and

$$\hat{\mathcal{D}}G = \delta(t). \tag{2.48}$$

Because $\hat{\mathcal{D}}$ is linear, the superposition principle holds, and the particular solution of Equation 2.47 is

$$\widetilde{\zeta}(\mathbf{k}_\parallel, t) = -\int_{-\infty}^{\infty} G\left(t - t'\right) k\widetilde{P}_d(\mathbf{k}_\parallel, t) dt'. \tag{2.49}$$

Now, it is demanded that the propagator G only propagates forward in time, which is to say that cause predates effect. To solve Equation 2.48 for G we need two ICs. The first is that G needs be zero at $t = 0$ because it cannot react instantaneously to an impulse. The second is obtained from integrating both sides of Equation 2.48 over the pulse, giving the initial time derivative of G. Then the ICs are

$$\begin{aligned} G(\mathbf{k}_\parallel, 0) &= 0 \\ \dot{G}(\mathbf{k}_\parallel, 0) &= 1. \end{aligned} \tag{2.50}$$

It is obvious that Equation 2.48 is homogeneous for $t > 0$, which gives the general solution

$$G(\mathbf{k}_\parallel, t) = G_1(\mathbf{k}_\parallel)\cos(\omega t) + G_2(\mathbf{k}_\parallel)\sin(\omega t). \tag{2.51}$$

Inserting the ICs from Equation 2.50, the propagator G becomes

$$G(\mathbf{k}_\parallel, t) = \frac{1}{\omega}\sin(\omega t), \tag{2.52}$$

where $t > 0$. Inserting this into Equation 2.49 gives the final expression for the Fourier transformed surface elevation

$$\widetilde{\zeta}(\mathbf{k}_\parallel, t) = -\frac{k}{\omega}\int_{-\infty}^{t} \widetilde{P}_d(\mathbf{k}_\parallel, t')\sin[\omega(k)\left(t - t'\right)] dt'. \tag{2.53}$$

This represents a direct relationship between the dynamic vortex pressure and the surface elevation in k-space. With the known DVP distribution from section 2.3, we can then Fourier transform $P_d$ with the Fast Fourier Transform (FFT) algorithm, evaluate the integral in Equation 2.53, and do the inverse FFT to get the surface elevation.

## 2.5 Instantaneous Surface Deformation

The expressions for $\zeta$ found in section 2.4 are valid, given the assumptions made. However, it will prove to be useful to do a multiple-scale expansion in order to evaluate the instantaneous surface deformation. This is done by assuming a timescale for the eddies $T$ that is much slower than the

capillary waves and non-dimensionalize time as $\tau = \omega t$ and $\varepsilon = \omega T$, such that $\varepsilon \ll 1$. Then Equation 2.53 becomes

$$\widetilde{\zeta}(\mathbf{k}_{\parallel}, \tau, \epsilon\tau) = -\frac{k}{\omega^2} \int_{-\infty}^{t} \widetilde{P}_d(\mathbf{k}_{\parallel}, \varepsilon\tau') \sin\left(\tau - \tau'\right) d\tau', \tag{2.54}$$

Tayor expanding the DVP inside the integral we get

$$\widetilde{P}_d(\mathbf{k}_{\parallel}, \varepsilon\tau) = \widetilde{P}_d + (\varepsilon\tau)\dot{\widetilde{P}}_d + \frac{1}{2}(\varepsilon\tau)^2 \ddot{\widetilde{P}}_d + \mathcal{O}((\epsilon\tau)^3) \tag{2.55}$$

Dropping higher order terms and inserting Equation 2.55 into Equation 2.54 and using the known integral solutions

$$\int_{-\infty}^{t} t' \sin\left[\omega\left(t - t'\right)\right] dt' = 0 \tag{2.56}$$

$$\int_{-\infty}^{t} t'^2 \sin\left[\omega\left(t - t'\right)\right] dt' = -\frac{2}{\omega^3}, \tag{2.57}$$

we get

$$\widetilde{\zeta}(\mathbf{k}_{\parallel}, t) \approx -\frac{k}{\omega^2} \widetilde{P}_d(\mathbf{k}_{\parallel}, t) + \frac{2k}{\omega^4} \ddot{\widetilde{P}}_d(\mathbf{k}_{\parallel}, t). \tag{2.58}$$

It is now evident that the capillary wave motion is due to the acceleration of the DVP, i.e. $\ddot{\widetilde{P}}_d$. This term is dependent on $\dot{x}^n$ and $\dot{L}^n$, and could be found analytically. However, the time evolution is ignored in the implementation, which will be elaborated on in section 3.3. This is equivalent to freezing the system in time. The instantaneous surface deformation can thus be written

$$\widetilde{\zeta}_{inst} \approx -\frac{\widetilde{P}_d}{g + \gamma k^2}. \tag{2.59}$$

## 2.6 Surface Deformation Energy

The Hamiltonian from Equation 2.13 only considers the energy of the eddies in an isolated system. However, the eddies interact with the surface and deforms it, which requires energy, both as added potential energy, and surface tension energy. Because we are only looking at instantaneous surface deformations, mechanical energy of the surface can be neglected.

Surface tension energy is by definition proportional to the change in surface area as $E_{ST} = \gamma \Delta S$ [26], where

$$\Delta S = \int_{-\infty}^{\infty} \left[\sqrt{1 + |\nabla_{\parallel}\zeta|^2} - 1\right] d^2 x_{\parallel}. \tag{2.60}$$

We have already assumed linearized surface waves, which means that the surface is non-steep, i.e. $|\nabla_{\parallel}\zeta| \ll 1$, which means $|\nabla_{\parallel}\zeta|^2 \lll 1$ and thus

$$E_{ST} = \gamma \Delta S \approx \gamma \int_{-\infty}^{\infty} |\nabla_{\parallel}\zeta|^2 d^2 x_{\parallel}. \tag{2.61}$$

The potential energy associated with the surface deformation is simply the evaluated integral

$$E_P = \int_{-\infty}^{\zeta} \int_{-\infty}^{\infty} \rho g z dz d^2 x_{\parallel} = \frac{1}{2} \rho g \int_{-\infty}^{\infty} \zeta^2 d^2 x_{\parallel} \tag{2.62}$$

Combining, we get the total (kinematic) surface deformation energy

$$E_{surf} = \frac{1}{2} \int \gamma |\nabla \zeta|^2 + g \zeta^2 d^2 x_{\parallel}. \tag{2.63}$$

The surface deformation $\zeta$ is a function of the state of all the eddies. Adding this term to the Hamiltonian should then conserve the total energy of the system. Thus the Hamiltonian is

$$H = K_{self} + K_{int} + E_{def}. \tag{2.64}$$

In theory, this could be confirmed by integrating $H$ with respect to $x_i^n$ and $L_i^n$ and check if the system in Equation 2.12 is fulfilled. However, this is a cumbersome process, and it will therefore be an assumption that this system indeed is Hamiltonian.

## 2.7 Prior Distribution

The parameters yet to be determined are the impulse- and location vectors of the eddies. We now seek to find a joint PDF for these parameters. This is possible to do using statistical mechanics, where the energy of a state is associated with the probability of said state. However, the Hamiltonian describing the energy of the system contains both an interactive term and a surface deformation term, severely complicating the issue. Therefore, we momentarily ignore these terms and focus only of the $K_{self}$ term. Later, in chapter 4, the distribution found in this section will then be used as a *prior* distribution in a random walk algorithm to find a *posterior* distribution that includes the other terms. With these simplifications, the problem becomes very similar to the probability distribution of a system of particles, which has a well known solution from statistical mechanics. See for example *Mandl* for a full review [28].

We want to find the PDF of the state of the system, where the state is defined to be the set of all position- and impulse vectors for the system $\theta = \{\mathbf{x}^n, \mathbf{L}^n\}$ . Assuming isolated turbulence with no interaction makes the impulse independent of position. When isolated, the turbulence should be uniformly distributed, and only the impulse distribution is necessary to find such that $\theta = \{\mathbf{L}^n\}$. Assuming equilibrium statistical mechanics, the Boltzmann distribution associates the probability of a certain state to the energy of that state as follows

$$\pi(\theta) = \frac{1}{Z} e^{-\beta K_{self}}. \tag{2.65}$$

Here, $K_{self}$ is the total energy of the state and the "inverse temperature" of the system $\beta$ is a measure of the degree of chaos. Much like how temperature is a measure of the kinetic energy of a collection of particles, $\beta$ is here a measure of the average kinetic energy of the eddies.

$Z$ is a yet undetermined re-normalization constant, such that the total probability is unity

$$Z = \int \exp(-\beta \sum_{n=1}^{N} K_{self}^n) \prod_{n=1}^{N} d^3 L^n. \tag{2.66}$$

These 3N integrals decouple because of the multiplicative property of sums in exponentials, such that

$$Z = \left( \int d^3 L^1 e^{-\beta E_1} \right) \ldots \left( \int d^3 L^N e^{-\beta E_N} \right) \tag{2.67}$$

$$Z = \left[ \int e^{-\frac{\beta L^2}{2M}} d^3 L \right]^N = \left[ \sqrt{\frac{2M\pi}{\beta}} \right]^{3N} = Z_0^N. \tag{2.68}$$

The distribution from Equation 2.65 is then

$$\pi(\mathbf{L}^n) = \frac{1}{Z_0^3} e^{-\frac{\beta}{2M}\left(L_x^2 + L_y^2 + L_z^2\right)} = \pi^3(L_c). \tag{2.69}$$

It is now obvious that this is simply three independent normal distributions for the three spacial components in $\mathbf{L}$, with variance $\sigma_c^2 = M/\beta$ and mean $\mu_c = 0$. The second moment of the components are $\langle L_c^2 \rangle = \sigma_c^2$, as is well known for the normal distribution. Therefore utilizing the fact that average kinetic energy of the eddies is $\langle K \rangle = \langle L^2 \rangle / 2M$ and $\langle L^2 \rangle = \langle L_x^2 \rangle + \langle L_y^2 \rangle + \langle L_z^2 \rangle = 3\langle L_c^2 \rangle$ we get the useful relationship

$$\langle K \rangle = \frac{3}{2\beta}, \tag{2.70}$$

known as the equipartition theorem. The length of the impulse vectors $|\mathbf{L}|$ can then be found by integrating over spherical shells in which, resulting in the Maxwell-Boltzmann distribution

$$\pi(L) = \sqrt{\frac{2}{\pi}} \frac{L^2}{\sigma_c^3} e^{-\frac{L^2}{2\sigma_c^2}} \tag{2.71}$$

# Chapter 3

# Numerical Model

It is sought to implement the theory outlayed above in practice. This chapter will go through the choices made in order to numerically model the theory. Throughout this chapter it will be assumed that the location- and impulse vectors of the eddies are drawn from some PDF. This PDF is hitherto undecided, but the entirety of chapter 4 is dedicated to the topic. Note that in general the same model is followed throughout the report, but some alterations are made in that chapter for practical reasons.

## 3.1 Non-dimensional Form

To avoid numerical issues, all governing equations should be non-dimensional before they are implemented. To do this, some reference parameters are needed. Because only kinematic parameters are considered, all expressions can be rewritten in relation to a lengthscale $\lambda$ and a timescale $\tau$.

We can rewrite all dimensional quantities as a product of a non-dimensional quantity, and its reference parameters. For example a parameter $a$ is rewritten as $a = \hat{a}a_{ref}$, where the non-dimensional parameter is denoted with a hat. Because all reference parameters are combinations of $\lambda$ and $\tau$ means that rewriting all dimensional quantities as above and inserting into the equations of interest, yield the same equations only non-dimensional. The exceptions are the equations with dimensional constants, namely the dispersion relation, the instantaneous surface deformation and surface deformation energy, given from equations 2.46,2.59 and 2.63. Using $\omega_{ref} = 2\pi/\tau$, $k_{ref} = 2\pi/\lambda$ and following the procedure above gives the following non-dimensional versions of the equations;

$$\hat{\omega}(\hat{k}) = \frac{1}{Fr}\sqrt{\frac{1}{2\pi}\hat{k} + \frac{2\pi}{Bo}\hat{k}^3} = \sqrt{\frac{1}{We}\left(\frac{Bo}{2\pi}\hat{k} + 2\pi\hat{k}^3\right)}, \tag{3.1}$$

$$\hat{\hat{\zeta}} \approx -\frac{Fr^2}{1 + \frac{4\pi^2}{Bo}\hat{k}^2}\hat{\hat{P}}_d = -\frac{We}{Bo + 4\pi^2\hat{k}^2}\hat{\hat{P}}_d \tag{3.2}$$

and

$$\hat{E}_{surf} = \frac{1}{2}Fr^{-2}\int Bo^{-1}|\nabla_{\parallel}\hat{\zeta}|^2 + \hat{\zeta}^2 d^2\hat{x}_{\parallel} = \frac{1}{2}We^{-1}\int |\nabla_{\parallel}\hat{\zeta}|^2 + Bo\hat{\zeta}^2 d^2\hat{x}_{\parallel}. \tag{3.3}$$

Here, $Bo = g\lambda^2/\gamma$ is the *Bond* number, $Fr = u/\sqrt{g\lambda}$ is the turbulent *Froude* number and $We = u^2\lambda/\gamma$ is the turbulent Weber number. The Bond number describes the relationship between gravitational force and surface tension. In effect, it determines the length scale in reference to the capillary length, where gravity exactly balances surface tension. For water this is approximately $\lambda_{cap} \approx 2.7mm$ [29]. The Froude number describes inertial forces compared to gravity, and the Weber number describes inertial forces compared to surface tension. The inertial forces at play are that of the turbulence, not the of the surface itself, which is why they are called the *turbulent* Froude- and Weber numbers. They give relationship between the strength of the turbulence and the surface. Which one describes the flow properties the best is dependent on relative effect of surface tension and gravity, which will be explored further shortly in 3.2.

All that is left is to choose the scales $\lambda$ and $\tau$. It is desired to choose them based on the turbulence only, in order for the non-dimensional groups to represent meaningful relationships between the turbulence and the surface. The length scale is chosen to be $\lambda = M^{1/3}$ as the mass inertia on kinematic form represent some scale of the volume of an eddy. Next we want to choose a velocity scale for the turbulence in order the kinetic energy to be $\mathcal{O}(1)$. Assuming that the velocity fluctuations for the flow field $u'$ are of the same order of the velocity of the eddies themselves $u = \lambda/\tau$, the kinetic energy per unit mass is $\mathcal{O}(u^2\lambda^3)$. We already know the expected value of $K_{self}$ from Equation 2.70 which is $K_{self} \sim \beta^{-1}$. Relating the two gives a velocity scale $u = \sqrt{1/(\beta M)}$, and a corresponding time scale $\tau = \sqrt{M^{5/3}\beta}$. The non-dimensional groups thus become

$$Bo = \frac{gM^{2/3}}{\gamma}, \tag{3.4}$$

$$Fr = \frac{1}{\sqrt{\beta g M^{4/3}}} \tag{3.5}$$

and

$$We = \frac{1}{\beta\gamma M^{2/3}} \tag{3.6}$$

Having non-dimensionalized the equations with length- and velocity scales based on the turbulence means that $\hat{M} = 1$ and $\hat{\beta} = 1$ because $M$ and $\beta$ are the reference parameters. Note that we now have assumed $K_{self}$ to be the dominating term for choosing a velocity scale. This must be the case in order to maintain $K > 0$ due to the far-field approximation made by truncating after the dipole moment. This will be elaborated on further in sections 3.5 and 4.2. If interaction energy would be more dominant, for example by doing a new analysis with quadruple moments, it might be better to relate the length scale to the density of eddies instead which says more about average interaction lengths.

## 3.2 Test Matrix

Computational resources are limited, such that only a limited number of MCMC simulations can be performed, and sequentially only the surface statistics from these parameters can be evaluated. The goal is to find out how the system scales with the non-dimensional groups, and we should therefore choose parameters over a large range to capture the different effects of the system. Figure 3.1 shows

a velocity-scale length-scale plot of the relevant scales for NST. The shaded region is what *Brocchini & Peregrine* rougly estimated to be the region of marginal breaking where the surface characteristics are neither flat nor breaking [30]. This is the region of interest for the model used in this paper because we cannot go beyond wave-breaking due to the linear wave approximation, where it was assumed $|\nabla_{\parallel}\zeta| \ll 1$, and the surface cannot be too flat, as it leads to convergence issues with the MCMC simulations. The latter effect will be explained further in chapter 4. Furthermore, there is a limit to the Bond number we can choose, related to the width of our domain. This is because smaller Bond numbers means that the large surface tension effect gives large wavelengths which the finite domain of our choosing cannot capture. That is unless the domain is scaled accordingly large, which would require a finer discretization and thus more computational resources. The details of this limitation will be explained in section 3.6.

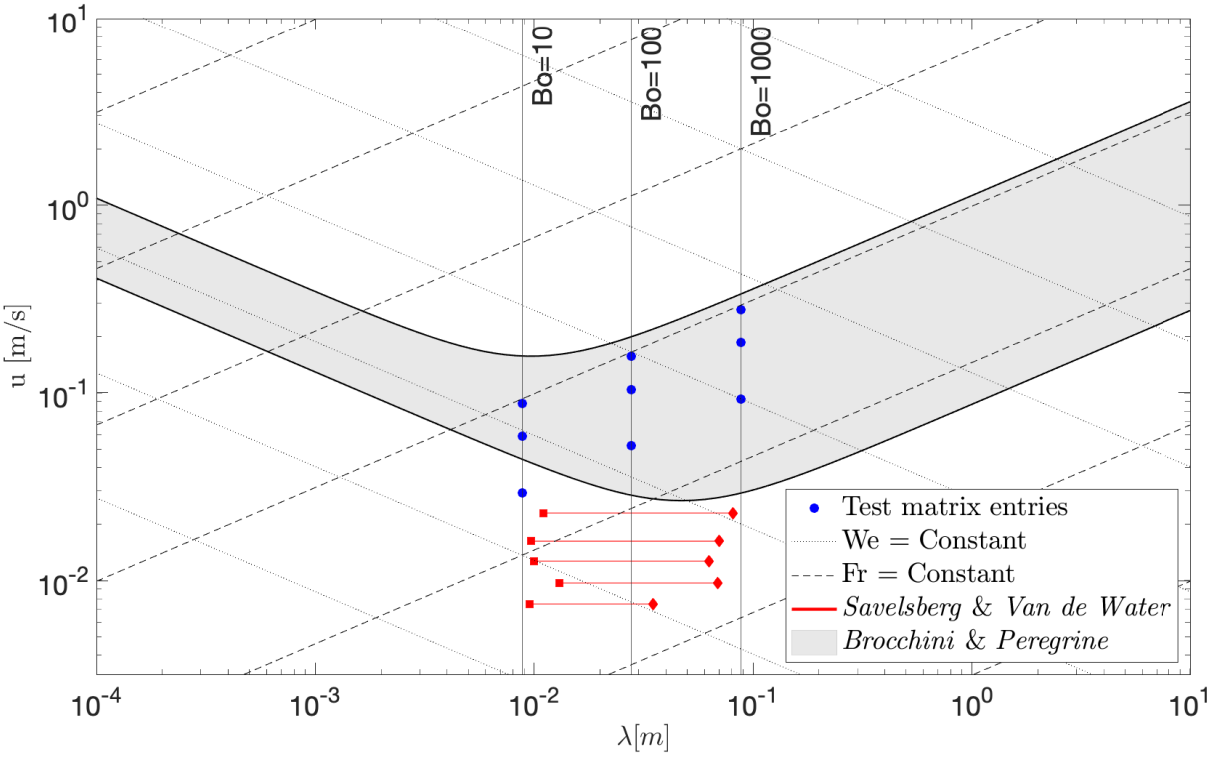



**Figure 3.1:** Length-scale velocity-scale phasespace plot with shaded regions indicating region of marginal breaking for FST as per *Brocchini & Peregrine* [30]. Dashed lines show constant Froude numbers and dotted lines show constant Weber numbers. Experiments from from *Savelsberg & Van de Water* [17] in red with range from Taylors microscale to integral scale. The simulations of this paper are represented as blue dots.

To get some indication to relevant length scales for FST we examine the experimental paper by *Savelsberg and Van de Water* [17], where different forcing protocols were used to generate homogeneous, isotropic turbulence. The Taylor microscales were then in the range $0.95-1.3$ cm, and integral scales ranging from $3.5-8.1$ cm. The model presented here has no viscosity, nor does it assume a length scale of the forcing. It is therefore logical to assume length scales in the inertial range between

these two scales. Choosing a typical length scale of the turbulence in this range gives Bond numbers $Bo \in [11, 729]$. Length-scales corresponding to $Bo = 10, 100$ and 1000 are chosen here to cover roughly the same range.

The left hand side of Figure 3.1 shows that the region of marginal breaking is dependent on $\lambda^{-1/2}$ i.e. $We$, while the right hand side shows a dependence on $\lambda^{1/2}$, i.e. $Fr$. The relevant length scales are in-between these regions of clear dominance and it is not obvious which parameter is most influential. It was chosen to vary the Froude number as this appears to have the widest region of marginal breaking at these length scales. The paper by *Savelsberg and Van de Water* also provide the corresponding turbulent fluctuation velocities to the given length scales; $u' \in [0.75, 1.27]$ cm/s, giving Fr numbers in the range $Fr \in [0.0081, 0.0415]$. These $Fr$ numbers give surface deformations too flat for the purpose for this report due to the aforementioned convergence issues. Values of $Fr = 0.1, 0.2$ and 0.3 were chosen to obtain a wide range of values while roughly staying in the region of marginal breaking.

## 3.3 Setup

The model described in chapter 2 is entirely analytic. The eddies can thus inhibit any point in $\mathbb{R}^3$, and are not limited to a numerical grid. Technically, this only holds down to the machine epsilon when implementing numerically, but that is still orders of magnitudes more accurate than what we require. We need to limit the extent of the eddies however to a finite domain, with width $w$ and height $h$, formally $\mathcal{D} = \{x, y \in [-w/2, w/2], z \in [-h, 0]\}$.

A 2D surface grid is defined in $z = 0$ with $GP$ gridpoints in each direction and resolution $\Delta x = w/(GP-1)$. Each point on this grid is also treated analytically and therefore no spacial discretization error is expected in these points. Inbetween points however, errors are to be expected.

### 3.3.1 Boundary Conditions

The choice of domain and surface grid is arbitrary, as it captures only a finite volume of a large sea. Eddies just outside of the domain might influence the surface deformation directly, or it might interact with an eddy inside the domain. To account for these effects, we utilize the Periodic Boundary Condition (PBC).

The surface deformation will automatically fulfill the PBCs due to the periodic nature of the Fourier transform [31]. However, the interaction does not and to account for this, copies of the eddies are made in 8 pseudo-domains around the main domain, including their respective mirror eddies. Far from all of these eddies will have any meaningful contribution for the interaction, and only those eddies within a Sphere Of Influence (SOI) of an eddy of interest $n$ will be included in the computation of $\dot{\mathbf{x}}^n$ and $\dot{\mathbf{L}}^n$. The radius of this SOI is found in section 3.5. Figure 3.2 shows the setup from above with the PBCs highlighted. Also a minimum interaction distance is established in section 3.4, which is shown in the figure.

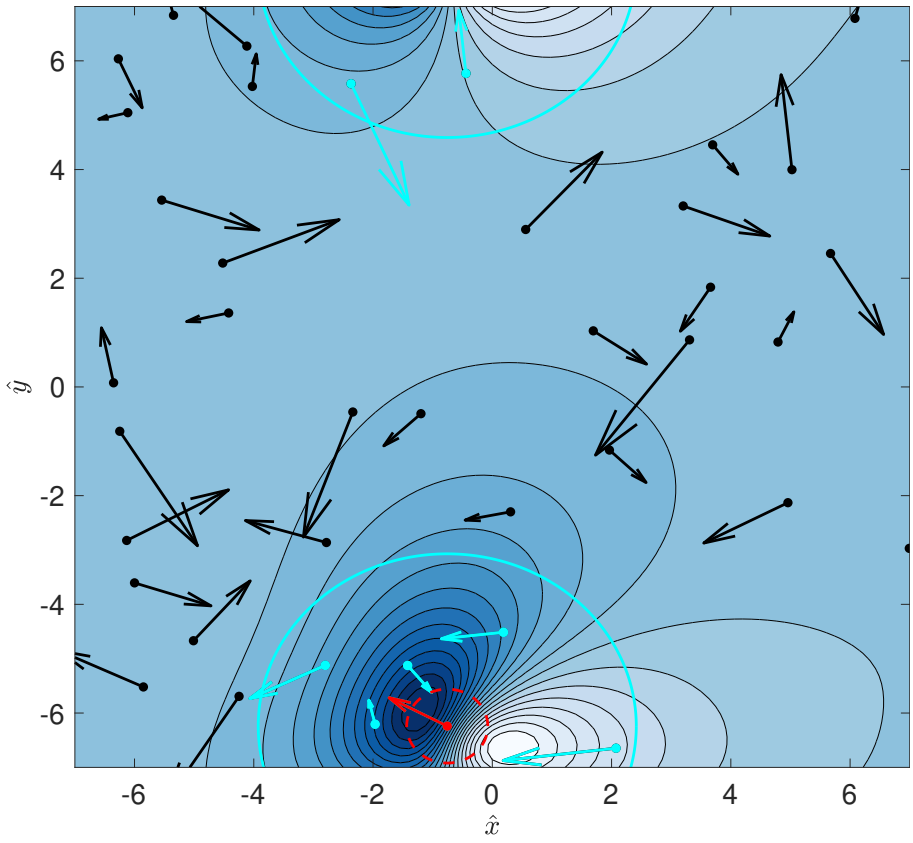



**Figure 3.2:** Setup as seen from above, demonstrating periodic boundary conditions and interaction. Eddies are depicted as dots and arrows, representing $\mathbf{x}^n$ and $\mathbf{L}^n$ respectively, superimposed on the surface. The contour plot shows the surface deformation from the eddy in red, and the sphere of minimum separation is shown as dashed red circle. The interaction sphere of influence for the eddy in red is shown as the circle in cyan and the interacting eddies within the sphere of influence are highlighted in cyan.

### 3.3.2 Statistical Surface Imprint Approach

As the Hamiltonian is given for the system of eddies, propagating the system forward in time should in theory be no issue. However, eddies that are to close to each might exchange an ever increasing amount of velocity and impulse, which can make the system blow up. This is a well known issue called "finite time blowup" and results from the lack of viscosity in the Euler equation which is being solved by the system [32]. Still, as long as the far-field approximation is retained, the Euler equations should be satisfactory. Therefore one must assure that the eddies are well separated at all times. When propagating the system in time, this is practically impossible to do. In section 2.2 a smoothing function was added to the propagators $\mathbb{C}$ and $\mathbb{D}$ in order to smooth out the effect for nearby eddies. During MCMC simulations, this kernel smoothing is essential to converge a solution, but during the surface statistics analysis we can avoid the issue altogether. Instead of numerically integrating the system in time, we rather generate a large number of systems frozen in time and calculate the instantaneous surface deformation. This approach is called ensemble averaging and each

system generated is called an ensemble. Provided enough ensembles are generated as to achieve a statistically steady state, this approach is equivalent to the time integration method [33]. If we make sure to generate each ensemble with enough spacing between eddies then the far-field approximation is guaranteed not to be violated as the next ensemble is generated independently. This spacing is the topic of section 3.4.

Even though the system will not be propagated in time, $\dot{\mathbf{x}}^n$ and $\dot{\mathbf{L}}^n$ is still needed, as they have an effect from the DVP. The time derivatives are affected by the surface, as $E_{surf}$ is included in the Hamiltonian. This will be ignored, and the time-derivatives from equations 2.14 and 2.15 are used. This is not expected to be a major issue, because the eddies are largely unaffected by the surface deep down whilst for shallower eddy depths, $P_{kin}$ will dominate due to its $r^{-6}$ dependence in regard to the surface. Had the time evolution been considered however, one would have to derive the Hamiltonian from Equation 2.64 according to Equation 2.12.

Ideally one would like to use a turbulence spectra closely resembling those found in nature, like for example a *Kolmogorov* $E(k) = k^{-5/3}$ spectra. However, our test matrix and thus PDFs in which to draw the eddies from only have three different length scales. Therefore, during the statistical analysis all eddies from the same simulation will be drawn from the same PDF distribution which means that the sea of eddies that is generated has the same length scale. In practice this means fixing the mass inertia as a constant for all eddies. The resulting energy spectrum should then be peaked around $k \sim \pi/\lambda$ [20].

## 3.4 Eddy Separation

It is important to separate the eddies sufficiently during the SDS simulations, as to not encounter blowup. However, nothing has been said so far about how large this separation distance must be. One constraint we can enforce is that the total kinetic energy of a system of isolated eddies far from a surface must be positive. The self energy is strictly positive, but the interaction energy can actually be negative. The reason for this is that eddies can cancel each others' velocity fields such that the system of eddies contain less energy than the eddies separately, i.e. separated an infinite distance from each other.

Consider two eddies separated at this critical separation distance $\delta$, antiparalell to each other, which is the orientation with the least interaction energy. The relationship between them is given by $L_2 = -cL_1$, where $c$ is the scaling factor between the two impulse vectors. Then from Equation 2.13 we have

$$K_{int}^{1,2} = \frac{-cL_1^2}{2\pi\delta^3} \tag{3.7}$$

and

$$K_{self}^{1,2} = \frac{L_1^2}{2M}\left(1+c^2\right). \tag{3.8}$$

By enforcing $K = K_{self} + K_{int} > 0$ we then get the inequality

$$\delta > \left(\frac{M}{\pi}\frac{c}{1+c^2}\right)^{1/3} \tag{3.9}$$

This function is maximized for $c = 1$, i.e. impulse vectors of equal magnitude $|L_1| = |L_2|$. Using this relationship gives

$$\delta > \left(\frac{M}{\pi}\right)^{1/3}. \tag{3.10}$$

On non-dimensional form this is a constant $\hat{\delta} > \pi^{-1/3} \approx 0.683$ because $\hat{M} = 1$ with the chosen length scale.

This does only take into account two eddies. However adding more eddies will increase $K_{self}$ linearly with the number of eddies, but the interaction energy will not. Say another eddy is added. The only orientation in which adding an extra eddy will give $K_{int}^{1,3}$ as large as $K_{int}^{1,2}$ is placing it on the line spanned by $L_1$ and $L_2$ in $\mathbf{x}_3 = -\delta\mathbf{e}_x$ with $L_3 = -L_1 = L_2$. Then $\mathbf{r}^{2,3} = 2\delta$ and the interaction energy is $K_{int}^{2,3} = K_{int}^{1,3}/8$. This pattern repeats for adding additional eddies.

Note that even though this $\delta$ ensures that the total kinetic energy is always positive, it does not tell us whether the far-field approximation holds. For this one would need to compare with quadrupole- and higher order moments and see at what distance they deviate significantly. For now, this $\delta$ will be used as a requirement for spawning the eddies in relation to each other when looking at the surface statistics. For the Monte Carlo simulations, a similar method will be used, explained further in section 4.2.

## 3.5 Interaction Length

A vast amount of the computational effort is devoted to calculating the pressure interaction terms, as these terms are essentially $\mathcal{O}(N^2)$. However, these effects are largely affected by only the eddies closest grouped together. Therefore it is desirable to find a maximum length $r_c$ in which eddies that are separated more than this distance would not contribute significantly to the interactive pressure terms. From Equation 2.36 we have

$$P_{int,vel}^n = \mathbf{u}_p^n \cdot \dot{\mathbf{x}}^n = \mathbf{u}_p^n \cdot \left(\sum_{m \neq n} \mathbb{C}(\mathbf{r}^{mn}) \cdot \mathbf{L}^m\right). \tag{3.11}$$

It is obvious that just like the energy, introducing more eddies only reduces the relative interaction pressure compared to self pressure, because it is the *total* induced velocity vector that matters to $P_{int}$. Thus, the induced velocity from one eddy might cancel some of the velocity from another. The maximum relative contribution to the DVP is therefore a system of two eddies, separated by the vector $r\mathbf{e}_{m,n}$, where $\mathbf{e}_{m,n}$ is the unit vector of a line through the two eddy centers. Like in section 3.6, a fractional pressure is introduced, comparing $P_{int,vel}$ to the maximum value, where two eddies are separated by $\delta$;

$$P_f^n = \frac{\mathbf{u}_p^n \cdot \mathbb{C}(r_c\mathbf{e}_{m,n}) \cdot \mathbf{L}^m}{\mathbf{u}_p^n \cdot \mathbb{C}(\delta\mathbf{e}_{m,n}) \cdot \mathbf{L}^m} = \frac{|\mathbb{C}(r_c\mathbf{e}_{m,n}) \cdot \mathbf{L}^m|}{|\mathbb{C}(\delta\mathbf{e}_{m,n}) \cdot \mathbf{L}^m|}, \tag{3.12}$$

where the vector $\mathbf{u}_p^n$ is independent on the interaction. The fractional pressure is then dependent on the direction of $\mathbf{e}_{m,n}$ as the propogator $\mathbb{C}$ is slightly anisotropic. Isosurfaces of the vector magnitude $|\mathbb{C}(r_c\mathbf{e}_{m,n}) \cdot \mathbf{L}^m]$ are ellipsoids, with the principal axis parallel to $\mathbf{L}^m$ slightly shorter than the others.

For the purposes of this analysis, they will be assumed spherical such that scaling arguments give $\mathbb{C}(\mathbf{r}^{mn}) \sim L^m (r^{mn})^{-3}$. Inserting this and solving for $P_f = 0.01$, which is considered negligible, we get $r_c = \delta / 0.01^{1/3} \approx 3.17$.

It should be noted that this analysis has been done with the interaction velocity term only. However, doing a similar analysis with the impulse term and cancelling the $\mathbb{B}^n \cdot L^n$ terms in its equivalent to Equation 3.12, the remaining contributing term scales with $\sim (r^{nm})^{-4}$, which result in a shorter $r_c$.

## 3.6 Domain Proportions

To capture all the essential details of the surface deformation, it is important to have a surface domain wide enough that the surface deformation does not interact considerably with its own image through PBCs. How large this is depends on the depth of the deepest eddy, as the surface deformations become less and less localized the deeper an eddy is submerged. This section aims to find a semi-analytic relation for the width to height ratio of the domain to satisfy this condition. It is difficult to find an analytical expression of the surface elevation, because one needs to go through the Fourier domain. Therefore we focus the first part of this analysis on the surface pressure. Because the surface deformation is only a smoothed out version of the DVP for the instantaneous surface deformation, the general shape will be the same, which means we can extrapolate some useful information used to do a similar analysis for the surface deformation. In section 3.4 it was argued that the interaction energy must be smaller than the self energy of the eddies to retain $K > 0$. It follows then that the interaction pressure must be smaller than the self pressure. This section therefore focuses solely on $P_{kin}$ and $P_{self}$.

There is only one degree of freedom for the pressure terms for an eddy with a fixed distance to the surface. That is the angle of its impulse vector with the $x-y$ plane. Let's name this angle $\alpha$. Now consider an eddy directly beneath the origin with impulse in the $x-z$ plane. The impulse vectors for an eddy and its mirror are then $\mathbf{L}^n = [L\cos(\alpha), 0, L\sin(\alpha)]$ and $\mathbf{L}^{\overline{n}} = [L\cos(\alpha), 0, -L\sin(\alpha)]$, where $\overline{n}$ denotes the mirror to eddy $n$. Also let's denote the distance to the surface $s$, such that $\mathbf{x}^n = [0, 0, -s]$ and $\mathbf{x}_{\overline{n}} = [0, 0, s]$ and the relative position vector between a point on the surface $p$ with coordinates $[x, y]$ and the eddies are respectively $\mathbf{r}^{np} = [x, y, -s]$ and $\mathbf{r}^{\overline{n}p} = [x, y, s]$. The pressure field terms along the x-axis evaluated with Equation 2.35 are then

$$P_{kin,x}^{n+\overline{n}} = \frac{\left[L\cos(\alpha)(2x^2 - s^2) + 3xsL\sin(\alpha)\right]^2}{8\pi\left(s^2 + x^2\right)^5}, \tag{3.13}$$

and

$$P_{self,x}^{n+\overline{n}} = \frac{s^3}{2}\frac{\left[\cos^2(\alpha)(2x^2 - s^2) + 6xs\cos(\alpha)\sin(\alpha) + \sin^2(\alpha)(2s^2 - x^2)\right]}{(x^2 + s^2)^{5/2}}. \tag{3.14}$$

The maximum for these expression are easily found by evaluating $\partial_\alpha P_{self}^{n+\overline{n}} = 0$ and $\partial_\alpha P_{kin}^{n+\overline{n}} = 0$. They are respectively

$$P_{kin,max}^{n+\overline{n}} = P_{kin,x}^{n+\overline{n}}(x = 0, \alpha = 0) = \frac{L^2}{8\pi^2 s^6} \tag{3.15}$$

and

$$P_{self,max}^{n+\overline{n}} = P_{self}^{n+\overline{n}}(x = 0, \alpha = \frac{\pi}{2}) = \frac{L^2}{M\pi s^3}. \tag{3.16}$$

Note that $P_{self,max}^{n+\overline{n}}$ and $P_{kin,max}^{n+\overline{n}}$ equal at $s = (M/8\pi)^{1/3}$, where the maxima of the two terms are in equilibrium. Below this value, $P_{self}$ is the dominating term, and above it, $P_{kin}$ is dominating. This depth will be called $s_{eq}$ for future reference.

Normalizing Equation 3.13 and Equation 3.14 with their respective maxima from Equation 3.15 and Equation 3.16 means getting rid of the dependence on the magnitude of the impulse $L$. Then it is of interest the distance of which the normalized pressure will be at 5% of its maximum, which here is considered an acceptable cutoff criterion. Unfortunately, $P_x/P_{max} = 0.05$ is not easily solvable analytically for the two cases. However, the solution can be approximated in *Matlab* by varying $\alpha$ and using *fzero* [34], essentially giving a numerical expression for $x_{max}(\alpha)$. The max value of $x_{max}$ is found to be for $\alpha_{max} \approx \pi/4 + \pi q$ for the kinetic term and $\alpha_{max} \approx 11\pi/90 + \pi q$ for the self motion term, where $q$ take integer values. With these $\alpha$-values, there are several solutions, all with linear dependence of $s$. The ones with the largest slopes are respectivly $x_{max} \approx 1.74s$ and $x_{max} \approx 2.52s$. Having only examined the one-dimensional case in the x-direction is no guarantee that other directions does not give a larger maximum displacement from the origin off course. However, the x-direction was chosen because it showed the largest displacements, and it is easily confirmed visually. We have then found a circle with radius as a function of eddy depth s in which 95% of the DVP is concentrated, regardless of the $x - y$ orientation of the eddy.

Doing the same analysis for the surface elevation is trickier, as mentioned. We do the same procedure only directly evaluating both terms simultaneously with the numerical procedure explained above, and once again using *fzero* to find where the surface elevation crosses our threshold of 5%. The surface elevation is now calculated with the normalization constant is $\zeta_{max} = \zeta(\alpha = \pi/2, x = 0)$ which comes from the fact that the maxima of the surface elevation will be at the same angle and surface coordinates as for the DVP, only smoothed. Likewise, $\alpha_{max} = 11\pi/90$ during this analysis as this produces the largest $x_{max}$ for the DVP.

The surface deformation transfer function defined in Equation 3.2 is dependent on gravity and surface tension. The gravity only scales the pressure, while surface tension also smooths it out. It is therefore expected that the shorter the length scale $\lambda$ we have, i.e. lower Bond number, the larger the domain we would require. Figure 3.3 shows the effect of varying $Bo$ on the required width of our domain. It is clearly visible that for $Bo \to \infty$, the result is the $2.52s$ dependence found for the pressure analysis. This is unsurprising because $Bo \to \infty$ removes the surface tension smoothing effect and varying the $Fr$ does not scale the normalized surface deformation, such that we obtain the results from above. Increasing the surface tension, an ever increasing domain width is required. Note that for shallow eddy submergence there is a steep incline in required domain width. This is due to the $r^{-6}$ dependence of the kinetic pressure term in this region. This can safely be ignored because surface deformations as steep as these require an enormous amount of energy, which is extremely unlikely. This will be discussed further in chapter 4.

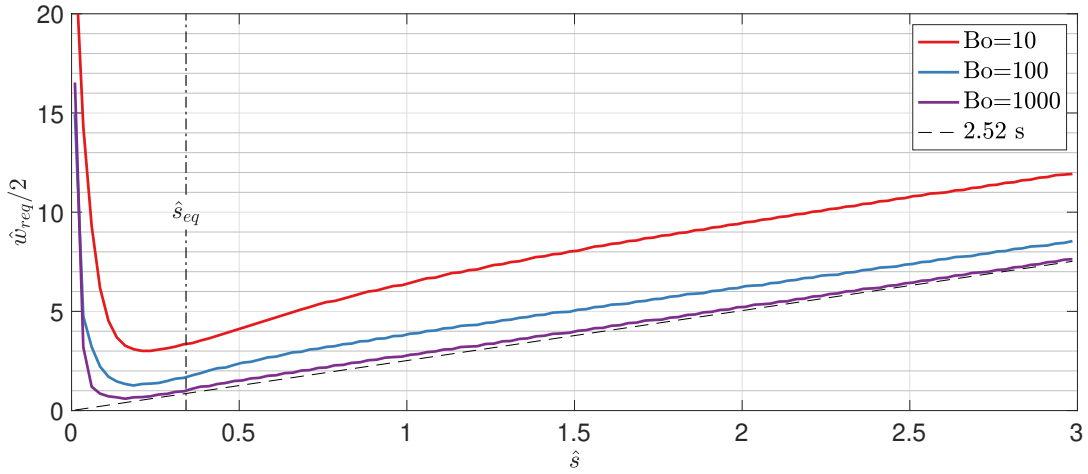
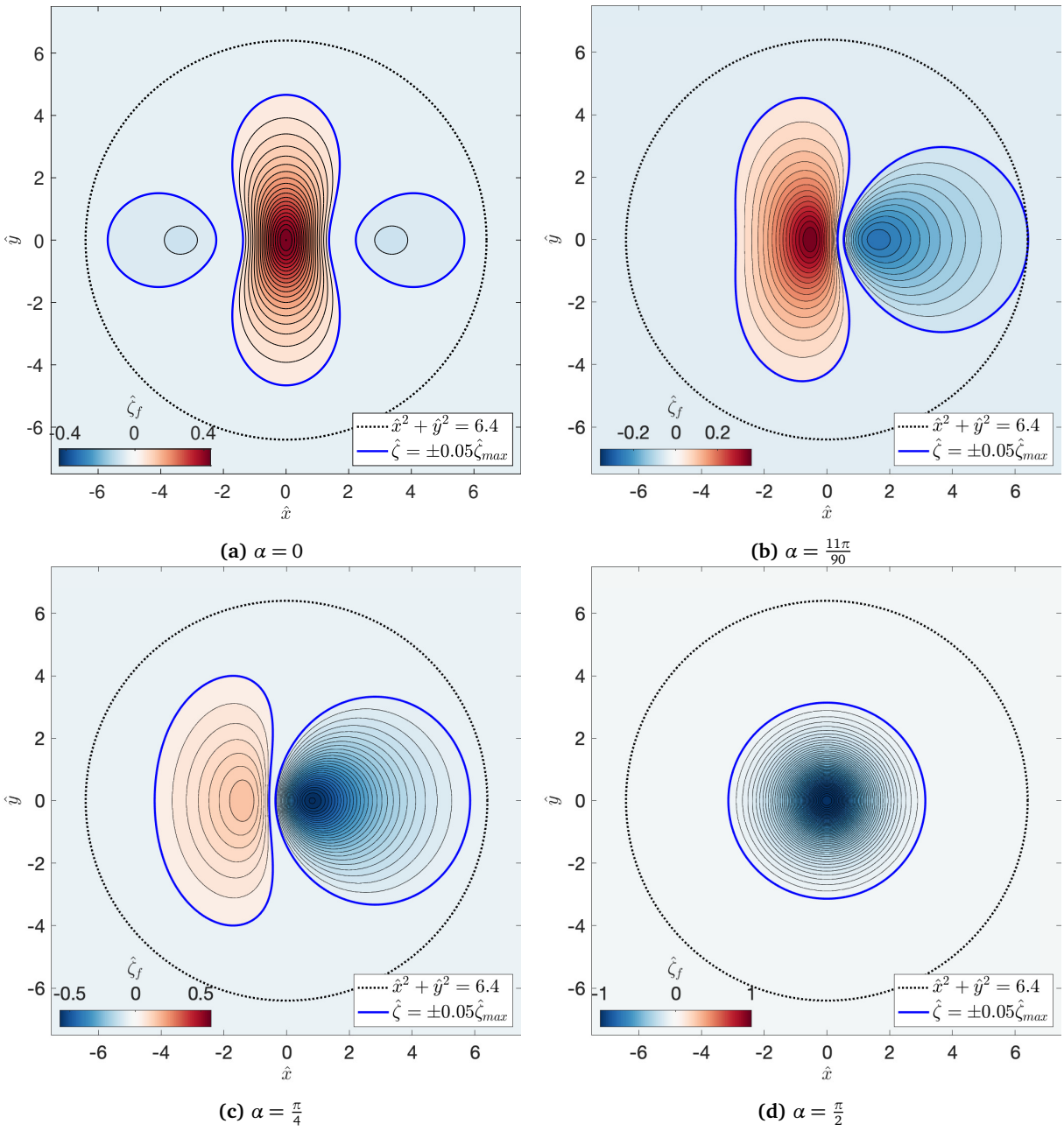



**Figure 3.3:** Domain width required to capture the 95% highest surface elevations as a function of eddy submergence s for different combinations $Bo$. The dashed line represent the linear relationship with the slope found from a similar analysis witht the DVP

Now the issue of low Bond numbers raised in section 3.2 is clearer, as the extra smoothing lengthens the wavelengths of the surface deformation, requiring a larger domain to capture. There is a significant drawback of having a large domain. This is because a larger domain require more gridpoints to aquire the same resolution. The computation time scales with $GP^2$ and it is therefore desirable to limit the domain width as much as possible. Through trial and error, a domain height $h = 1$ was chosen as this shows to capture the essential depth dependence fairly well. It is also far deeper than $s_{eq}$, such that $P_{self}$ is dominant, meaning the turbulence at this depth should be approximately independent of the surface.

The required domain width from Figure 3.3 is $w_{req} = 2x_{max}(s = h) = 12.8$. This is rounded up to 14 as to avoid errors from surface deformations below 5% adding up at the boundaries due to PBCs. Figure 3.4 shows that the fractional surface deformations $\zeta_f = \zeta/\zeta_{max}$ are indeed bounded to below 5% within a circle with diameter 6.4 as calculated. Note that the mean surface deformation is non-zero due to the deformations below the cut-off criterion adding up at the boundaries. This is not an issue however, as it is the higher order statistical moments that are interesting for the SDS simulations.




**(a)** $\alpha = 0$ **(b)** $\alpha = \frac{11\pi}{90}$

**(c)** $\alpha = \frac{\pi}{4}$ **(d)** $\alpha = \frac{\pi}{2}$

**Figure 3.4:** Surface elevation from an eddy and its mirror normalized by the maximum value for different impulse angles $\alpha$. Blue contour shows where 5% of maximum surface elevation is, and black dotted circle bounds this for all $\alpha$.

## 3.7 Grid Dependence

To make sure numerical discretization errors are kept to a minimum, a grid dependence study is performed. Recall that all gridpoints have an analytic solution for $\zeta$, while errors are present in-

between. The parameter of interest will be the surface deformation energy, as it sums up all these errors over the surface.

Figure 3.5 shows the surface energy from a single eddy and its mirror, plotted for some different eddy depths and for the different Bond numbers of our test matrix. This is done at an angle of $\alpha = \pi/2$, because this is the orientation with the most surface deformation as well as having the strongest gradients $\nabla_{\|}\zeta$, meaning it is the most prone to error.

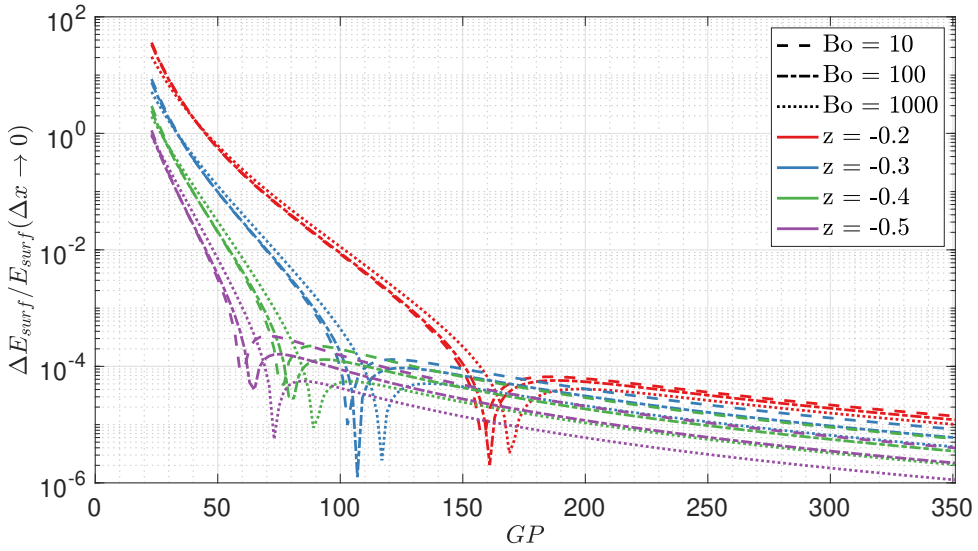



**Figure 3.5:** Grid dependence of surface deformation energy with the given domain width for four different eddy depths and the Bond numbers from the test matrix.

The dependence on $Bo$ is visible, as higher Bond numbers means higher surface gradients and thus more error. The dips in the curves is simply a result of overshooting, making an impression of low error on the way over the line. It is clear that the closer an eddy is to the surface, the more discretization errors there will be. This is due to the $r^{-3}$ and $r^{-6}$ dependencies of the DVP. The error will be infinite for $z \to 0$ and we can therefore only set a limit on discretization error *below* a certain threshold of depth.

During MCMC simulations, computing the surface grid is without a doubt the clear bottleneck. Then, as we will see in chapter 4, computing the energy is essential. Computing self energy and interaction energy needs only be done at the eddy center, a single point in $\mathbb{R}^3$ for each iteration. The surface deformation energy needs to be computed at all $GP^2$ gridpoints, in addition to then numerically computing the gradient field and integrate. The computation time is on the order of days, and we are therefore forced to accept a relatively coarse resolution. It was decided to accept surface discretization errors up to 5% below a depth of $z = 0.2$. Using 81 gridpoints in each direction will then be satisfactory. When examining the final surface statistics, we can allow for a higher resolution, as the computation is more efficient. Then a 1% error is accepted up to $z = 0.2$, corresponding to 101 gridpoints.

An idea that was explored to ease computational cost was to calculate the surface deformation in a relatively rough grid and interpolate in-between. The idea was however disgarded because the

added computational cost of interpolating coincidentally turned out to cancel the benefit for the specific number of gridpoints chosen here. It might be worth investigating further in the future if other domain sizes or error thresholds are chosen.

# Chapter 4

# Markov Chain Monte Carlo

The statistical description in section 2.7 assumed no interaction energy between eddies, nor surface deformation energy, for the sake of simplicity. It is desirable to perform a similar analysis with these effects accounted for, but the integrals will not decouple and are difficult, if not impossible to solve analytically. MCMC is a family of computational methods to circumvent these integrals. The idea is to "walk" through phase space using a Markov Chain and approximate the target distribution with the samples collected, like one does in Monte Carlo simulations. By assigning a state of the system with a probability, the chain spends the most time in high probability state areas, so-called *typical sets* [35]. These regions will thus be sampled more frequently, in accordance to the probability distribution. MCMC methods are often used in Bayesian statistics to find a probability distribution for high dimensional state-spaces in this manner.

The distribution from section 2.7, i.e. $L_c \sim \mathcal{N}(0, \sqrt{\frac{M}{\beta}})$ and $x_c \sim \mathcal{U}(-d_z, 0)$ will be used as the *prior* distribution, a first guess in the MCMC methods in order to find more likely states of the system including surface deformation energy and interaction energy. The state of the system $\theta$ is defined as a point in the $6N$ dimensional phase-space containing position and momentum vectors for all $N$ eddies, i.e. $\theta = \{\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_N, \mathbf{L}_1, \mathbf{L}_2, ...\mathbf{L}_N\}$. The distribution we would like to sample, the *posterior*, is in our case only $4N$-dimensional, because the $x-y$ plane is arbitrarily defined. Eddies should therefore have no preference to their position in $x$ or $y$, nor should $\mathbf{L}$ be dependent on them. One should neither expect any preference in $L_x$ and $L_y$. Technically one could thus combine these two impulse components to a single parameter and perform the simulations in the 3-dimensional phase-space, but as a redundancy measure we rather use this to evaluate convergence.

Ideally, one would like to do the MCMC simulations with several eddies to get a full picture. However, each eddy added to the simulation adds 6 dimensions to step around in phase space, which dramatically slows down computational efficiency. Therefore, only one eddy plus its mirror is regarded during this MCMC analysis. The valid information lost in this simplification will be the surface deformation energy due to the interaction term of the DVP. However, as we saw in section 3.4, the interaction pressure terms must be smaller than the kinetic- and self term under the dipole model as to assure positive kinetic energy. What *will* be included is the interaction between an eddy and its appropriate mirror, as this can impact both the depth distribution $p(z)$ and the impulse distribution $p(\mathbf{L}|z)$.

## 4.1 Random Walk Metropolis

Random Walk Metropolis (RWM) is the oldest and simplest of the MCMC algorithms, introduced in the classical paper by *Metropolis et.al.* [36]. The algorithm is designed to suggest a new state and accept the step based on the probability associated with it.

An eddy is drawn from the prior distribution. Then a step to a new proposed state $\theta_*$ is suggested with a jumping distribution $J(\theta_*|\theta)$. This jumping distribution must be symmetric, i.e. $J(\theta_*|\theta) = J(\theta|\theta_*)$. To meet this demand, a normal distribution centered arund the current state is used $J(\theta_*|\theta) \sim \mathcal{N}(\theta, \Sigma)$. This jumping distribution is in theory 4-dimensional, but each component is independent, i.e. the covariance matrix $\Sigma$ for the 4 variables is purely diagonal. So in practice each proposed component is drawn from its marginal normal distribution, $J(z^*|z) \sim \mathcal{N}(z, \Delta z)$ and $J(L_c^*|L_c) \sim \mathcal{N}(L_c, \Delta L)$. The probability of the step $p(\theta_*)$ is then calculated with regards to the energy of the system, i.e. the Hamiltonian from Equation 2.64 for the proposed state compared to the current state as

$$p(\theta_*) = \frac{e^{-\beta H_*^n}}{e^{-\beta H^n}} = e^{-\beta \Delta H}, \tag{4.1}$$

where

$$\Delta H = \Delta K_{self} + \Delta K_{int} + \Delta E_{def}\,. \tag{4.2}$$

The step is then accepted with a probability $p(\text{accept}) = min[1, p(\theta_*)]$, assuming the step is legal. A legal step in this context means that the eddy remains in the domain. The process is then repeated many times, until we have reached a statistically steady state for the parameters of interest. The algorithm tailored to this paper is summed up in Algorithm 1.

**Algorithm 1** Random Walk Metropolis

1: Draw $\theta$ from $\pi(z, \mathbf{L})$
2: **for** $it = 1 : iterations$ **do**
3: Draw $\theta_*^n$ from $J(\theta_*|\theta)$
4: $\theta_*^{\overline{n}} \leftarrow$ mirror$(\theta_*^n)$
5: Update $\dot{x}^n$,$\dot{L}^n$ with Equation 2.14 and Equation 2.15
6: $P_d^n \leftarrow$ Equation 2.36 with $\mathbf{u}_p^{tot} = \mathbf{u}_p^n + \mathbf{u}_p^{\overline{n}}$
7: $\zeta^n \leftarrow$ Equation 2.59
8: $\Delta H \leftarrow$ Equation 4.2
9: **if** $\mathcal{U}(0,1) < e^{-\beta \Delta H}$ && $\mathbf{z}_* \in \mathcal{D}$ **then**
10: $\theta^n = \theta_*^n$
11: **end if**
12: **end for**

## 4.2 Constraining Interaction Energy

During MCMC simulations, one cannot use the separation distance found in section 3.4. This is because the eddy position changes during the simulation based on the current step. Because $\delta$ is a discrete cutoff-value one would lose all information about the posterior distribution above $z = -\delta/2$.

Still, for the Monte Carlo simulations not to diverge, it is essential to bound the kinetic energy $K > 0$. The reason is that by design, any step in the direction of less energy $\Delta K < 0$ is accepted, which is clear from Equation 4.1. An eddy has an opposite $L_z$-component to its mirror, an orientation with negative interaction energy. A step in phase-space will therefore further minimize this energy by either increasing the magnitude of the $L_z$ component, or by decreasing the distance to the surface. The former means that $K \to -\infty$ for $L \to \infty$, meaning the simulations diverge, in addition to being a clear violation of physics. Therefore we must ensure that the self energy is at least as large as the minimum value for the interaction energy, in order for the total kinetic energy to always be positive, such that $K \to +\infty$ for $L \to \infty$. Instead of separating the eddies with a minimum distance, we utilize kernel smoothing, as described in section 2.2 and bound the eddy radius $R$.

With smoothing, the minimum interaction energy configuration is an eddy on the surface with an impulse vector in the $z$-direction, such that the mirror is anti-parallel. This gives

$$K_{int,min}^{n+\bar{n}} = \mathbf{L}^n \cdot \breve{\mathbb{C}}(\vec{0}) \cdot \mathbf{L}^{\bar{n}} = -\breve{C}_{33}(\vec{0})|L^n|^2 \tag{4.3}$$

The corresponding self energy to the same configuration is $K_{self}^{n+\bar{n}} = |L^n|^2/M$. Setting $K_{tot} > 0$ results then in the inequality

$$R > \begin{cases} \left(\frac{M}{2\pi}\right)^{1/3} & \text{for } f_1(r), f_2(r) \\ \left(\frac{5M}{4\pi}\right)^{1/3} & \text{for } f_3(r), f_4(r) \end{cases} \tag{4.4}$$

Because we would want to retain the far-field approximation as well as possible, smoothing function $f_2(r)$ will be used for the simulations. It is the one approaching the far-field the quickest, while also limiting the size of the eddies by Equation 4.4.

Figure 4.1 show a particular segment of iterations for a typical MCMC simulation with the interaction energy bounding method. It is evident that whenever the $z$-component of the impulse vector start to grow, and correspondingly the interaction energy begins to diverge, the self energy grows in the positive direction and keeps the total energy above zero. This shows that the self energy successfully bounds the interaction energy with the given smoothing radius.

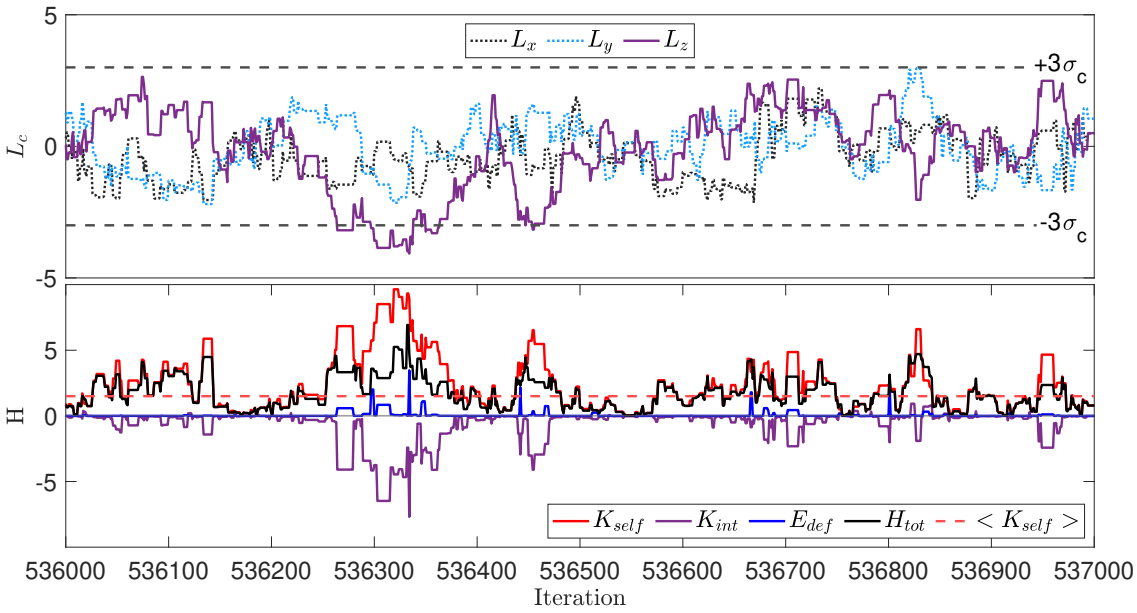



**Figure 4.1:** Demonstrating effect of bounding interaction energy for a typical simulation. Top figure showing impulse components over a segment of iterations. Bottom figure showing terms of the Hamiltonian for the same iterations.

## 4.3 Parameter Tuning

For the RWM algorithm we have yet to tune the covariance matrix of the jumping distribution, which in practice is the two parameters $\sigma_{\Delta z}$ and $\sigma_{\Delta L}$. Setting these parameters too small will lead to slow exploration of phase space, resulting in a need for more iterations. Larger steps will converge faster toward the typical sets, but will tend to get stuck there because the steps away will be very unlikely. Finding this balance is called the *optimal scaling problem* [37]. *Gelman et. al.* found that the optimal acceptance rate to be about 44% for one dimension, and about 23% for the infinite dimensional problem [38]. The optimal acceptance rate for this 4-dimensional problem should then be somewhere in-between these extremes.

By trial and error, it was found that $\sigma_{\Delta x} = 0.5h$, $\sigma_{\Delta L} = 0.5\sigma_c$ generally produces consistent acceptance ratios in this range. Table 4.1 shows the acceptance ratio of the simulations performed. All the simulations ended up in the desired range, meaning that the chosen $\sigma_{\Delta z}$ and $\sigma_{\Delta L}$ are satisfactory for fast convergence.

| Acceptance Ratio | | | |
|---|---|---|---|
| $Fr$ \ $Bo$ | 10 | 100 | 1000 |
| 0.1 | 0.371 | 0.375 | 0.370 |
| 0.2 | 0.375 | 0.372 | 0.363 |
| 0.3 | 0.375 | 0.369 | 0.357 |

**Table 4.1:** Acceptance ratio at final iteration for RWM simulations with different parameters

## 4.4 Convergence

Judging convergence of MCMC methods is by no means trivial. One approach is to monitor a parameter of interest until it does not longer oscillate, and settles down to sort of a steady state situation. *Muller* found this approach to be satisfactory despite its simple nature [39]. Here, this means monitoring the maximum change in $z$ for $p(z)$ and $p(\mathbf{L}|z)$, where the latter we monitor the Standard Deviation (SD) of the distribution. For simplicity, only $\Delta\sigma_{|\mathbf{L}|}$ is shown, as this incorporates all the errors from each impulse component. In Figure 4.2 we see that the maximum change in the sampled $\Delta p(z)$ decays smoothly and flattens out. The same applies for $\max(\Delta\sigma_{|\mathbf{L}|})$, albeit in a noisier manner.

The simulations in the figure are grouped by color and color intensity, where the lower Froude number simulations have a lighter color. Note that it is these simulations which show the greatest changes, and are the slowest to converge. This is because the weaker surface allows eddies closer to itself, as well as larger impulse vectors. Hence there is a larger phase-space to explore and less samples are recorded in each state. The flatter the surface characteristics, the more difficult it is to converge the MCMC simulations. This is the reason it proved disadvantageous in section 3.2 to use simulations below the region of marginal breaking.

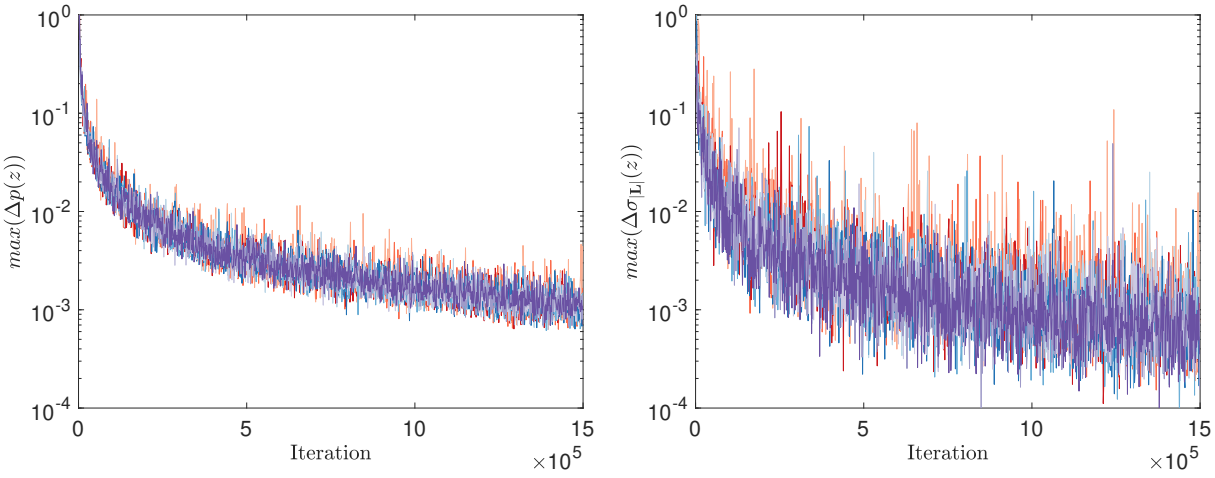



**Figure 4.2:** Monitoring change in the maximum change over $z$ in sampled parameters of interest $p(z)$ and $\sigma_{|\mathbf{L}|}(z)$ for an increasing number of iterations for all 9 simulations in the test matrix.

In addition to monitoring the convergence of the parameters of interest, it is crucial to check that

the assumptions we have made are correct in order to be confident of convergence. Firstly we must make sure that the Markov Chain has forgotten all about the initial conditon, since this was randomly chosen. A way to do this is to check the AutoCorrelation Function (ACF) to see how correlated the Hamiltonian is to itself from previous iterations. Figure 4.3 depict a typical ACF for the simulations, where it is obvious that the correlation becomes insignificantly low after the initial period of about 150 iterations.

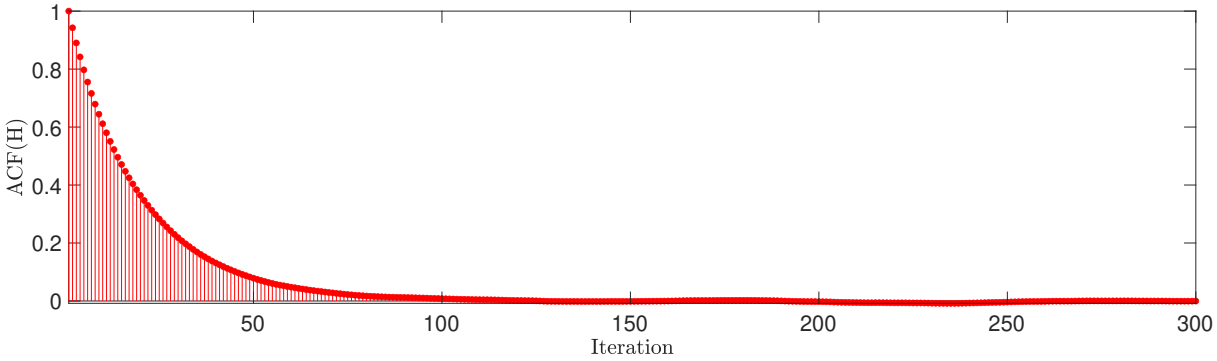



**Figure 4.3:** Autocorrelation function of the Hamiltonian for a typical simuation. Here, the first 300 iterations are shown to demonstrate the burn-in period.

Secondly we must check that the assumption of independent $L_x$ and $L_y$ components holds. For a simulation to have converged to a statistically steady state one would expect $\text{Corr}(L_x, L_y) = 0$. We cannot use the Pearsons correlation coefficient as per usual, because it assumes that the parameters constitute a bivariate normal distribution [40], while our distribution is $z$-dependent. Instead, the *Spearman* correlation coefficientis used, as it does not assume a probability distribution [40].

Figure 4.4 shows how the Spearman correlation evolves with the number of iterations. All simulations stabilize around a correlation coefficient of less than 0.025. By itself, this value does not mean much, and it would therefore be desirable to perform a hypothesis test to confirm whether or not these correlations are statistically insignificant. However, such a test requires independence of observation [41]. A Markov chain is by its very nature dependent on some earlier iterations. The ACF for $L_x$ and $L_y$ shows that they have completely "forgotten" about their previous states after about 250 iterations. In evaluating the p-value therefore, we only use every 250th iteration of the simulation to calculate the Spearman correlation coefficient. The p-values of the simulations for the final iteration are summarized in Table 4.3. They are computed with a 95% confidence interval. All of them are comfortably above the significance level of 0.05, meaning we can reject the null-hypothesis that $L_x$ and $L_y$ are correlated. It is important to note that there is always a chance of type $II$ errors, so-called false negatives [41].

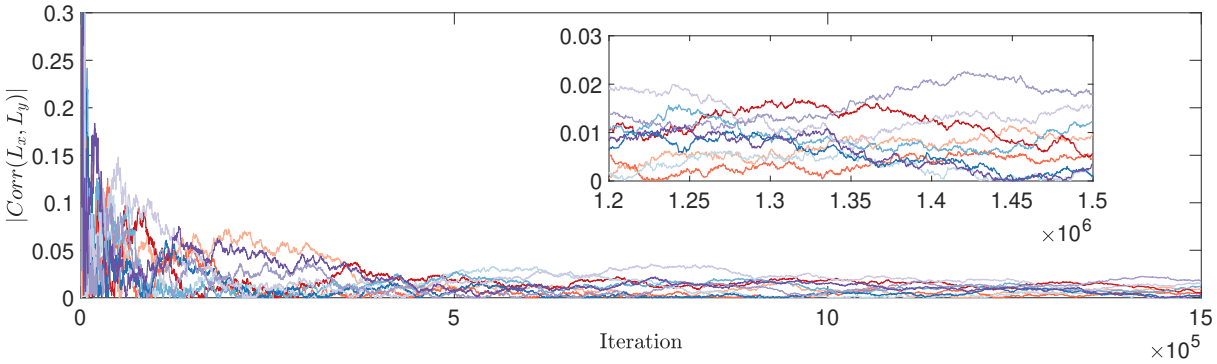



**Figure 4.4:** Evolution of absolute value of Spearman correlation between $L_x$ and $L_y$ for an increasing number of iterations. Subfigure shows a zoomed in version of the final 300,000 iterations

| $Corr(L_x, L_y)$ | | | |
|---|---|---|---|
| $Fr$ \ $Bo$ | 10 | 100 | 1000 |
| 0.1 | -0.010 | 0.001 | 0.016 |
| 0.2 | 0.005 | -0.012 | -0.018 |
| 0.3 | -0.006 | 0.001 | -0.004 |

**Table 4.2:** Spearman correlation between $L_x$ & $L_y$ at final iteration for RWM simulations

| p-value | | | |
|---|---|---|---|
| $Fr$ \ $Bo$ | 10 | 100 | 1000 |
| 0.1 | 0.454 | 0.952 | 0.228 |
| 0.2 | 0.727 | 0.338 | 0.167 |
| 0.3 | 0.666 | 0.956 | 0.781 |

**Table 4.3:** p-value of Spearman correlation between $L_x$ and $L_y$ at final iteration for RWM simulations with 95% confidence interval

## 4.5 A Note on Hamiltonian Monte Carlo

It would be interesting to examine a system with more eddies in order to include interaction. Then however, RWM would probably be insufficient. Due to the randomness in the jumping distributions, the RWM algorithm is known to struggle in high dimensional cases, as the number of directions in phase space to step in increases [35]. It is therefore advantageous to use a method that glides smoothly through the typical sets of high probability. This is the idea of Hamiltonian Monte Carlo (HMC). Because the Hamiltonian is conserved during a trajectory through phase space, it will move on a hypersurface of constant probability density. Another major flaw in the RWM algorithm is that it does not account for the dynamics of the system. This will tend to propagate the system to a lower energy state regardless of the physics involved.

In theory, HMC solves both of these problems. A full review of the algorithm is outlayed in *Neal* [42]. In short, the algorithm updates only the impulse variables independently of the current impulse. Then the system is propagated in time with step-size $\Delta t$ for $S$ number of steps using $\dot{x}$ and $\dot{L}$ from the Hamiltonian dynamics. The final step on this trajectory is then the proposed new state $\theta_*$, however with flipped sign for the impulse as to keep the proposal symmetric. Then a Metropolis update is done, like in the RWM method outlayed above. HMC is expectedly also easier to tune, as the timestep $\Delta t$ can be fixed to yield an acceptable error, such that only $S$ is tuned.

Whilst the algorithm looks promising for the present case, one would need to overcome one important obstacle in addition to finding $\dot{\mathbf{x}}$ and $\dot{\mathbf{L}}$ including $E_{surf}$. The numerical integration method used to update the system must be *symplectic* in order to remain stable for Hamiltonian systems [42]. This means it should preserve volume in phase space under integration, like the Hamiltonian equations do. When constructing a symplectic integrator it is common to separate the Hamiltonian into terms containing each of the canonical vectors. The Hamiltonian examined here has an interaction term *and* a surface deformation term, both of which are not separable. *MacLachlan & Atela* presents some implicit methods to generate symplectic integrators for non-separable Hamiltonians [43]. However, it requires high order derivatives of Equation 2.64, which is cumbersome to do analytically. Nevertheless, with the benefits of faster convergence of higher dimensional probability distributions, it is undeniably worth the effort for further work on the topic.

# Chapter 5

# Numerical Implementation

This chapter will look at how the numerical model in chapter 3 was implemented in *Matlab*. Firstly, section 5.1 explains the class structure of the code. Then some non-trivial details about the code implementation is discussed in section 5.2. Lastly the efficacy of the code is discussed in section 5.3, alongside some efficiency improvements. The main scripts are attached in Appendix A, with the necessary classes, scripts and functions.

## 5.1 Class Structure

In the code, there are many variables to keep track off. The object oriented functionality in *Matlab* was used to better keep track. The code uses three main classes; Eddy, Turbulence and SurfaceSection. The Eddy class contains the properties defining a single eddy; $\mathbf{x}_n$,$\mathbf{L}_n$,$\dot{\mathbf{x}}_n$,$\dot{\mathbf{L}}_n$, as well as a reference to a SurfaceSection object. Note that $\hat{M}$ is set as constant in the class with value 1 because of the way the system is non-dimensionalized. In the future if more length-scales were present, it could easily be set as a class variable instead. The Eddy class also contains useful member-funtions. Among other things these functions can produce a mirror eddy, calculate surface pressure terms evaluated at the SurfaceSection grid, calculate the energy terms and calculate interaction properties given other Eddy objects.

The Turbulence class contains a collection of Eddy objects as well as overall turbulence parameters common to all eddies, namely $\delta$,$r_c$, $R$, smoothing function number and domain. No smoothing is equivalent to specifying 0 for the smoothing function, such that the same layout can be used with or without smoothing. All the eddies are collected in a matrix of Eddy objects, where the first row contains the real eddies in the domain, the next 8 rows contains the copies around the domain to fulfill PBCs, and the rest are the respective mirror eddies of the first 9 rows of eddies. This way it is easy to keep track of what copies and mirrors correspond to which eddy. The copies and mirrors are made upon construction of the object, using the domain parameters and the mirror function of the Eddy objects. The Turbulence class contains member-funtions that can easily sum up the $P_d^n$ terms or Hamiltonian terms of the eddies. The class can also update $\dot{x}^n$ and $\dot{L}^n$ for all eddies at once, with a member-funtion that uses the SOI method outlayed in section 3.3.

The SurfaceSection class contains the gridpoints for the 2D surface grid, both in real space and in wavenumber space. How the wavenumber grid is made is described in subsection 5.2.2. The class also

contains the surface parameters $Bo$ and $Fr$ such that a SurfaceSection object can calculate surface deformation and surface deformation energy with a given DVP from the Eddy or Turbulence class.

All the member variables of the SurfaceSection class remain constant during each SDS and MCMC simulation. However, all Eddy objects need to access the SurfaceSection object to calculate the DVP and surface deformation. To avoid unnecessarily many copies of this SurfaceSection object to be stored in memory, the SurfaceSection class is set to be pass-by reference, which in *Matlab* means deriving from the *handle* class [34]. It could be useful in the future to define also the Eddy class to be pass-by-reference, especially if propagating the system forward in time, which would eliminate the need to make new objects every time the state is changed. With the ensemble averaging approach presented, the eddies only change state during MCMC, and using the Eddy class as pass-by-reference then surprisingly slowed computation by a little margin.

There is a large interplay of member-funtions and member variables which all communicate and it is thus hard to keep track of and update all member variables. Therefore, many of the variables are set to be *dependent properties*, which means the variables are "pulled" by a predefined get-function when the parameter is requested. For example, when calculating the kinetic term of the DVP, the induced velocity components from an eddy at the surface are needed in all $GP^2$ gridpoints, as a cell array $\mathbf{u}_p^n$. To compute this cell array, another cell array is needed in turn, the vectors describing the position vector from each gridpoint to the eddy $\mathbf{r}_p^n$. Say an eddy is moved, as is the case during MCMC simulations. Without dependent variables one would need to update these arrays sequentially before computing the DVP components of interest, allthewhile needing additional variables to keep track of which variables have already been updated from calls in other functions. With dependent properties, all the necessary variables are updated automatically when called.

## 5.2 Code Implementation

### 5.2.1 Computing Dynamic Vortex Pressure

Only the self motion term in Equation 2.36 is completely independent of other eddies, as the kinetic term needs the *total* induced velocity at the surface from all eddies, and the interaction terms depend on induced velocity and momentum. $P_{self}^n, P_{int,vel}^n \& P_{int,imp}^n$ are set as dependent properties for the Eddy class, with the requirement that $\dot{x}^n$ and $\dot{L}^n$ are updated. This requirement is checked with some boolean values, which will be described in subsection 5.2.4. $P_{kin}^n$ is calculated using a separate function with a cell array $\mathbf{u}_p^{tot}$ containing the total surface velocity from all other eddies, as an input. If an eddy is considered in isolation, then this is simply a cell array filled with the zero-vector. In the case of the Turbulence class, $P_{kin}$ is set as a dependent property because $\mathbf{u}_p^{tot}$ is itself a dependent property in the class, which is computed by summing up the $\mathbf{u}_p^n$ contributions from all eddies with a Turbulence class get-function.

When evaluating the pressure terms, one needs to compute the propagators $\mathbb{C}$ for all points on the surface, and additionally $\mathbb{B}$ and $\mathbb{D}$ for the interaction terms. Instead of computing all these propagators, storing them in cell arrays and then iterate through points on the surface, it is both convenient and computationally more efficient to use the built in *Matlab* function *cellfun*. This way, all elements of the $\mathbf{r}_p^n$ cell array are subject to the same function, taking in a vector and outputting the scalar pressure value after evaluating the tensors in a function called within *cellfun*. Because $\mathbf{r}_p^n$ is a dependent property, it is pulled whenever the pressure terms are called, such that the pressure

is guaranteed to use the updated state of the eddy. For example, calculating $P^n_{self}$ can be evaluated in a single line of code as such

```
function P_SELF_n = get.P_SELF_n(obj)
    P_SELF_n = cellfun(@(r_np) (1/obj.massInertia)*dot(obj.imp,C_point(r_np)*obj.imp
        ') , obj.eddy2surfaceVecs);
end
```

### 5.2.2 Computing Surface Elevation

Having $P_d$ evaluated over the surface grid, the theory from section 2.5 can be utilized. However, first we need to map the spacial coordinates into k-space coordinates. The number of gridpoints will remain the same as in the spacial domain, but the spacing $\Delta k$ and limits will obviously not. We have a spacial resolution to sample wavelengths $l$ down to $\Delta x$. To avoid aliasing effects, the wavenumber $k = \frac{2\pi}{l}$ should be at a maximum half the sampling wavenumber. This is known as the *Nyquist sampling theorem* and the maximum wavenumber is analagous to the Nyquist frequency in the time-frequency domain [44]. Thus the maximum wavenumber is $k_{max} = \frac{\pi}{\Delta x}$. With the same number of gridpoints $GP$, the k-space resolution becomes $\Delta k = \frac{2\pi}{GP\Delta x}$.

With the proper k-space grid, $k = \sqrt{k_x^2 + k_y^2}$ can be evaluated over the grid, which is done upon construction of the SurfaceSection object. Note that the fraction term in Equation 2.59 is a transfer function, completely independent of any of the eddies. It is therefore defined to be a dependent property matrix in the SurfaceSection class. A member-function in the SurfaceSection class can thence take in the DVP from an Eddy or Turbulence object and return the corresponding surface elevation. This is done by doing a 2D FFT of the DVP, piece-wise multiplying it with the transfer matrix, and returning the inverse 2D FFT of this matrix product.

When the surface elevation is calculated, it is fairly straight forward to compute the surface energy. The gradient vector field is calculated with the built in *Matlab*-function *gradient*, then Equation 3.3 is evaluated and integrated numerically using the *trapz* function in both dimensions.

### 5.2.3 Generating Turbulence From a Posterior Distribution

The sampled posterior distribution for our simulations is essentially a collection of vectors for $z$ and $\mathbf{L}$, for a long list of iterations. For each ensemble, a list of random indices are generated, corresponding to the number of eddies in the main domain. Then a vector of empty Eddy objects is created, which is to be filled with eddies containing the $z$ and $\mathbf{L}$ values corresponding to the random indices. Recall that the eddies must be separated by at least $\delta$ from each other. The $x-y$ coordinates, drawn from a uniform distribution are therefore generated within a while-loop, until this separation condition is maintained in accordance to all other eddies in the Eddy-vector. Only then is the Eddy object constructed and added to the Eddy-vector. When all entries are filled, a Turbulence object is created, which automatically generates all copies and mirrors in the class constructor.

### 5.2.4 Updating Interaction Properties

The interaction variables for the Eddy class $\dot{\mathbf{x}}^n$ and $\dot{\mathbf{L}}^n$ cannot be set as dependent, because they are functions of properties of other Eddy objects. Because the interaction variables are outdated whenever an eddy changes its state, two boolean variables are introduced to tell when an eddy is in its updated state. To switch these variables off when updating the state of the eddy, all updates need to be performed in a custom update-function, instead of the pre-defined set-operator "=". The access of the state variables are set to private in order to assure this operator is not used. All eddies may be updated at once with the Turbulence object member-funtion *updateInteractionProperties()*. This function loops through each Eddy object in the main domain and sums up the induced properties from all other Eddy objects within its SOI, using equations 2.14 and 2.15. Then the copies and mirrors of the updated eddies are also updated.

## 5.3 Computational Efficiency

We have already made some simplifications in the numerical model to achieve shorter run-times. This included using a relatively course grid resolution, as well as using only interaction between eddies within a SOI. Additionally, some time saving methodologies could be used during the *implementation* of the code without loss in accuracy.

### 5.3.1 Parallelization

In order to best utilize the computational power, it is desirable to parallelize the code in order to run loops on multible cores simultaneously. This requires that the loops are completely independent of each other. Because Markov chains in their very nature depend on the iteration before, it was not possible to run the MCMC simulations in this manner. The SDS simulations however are run over ensembles that are generated completely independently of each other. The Parallel Computing Toolbox was used for this, as *Matlab* only utilizes a single computer core by default.

### 5.3.2 Vectorization

*Matlab* is optimized for operations involving matrices and arrays [34]. There are therefore possibilities of gaining computational efficiency by replacing for-loops with vectors and matrices, which is a process called *vectorization*. This can be done in the present case by using array multiplication instead of index-notation.

It was mentioned above how the propagation tensor $\mathbb{C}$ must be calculated in each gridpoint on the surface for each eddy when evaluating all the terms of the DVP. Additionally, $\mathbb{B}$ and $\mathbb{D}$ must be calculated when including the interaction pressure. The evaluation of these propagators is the process taking the longest to compute, and a small gain in efficiency for computing these might therefore greatly boost performance. Calculating and using this on array form instead of with index notation in for-loops saw a 40% and 30% boost in calculating $\mathbb{B}$ and $\mathbb{C}$ respectively. The former propagator is evaluated directly as a vector, but for the latter we must first compute the outer tensor product $r_i r_j$ in matrix form, i.e. $\mathbf{r}\otimes\mathbf{r}$, before performing the desired calculation. This premeasure slightly counteracts the gain from vectorization, but the gain is clearly higher than the loss. This is however not the case for $\mathbb{D}$, because both the tensor product $r_i r_j r_k$ as well as the three Kronecker delta functions $\delta_{ij}$,

$\delta_{ik}$ and $\delta_{jk}$ needs to be defined in 3D arrays before computing the desired propagator. Doing this annuls the gain in vectorization, and the propagator calculations is thus left in index notation. Also the pressure terms are computed in vector form for the same purpose, using Equation 2.36 instead of Equation 2.35.

### 5.3.3 Performance

The computational resources available was a 12-core computer with Intel Xeon Gold 5218 processors running at 2.30GHz. The MCMC simulations were run simultaneously in separate *Matlab* programs, each simulation using a single core. The running time for the $1,500,000$ iterations that it was run for took about 5 days. The SDS simulations which were run in parallel took about a day and a half, simulating 20 main eddies for 5000 ensembles. These run-times are *after* the aforementioned simplifications and time-saving methodologies, which goes to show the importance of optimizing efficiency.

# Chapter 6

# Results

## 6.1 Posterior Distribution

Figure 6.1 shows $p(z)$ in the posterior distribution for the different simulations. The simulations are grouped in color by $Bo$ and in $Fr$ by color strength, which is the trend throughout this chapter. All parameters are non-dimensional, such that the hat notation is dropped in this chapter.

Note that all the PDFs experience a "buffer zone" near the surface where the probability of finding an eddy is zero. They also have a peak more or less around $s_{eq}$. The lower Froude number simulations have more distinct peaks, and they are situated at slightly lower depths. The effect of the Bond number is a larger buffer zone and less distinct peaks.

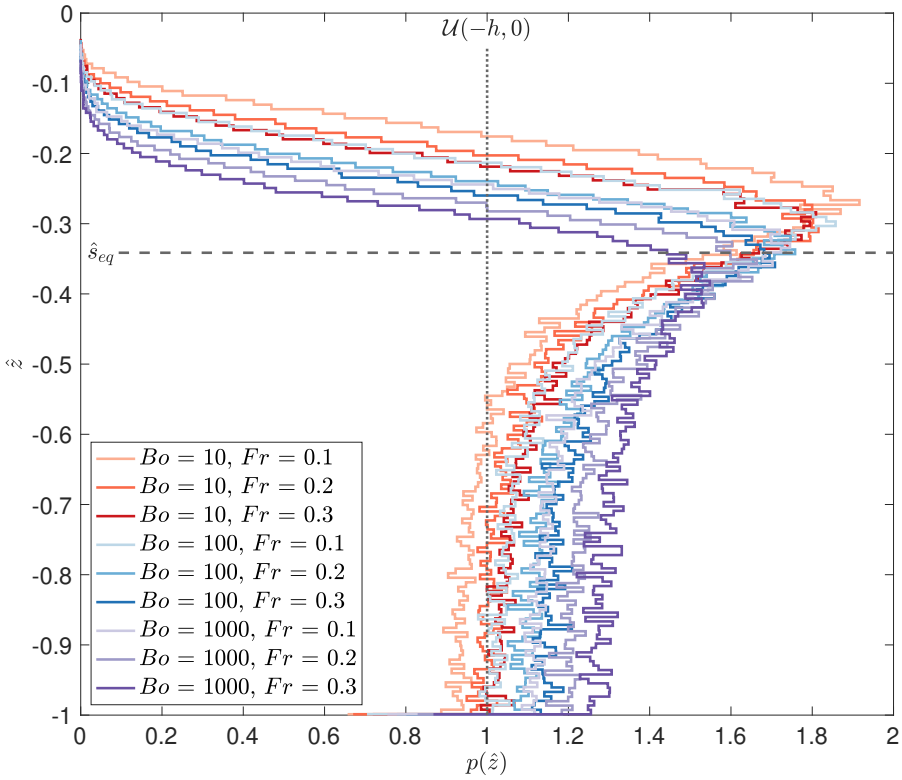



**Figure 6.1:** Normalized frequency distribution histograms of sampled $p(z)$ for the different simulations parameters in the test matrix. The dotted line is the uniform distribution for the spacial prior distribution, and the dashed line is the equilibrium depth discussed in section 3.6

Figure 6.2 and Figure 6.3 gives an overview of the sampled $p(L_\perp, z)$ and $p(L_\parallel, z)$ distributions for all the simulations as 2D normalized histograms. $L_\parallel$ is the horizontal impulse component parallel to the surface and $L_\perp$ is the vertical component normal to the surface. For each slice in $z$, the impulse component distributions appear to be normal distributions centred in zero for both cases. It is obvious that the shape of the histograms are quite dependent on $Fr$ and $Bo$ for the $p(L_\perp, z)$ case, whilst in the $p(L_\parallel, z)$ case there are no clear qualitative differences. To distinguish the simulations in a clearer manner, the SD of $p(L_\perp|z)$ and $p(L_\parallel|z)$ are plotted in Figure 6.4, denoted by the shorthand notation $\sigma_\parallel(z)$ and $\sigma_\perp(z)$. Because there are far less sample points for eddies close to the surface, the SD will not be statistically meaningful there, and thus only those $z$-bins with more than 100 samples are considered.

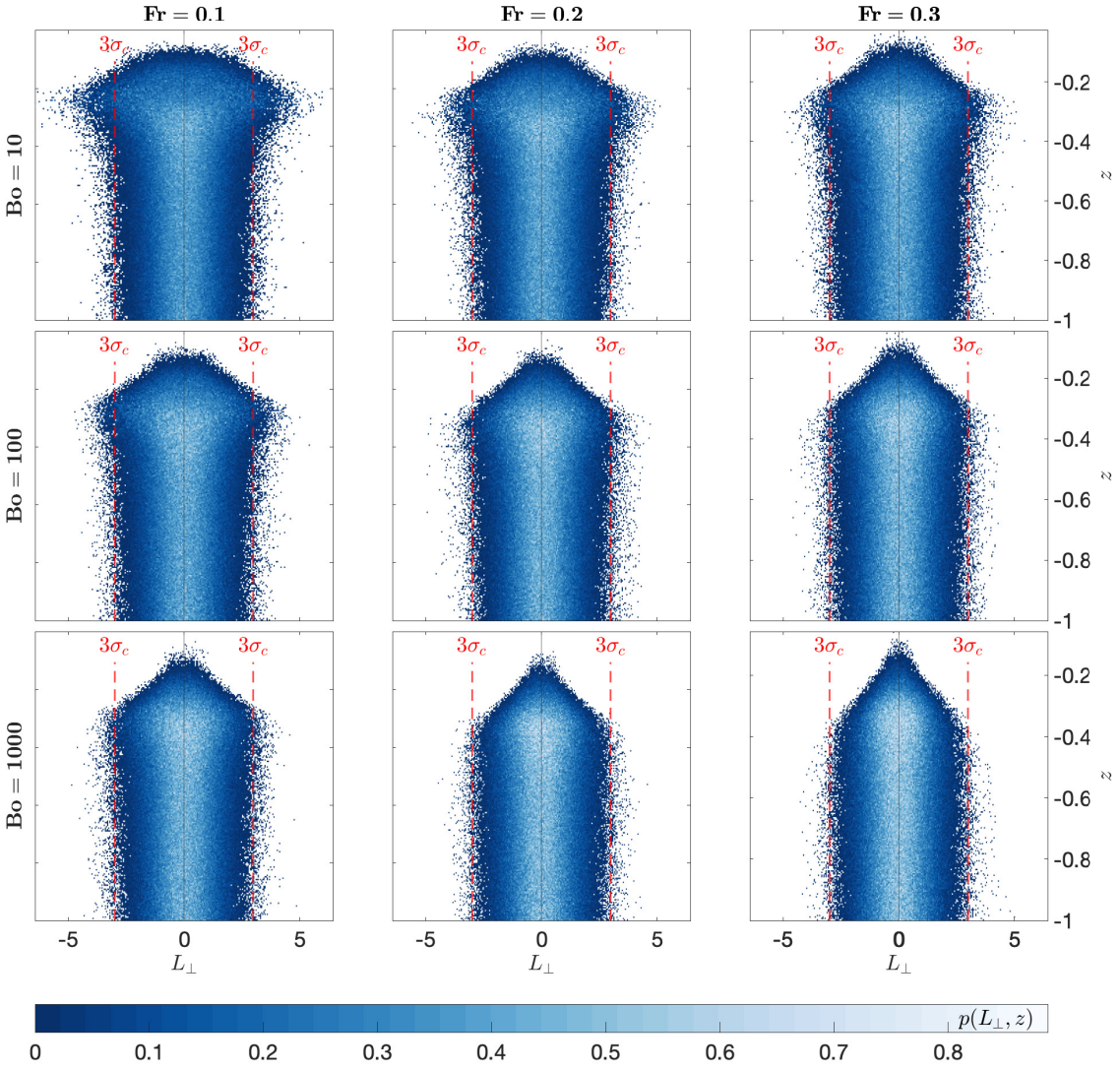



**Figure 6.2:** Two-dimensional normalized histograms of the $p(L_\perp, z)$ distributions for the different simulations. The red dashed lines show the 99.7% confidence interval of the prior distribution.

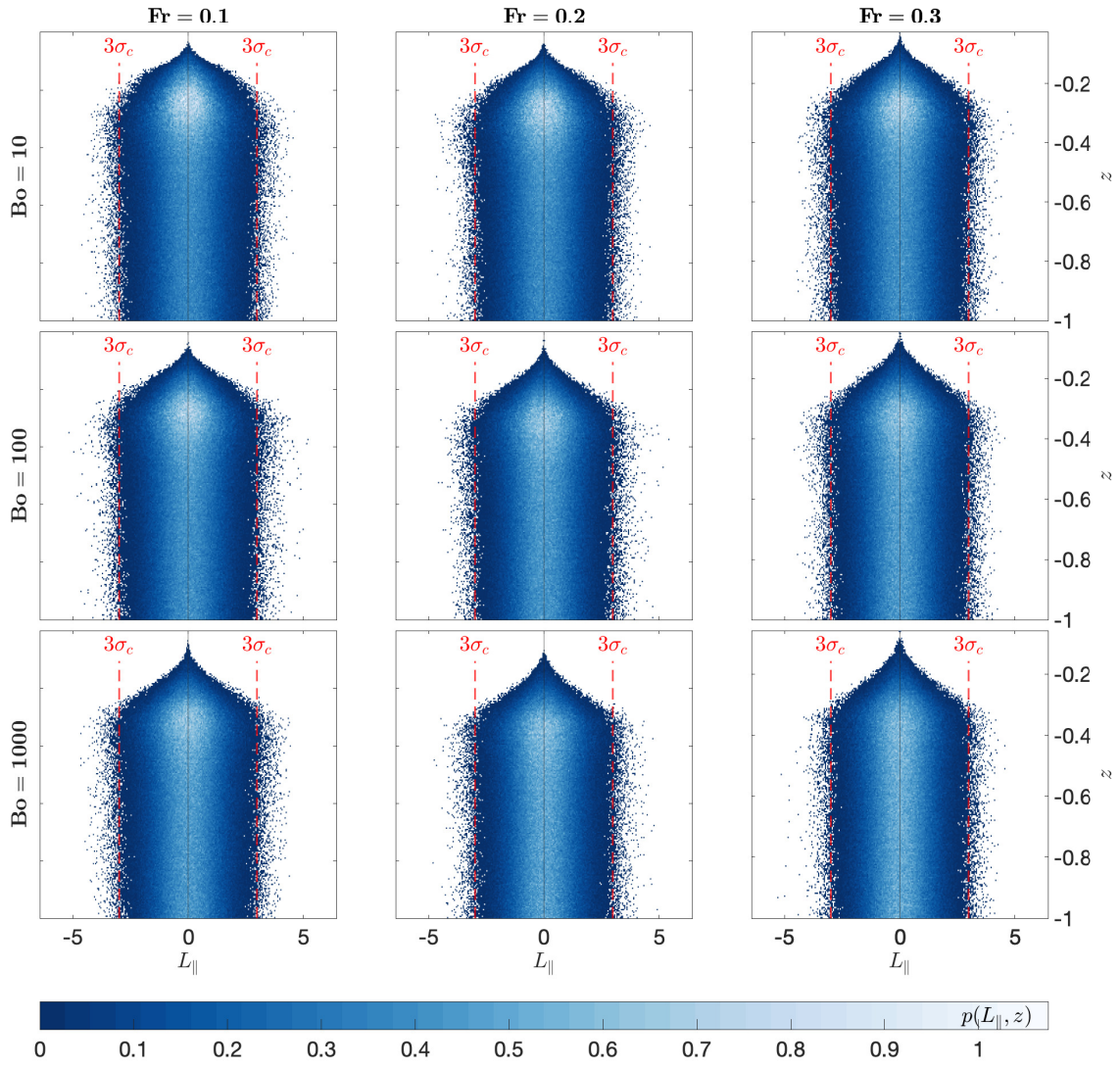
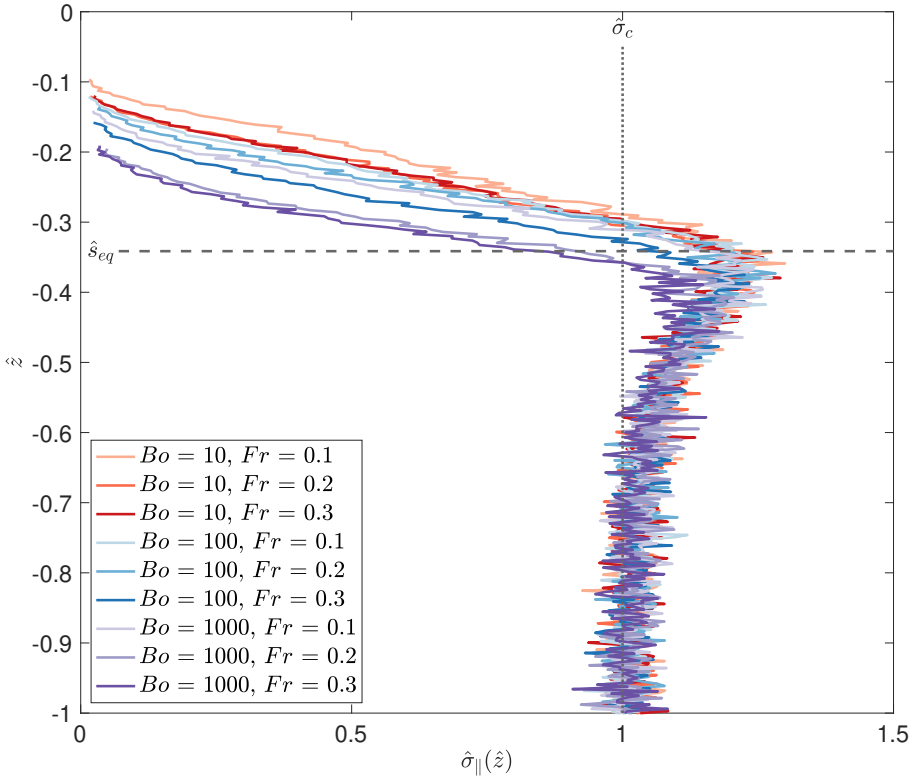
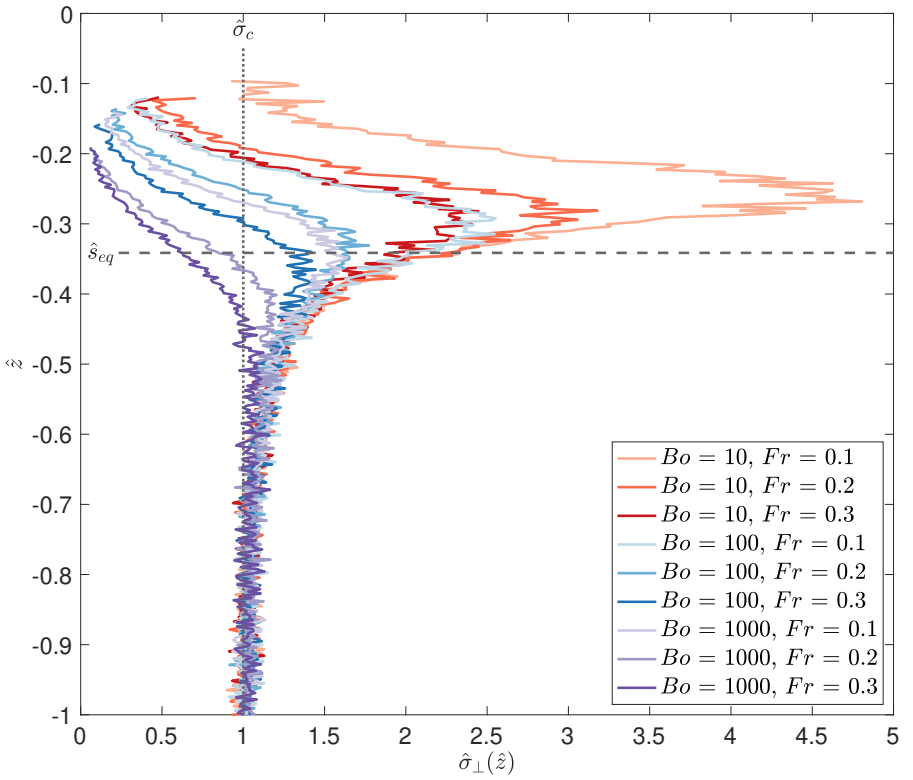



**Figure 6.3:** Two-dimensional normalized histograms of the $p(L_\parallel, z)$ distributions for the different simulations. The red dashed lines show the 99.7% confidence interval of the prior distribution.




**(a)** Horizontal impulse component




**(b)** Vertical impulse component

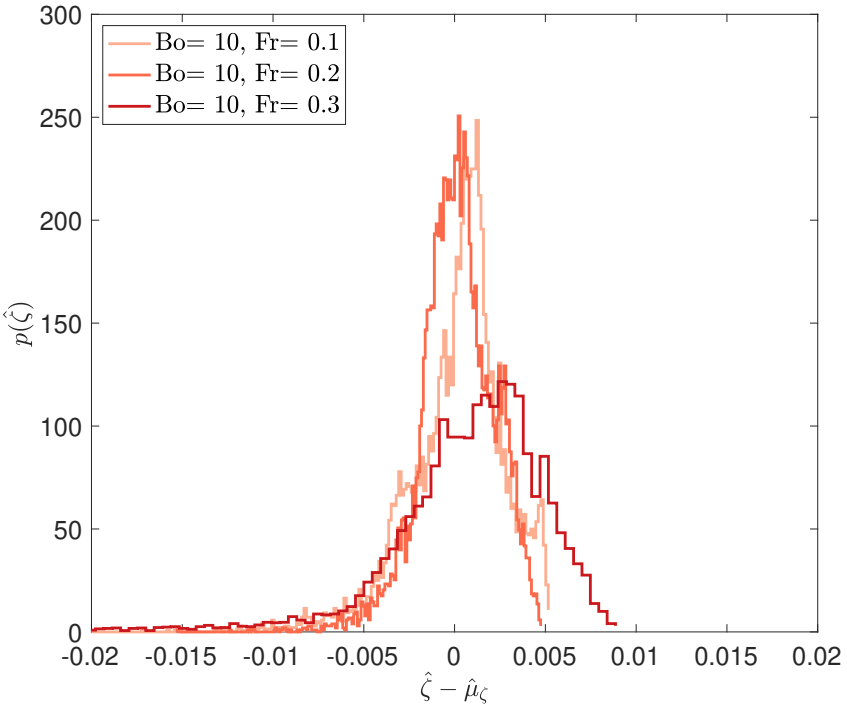
**Figure 6.4:** Standard deviation of impulse components as a function of depth for the different simulations in the test matrix. The dotted line represents the standard deviation of the prior distribution.

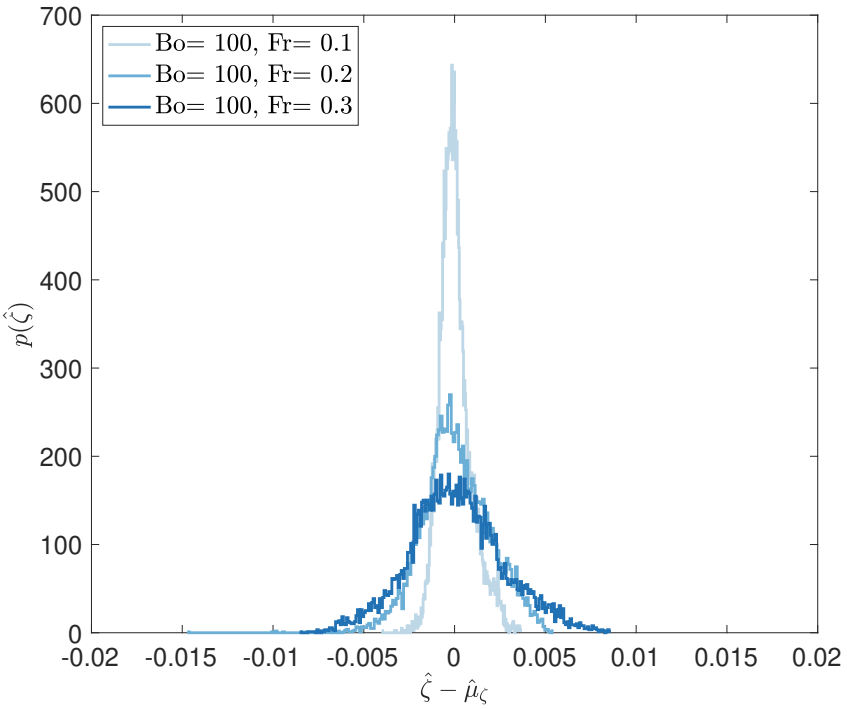
Note that $\sigma_{\parallel}(z)$ shows little variance between simulations, while $\sigma_{\perp}(z)$ vary greatly. However, there are slight peaks also for $\sigma_{\parallel}(z)$ below $s_{eq}$. For the vertical component, note that the peaks correspond to those of the $p(z)$ distribution. The peaks in $\sigma_{\perp}(z)$ are affected in the same manner by the non-dimensional groups as the $p(z)$ distribution, i.e. deeper and less distinct peaks for increasing $Bo$ and $Fr$. Furthermore, $\sigma_{\perp}(z) \approx \sigma_{\parallel}(z)$ for $Bo = 1000, Fr = 0.2$ and for $Bo = 1000, Fr = 0.3$ the parallel component is slightly higher around the peak value. All SDs from the simulations approach the prior distribution $\sigma_c$ for $z \to -h$.

## 6.2 Surface Statistics

Figure 6.5 shows the normalized frequency distributions of the surface elevation for eddies drawn from the different posterior distributions found in section 6.1. 5000 iterations were run with 20 eddies in the main domain. Due to time constraints it was not feasible to run more ensembles that this, nor a different number of eddies. The simulations are grouped by $Bo$ for demonstrative purposes. Because of the issue with a finite domain described in section 3.6, the simulations all have a non-zero mean, and they are therefore centered in $\mu_\zeta$ to easier compare their characteristics. Note that the histograms in general become flatter and wider for increasing $Fr$, with the exception of for the two lowest Froude numbers for $Bo = 10$.

The second, third and fourth central statistical moments of $\zeta$ at the end of the simulations are listed in tables 6.1, 6.2 and 6.3. To judge convergence of the simulations, Figure 6.6 presents these moments for an increasing number of ensembles, giving an indication on the convergence of the simulations. Notice how some of the simulations, especially those with $Fr = 0.1$ experience sharp jumps in skewness and kurtosis. A table of the maximum value of $|\nabla\zeta|$ for the different simulations is also presented in Table 6.4 in order to see whether the linear surface wave approximation is kept.


Bo= 10, Fr= 0.1
Bo= 10, Fr= 0.2
Bo= 10, Fr= 0.3
$p(\hat{\zeta})$
$\hat{\zeta} - \hat{\mu}_\zeta$
300
250
200
150
100
50
0
-0.02
-0.015
-0.01
-0.005
0
0.005
0.01
0.015
0.02


Bo= 100, Fr= 0.1
Bo= 100, Fr= 0.2
Bo= 100, Fr= 0.3
$p(\hat{\zeta})$
$\hat{\zeta} - \hat{\mu}_\zeta$
700
600
500
400
300
200
100
0
-0.02
-0.015
-0.01
-0.005
0
0.005
0.01
0.015
0.02

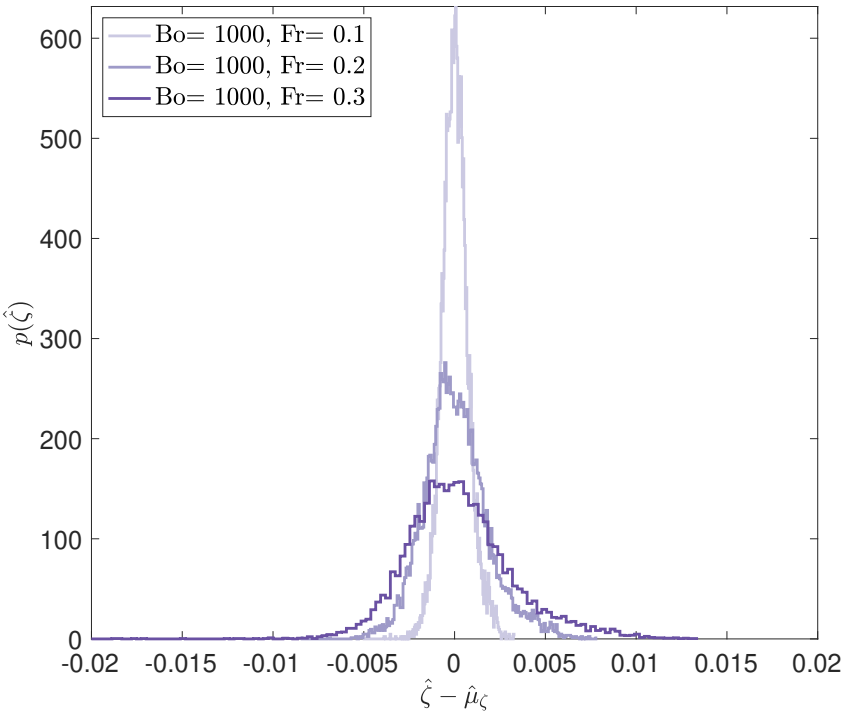
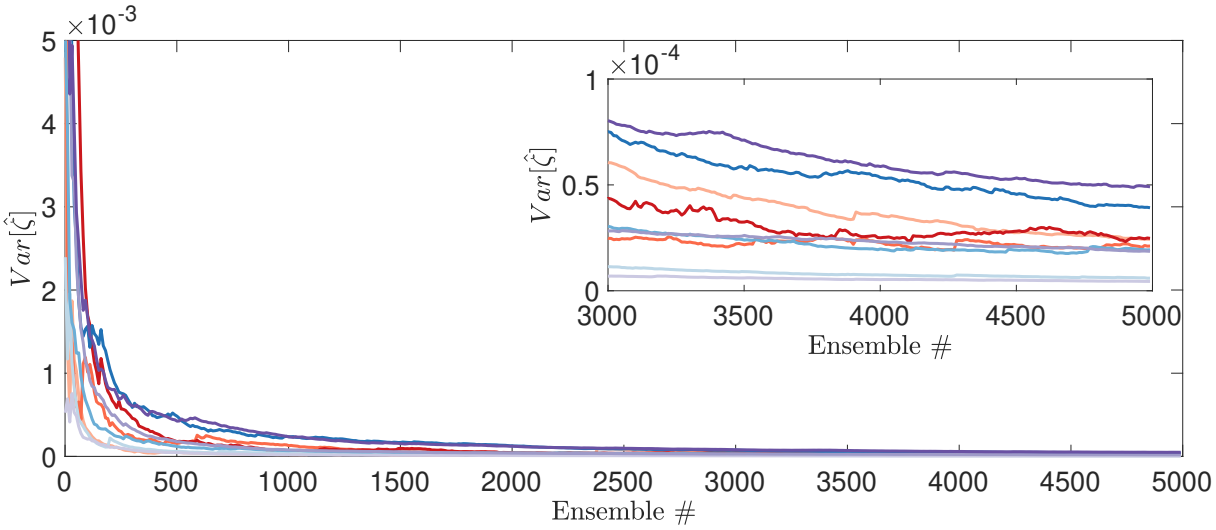
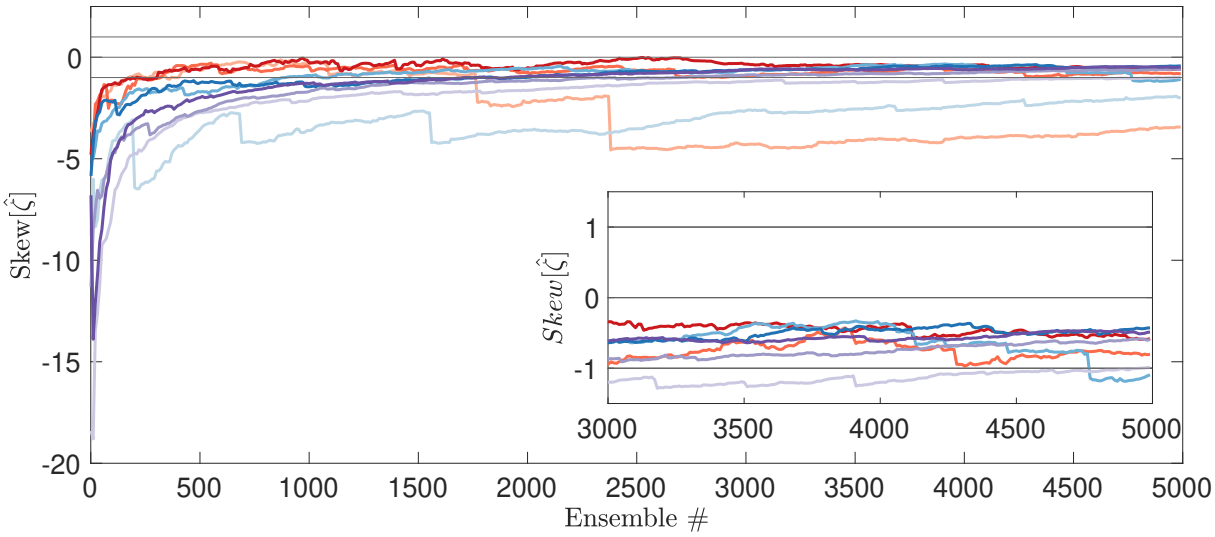


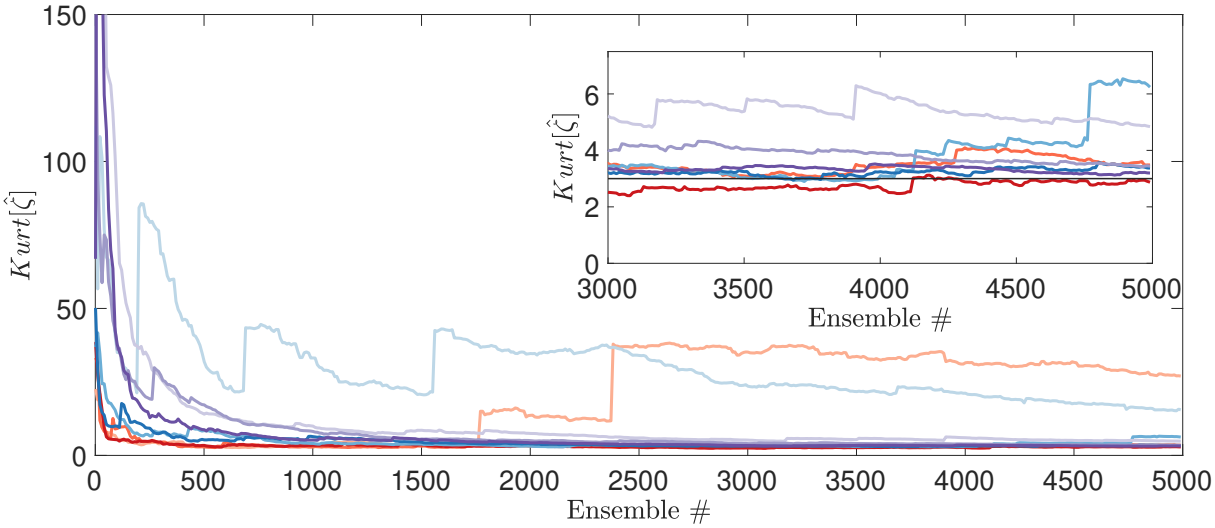

**Figure 6.5:** Normalized histograms of sampled surface elevation distributions away from mean. The different simulations are grouped by Bond number.




**(a)** Variance




**(b)** Skewness




**(c)** Kurtosis

**Figure 6.6:** Statistical central moments for an increasing number of ensembles

| **Var**$[\hat{\zeta}] \times 10^4$ | | | |
|---|---|---|---|
| $Fr$ \ $Bo$ | 10 | 100 | 1000 |
| 0.1 | 0.24 | 0.06 | 0.04 |
| 0.2 | 0.21 | 0.19 | 0.19 |
| 0.3 | 0.25 | 0.39 | 0.49 |

**Table 6.1:** Variance of sampled surface deformation distribution

| **Skew**$[\hat{\zeta}]$ | | | |
|---|---|---|---|
| $Fr$ \ $Bo$ | 10 | 100 | 1000 |
| 0.1 | -3.43 | -1.98 | -0.99 |
| 0.2 | -0.8 | -1.09 | -0.59 |
| 0.3 | -0.58 | -0.43 | -0.49 |

**Table 6.2:** Skewness of sampled surface deformation distribution

| **Kurt**$[\hat{\zeta}]$ | | | |
|---|---|---|---|
| $Fr$ \ $Bo$ | 10 | 100 | 1000 |
| 0.1 | 26.99 | 15.51 | 4.84 |
| 0.2 | 3.48 | 6.24 | 3.44 |
| 0.3 | 2.87 | 3.37 | 3.20 |

**Table 6.3:** Kurtosis of sampled surface deformation distribution

| **Max**$[\lvert\nabla\hat{\zeta}\rvert]$ | | | |
|---|---|---|---|
| $Fr$ \ $Bo$ | 10 | 100 | 1000 |
| 0.1 | 0.04 | 0.01 | 0.03 |
| 0.2 | 0.03 | 0.03 | 0.03 |
| 0.3 | 0.15 | 0.02 | 0.13 |

**Table 6.4:** Maximum surface gradient

# Chapter 7

# Discussion

## 7.1 Posterior Distribution

Both the surface and the eddies try to minimize their energy during the MCMC simulations, resulting in two competing effects. The surface repels eddies so that its deformation and thus deformation energy decreases. Meanwhile, the eddy will minimize its energy by decreasing interaction energy with its own mirror image. Like already mentioned, two anti-parallel eddies will have negative interaction energy between them, as some of the velocity field is cancelled. The system of an eddy and its mirror which naturally has opposite $L_\perp$ components will therefore further minimize the interaction energy by reducing the distance between each other, or by maximizing the impulse components normal to the surface. An eddy is therefore attracted to the surface. The mirror image is of course not a real eddy but rather a tool to satisfy the dynamic BC. The attraction to the surface can therefore be interpreted as a way for the eddy to cancel some of its velocity field.

The peaks in the $p(z)$ and $p(L_\perp, z)$ distributions are located at the depth in which these two effects are in equilibrium. Recall that in the limit $Bo \to \infty$, the surface deformation is a scaled version of the DVP, and it is therefore logical that the peak of the distributions then closely approximate $s_{eq}$ for the higher $Bo$ simulations. It is also logical that decreasing the Bond number, i.e. smoothing out the surface should result in a shallower peak, because the flatter surface have less potential energy to drive the repulsion of the eddy. The same goes for decreasing the Froude number, as the weaker turbulence deforms the surface less, resulting in less repulsion from the surface. We can see that the effect of the two non-dimensional groups are similar, although they occur through different phenomena.

These competing effects also explain the strong preference for perpendicular impulse components, as the peaks in $\sigma_\perp(z)$ coincide with the peaks in $p(z)$, signifying that the eddy-mirror interaction is the main reason the eddies are observed more frequently in this region. Another tell-tale sign is that the strength of this interaction effect coincides with the relative strength of the eddy in relation to the surface, as seen by varying $Fr$ and $Bo$. The difference across the simulations in $\sigma_\perp(z)$ can be explained in the same manner as for the $p(z)$ distribution, i.e. a stronger surface has further reach in depth to repulse an eddy, and kill its impulse.

Recall from section 3.6 that above $s_{eq}$, the maxima of $P_{kin}^{n+\bar{n}}$ is for $\alpha = 0$, i.e. $\mathbf{L} = L\mathbf{e}_\parallel$, whilst for $P_{self}^{n+\bar{n}}$ it is for $\alpha = \pi/2$, i.e. $\mathbf{L} = L\mathbf{e}_\perp$. It is therefore to be expected that the surface energy will be minimized by horizontal impulse components below $s_{eq}$ and by vertical impulse components above. This effect is

hard to distinguish for most of the simulations because of the strong interaction effect concealing it. However in the simulation with the highest relative surface energy $Bo = 1000, Fr = 0.3$, the effect is visible, as $\sigma_{\parallel}(z) > \sigma_{\perp}(z)$ below $s_{eq}$ as opposed to above, where the opposite is observed. This effect is also likely the reason that there are small peaks also in the $\sigma_{\parallel}(z)$ distribution below $s_{eq}$. They are likely much smaller because of the $r^{-3}$ dependence of the self-motion term of the DVP compared to the $r^{-6}$ dependence of the kinetic term. In the future if the model is expanded beyond linear order, it would be interesting to examine higher Froude numbers to see this effect more clearly.

### 7.1.1 Limitations

In the grid dependence study, it was decided to accept 5% errors up to $z = -0.2$. It is clear that all the $p(z)$ distributions to some degree surpass this threshold, more so for the simulations with the flatter surface characteristics. Regardless, for this part of the analysis, the errors are deemed to be at an acceptable level because the intended objective to see how the system scales is achieved. However, this resolution makes it hard to converge the SDS simulations because of an issue which will be discussed shortly.

Another source of error to the results presented above could be the kernel smoothing used during MCMC simulations. As seen from Figure 2.2, deviations from the far-field velocity field occur for $r \lesssim R$. For the simulations presented here, $\hat{R} = (2\pi)^{-1/3} \approx 0.54$, which means some errors for the interaction energy is expected when $\hat{z} \gtrsim -0.27$. This is around the peak of the uppermost distribution. The surface velocity and the following surface energy was not calculated with smoothing however, and does not suffer errors. The model does throughout assume that the eddy maintains an enclosed volume of vorticity. In nature, it is observed that eddies break up and connect with the surface as discussed in section 1.2. Because the eddies throughout the MCMC simulations are presumed spherical under kernel smoothing, this effect is ignored, and could be another source of error.

## 7.2 Surface Statistics

Because of the $r^{-6}$ dependence of the kinetic pressure, there is a heavy bias for the surface deformations by eddies close to the surface. Although the probability of eddies close to the surface go to zero as we have seen in the posterior distributions, the few outlier eddies that are sampled close to the surface will drown out the effects of those further down. We can see from the skewness and kurtosis in Figure 6.6, that whenever an eddy gets sampled close to the surface, the whole distribution changes drastically. The effect is less substantial when the surface is strong, as we can see from larger jumps for the simulations with lower Froude numbers. This is likely from the fact that flatter surface characteristics allow for eddies closer to the surface, where the bias is stronger. It could also result from a low resolution in this area from the higher discretization errors, which was discussed in section 7.1. A lot of eddies in this region closest to the surface needs to be examined in order to compensate for the bias, which in turn requires a huge number of ensembles. Running more ensembles was not feasible however, due to the limited time frame. These unforeseen convergence difficulties means that not all the simulations are sufficiently converged to a statistically steady state. As a result, the statistics presented from the SDS simulations will contain large errors.

Having said this, there are still some notable trends in the data worth mentioning. The flatter $p(\hat{\zeta})$ distributions for lower Froude numbers, fits well with the definition of the Froude number. The

variance trend in Table 6.1 confirms the visual. Furthermore, all simulations are negatively skewed, i.e. longer left tails, indicating that the surface experience more downward than upward surface deformations. A vertical impulse vector for an eddy corresponds to a sharp dip in the surface, like shown in Figure 3.4d. One would therefore expect a preference for vertical impulse components to give a stronger preference for downward surface deformations. Table 6.2 shows that skewness become monotonically more negative for decreasing Froude numbers, which fits with this idea of lower Froude numbers having a stronger preference for vertical impulse. One would also expect more randomly oriented impulse vectors to give a more random surface distribution. For increasing Froude-number, also the excess kurtosis drops monotonically, indicating a surface distribution more closely resembling a random sea. This again fit nicely with the lower vertical impulse preference for stronger surfaces with higher Froude numbers. The points made in this paragraph is only speculation however until more converged simulations are obtained.

The heavy depth dependence in the SDS simulations suggest that $P_{kin}$ is the dominating term for the surface crispations, as all $p(z)$ distributions have a region of non-zero probability above $s_{eq}$. This could also indicate that the effect of eddy-eddy interaction on the surface is negligible, in which case would simplify the surface statistics analysis significantly. Nevertheless, the simulations were only run with a single eddy density, so a more densely packed domain of eddies should be examined before drawing any conclusions. To obtain faster convergence one might also draw eddies exclusively from the uppermost part of the posterior distributions, where the kinetic term dominate, i.e. above $s_{eq}$. In order to do this however, one still needs a good resolution in the posterior, such that a more refined grid should be used either way.

It is common to denote waves with a length to height ratio below $1/7 \approx 0.143$ to be linear [45]. Table 6.4 show that the highest surface gradient during the surface statistics simulations is 0.15, right above this value, such that the linear approximation in this case might not be appropriate.

## 7.3 Suggested Model Improvements

It has been a theme throughout this report that we can only perform simulations from a narrow range of values in the $\lambda - u$ phase space, due to issues with the surface being too flat or too steep. The latter issue can be resolved by extending the model to second order steepness.

To address the former issue, we need a better grid resolution, because as we have seen, a flatter surface repels eddies less and therefore observe more localized surface disturbances. An idea worth pursuing in the future to counteract this problem is to use an adaptive depth dependent grid when running MCMC simulations, refining the grid when an eddy is close to the surface. This can improve performance because one would only use a fine surface grid when it is needed. To avoid infinitely many gridpoints when an eddy is close to the surface, the grid might perhaps be refined only when the course grid gives an acceptance-probability for the MCMC simulations above a certain threshold.

By covering more scales, one could simulate more realistic turbulence with power-law scaling. In practice, this could be done by choosing eddies with length- and velocity scales across a whole spectrum in the $u - \lambda$ phase space for MCMC simulations. Experiments might be useful to determine the relevant values for this spectrum.

Another idea is to extend the model to quadruple moments. This way, one might not need kernel smoothing for the MCMC simulations, if the surface is strong enough to repel the eddies significantly. This would remove an element of uncertainty in the simulations as the interaction energy calculations

would be more accurate. Furthermore, if one were to also find the the expressions for $\dot{\dot{x}}^n$ and $\dot{L}^n$ including surface deformation, in addition to a proper symplectic integrator, the system could be properly propagated in time. One would still need some way to limit eddy interaction, but it is likely that a similar kernel smoothing method to that presented here with quadrupole moments would suffice with a smaller smoothing radius, such that the smoothing would rarely be needed. Even then there would still be a strong resemblence to the Hill's spherical vortex, which is a solution to the Navier Stokes equations. One could then also use HMC as described in section 4.5 to include the dynamics of the system in order to obtain more accurate posterior distributions including interaction.

# Chapter 8

# Conclusion

It was found that there are two competing effects determining the shape of the joint probability distribution for the depth and impulse vector of an eddy, both effects seeking to minimize the energy of the system. The energy of the surface is minimized by repelling eddies away from itself due to the energy it requires to deform. The energy of the eddy is minimized by cancelling some of its velocity field. That happens when the eddy is oriented close to the surface, with its impulse-vector normal to the surface, due to the dynamic BC. When these two effects are in equilibrium, a peak occurs in the probability distribution for the depth and vertical impulse of the eddy, whilst the horizontal impulse is largely unaffected. This equilibrium point depends on the relative strength of the surface compared to the turbulence, controlled by the non-dimensional groups $Bo$ and $Fr$. For the higher Bond numbers, i.e. longer turbulence length-scales, the peaks in the distribution were situated around $\hat{s}_{eq} = (8\pi)^{-1/3}$, which is the equilibrium depth between the respective maxima for the self-motion- and kinetic terms of the DVP. When decreasing the relative strength of the turbulence, the peaks are observed increasingly higher than $\hat{s}_{eq}$ while also being more distinct. Below this equilibrium depth, the impulse components have a slight preference for horizontal impulse components, because of the lower self-motion pressure associated with it. Further down, all the probability distributions approach the prior distribution which is an isotropic state where the eddy is unaffected by the surface.

It was found that convergence of the SDS simulations was particularly sensitive on the resolution of the posterior distribution close to the surface, which is inconveniently also the region most prone to error under the current numerical model. This sensitivity is attributed to the steep probability density gradients close to the surface because of the $r^{-6}$ dependence of induced velocity at the surface. In order to draw any conclusions from the surface statistics, one must therefore simulate the posterior distributions anew with a greater resolution. An adaptive depth dependent grid method was proposed for achieving a better resolution in this area while limiting computational cost.

Although the simulations were not properly converged and the uncertainty in the data are large, some trends were found that are in line with the results found in the posterior distributions. All simulations resulted in a negatively skewed $\zeta$ distribution, which might indicate that downward surface deformations are more frequent on the surface, due to the vertical impulse preference outlayed above. Furthermore the simulations with the stronger surface characteristics gave less excess kurtosis, which might indicate less deviation from a Gaussian sea. This is consistent with the posterior distributions because the eddies are repelled further down to where the orientation of the impulse

vectors are more isotropic, resulting in a more random DVP profile.

# Bibliography


[1] E. A. Terray, M. Donelan, Y. Agrawal, W. M. Drennan, K. Kahma, A. J. Williams, P. Hwang and S. Kitaigorodskii, 'Estimates of kinetic energy dissipation under breaking waves,' *Journal of Physical Oceanography*, vol. 26, no. 5, pp. 792–807, 1996.

[2] S. E. Belcher, A. L. Grant, K. E. Hanley, B. Fox-Kemper, L. Van Roekel, P. P. Sullivan, W. G. Large, A. Brown, A. Hines, D. Calvert *et al.*, 'A global perspective on langmuir turbulence in the ocean surface boundary layer,' *Geophysical Research Letters*, vol. 39, no. 18, 2012.

[3] R. Wanninkhof, W. E. Asher, D. T. Ho, C. Sweeney and W. R. McGillis, 'Advances in quantifying air-sea gas exchange and environmental forcing,' 2009.

[4] D. T. Ho, C. S. Law, M. J. Smith, P. Schlosser, M. Harvey and P. Hill, 'Measurements of air-sea gas exchange at high wind speeds in the southern ocean: Implications for global parameterizations,' *Geophysical Research Letters*, vol. 33, no. 16, 2006.

[5] H. Tennekes and J. L. Lumley, *A first course in turbulence*. MIT press, 2018.

[6] H. v. Helmholtz, 'Über integrale der hydrodynamischen gleichungen, welche den wirbelbewegungen entsprechen.,' *Journal für die reine und angewandte Mathematik*, vol. 1858, no. 55, pp. 25–55, 1858.

[7] L. Onsager, 'Statistical hydrodynamics,' *Il Nuovo Cimento (1943-1954)*, vol. 6, no. 2, pp. 279–287, 1949.

[8] L. N. Hand and J. D. Finch, *Analytical mechanics*. Cambridge University Press, 1998.

[9] P. Roberts, 'A hamiltonian theory for weakly interacting vortices,' 1972.

[10] T. F. Buttke and A. J. Chorin, 'Turbulence calculations in magnetization variables,' *Applied numerical mathematics*, vol. 12, no. 1-3, pp. 47–54, 1993.

[11] D. Pullin and P. Saffman, 'Vortex dynamics in turbulence,' *Annual review of fluid mechanics*, vol. 30, no. 1, pp. 31–51, 1998.

[12] L. Bernal and J. Kwon, 'Vortex ring dynamics at a free surface,' *Physics of Fluids A: Fluid Dynamics*, vol. 1, no. 3, pp. 449–451, 1989.

[13] M. Song, L. P. Bernal and G. Tryggvason, 'Head-on collision of a large vortex ring with a free surface,' *Physics of Fluids A: Fluid Dynamics*, vol. 4, no. 7, pp. 1457–1466, 1992.

[14] M. Gharib and A. Weigand, 'Experimental studies of vortex disconnection and connection at a free surface,' *Journal of Fluid Mechanics*, vol. 321, pp. 59–86, 1996.

[15] J. G. Telste, 'Potential flow about two counter-rotating vortices approaching a free surface,' *Journal of Fluid Mechanics*, vol. 201, pp. 259–278, 1989.

[16] P. A. Tyvand, 'Motion of a vortex near a free surface,' *Journal of fluid mechanics*, vol. 225, pp. 673–686, 1991.

[17] R. Savelsberg and W. Van De Water, 'Experiments on free-surface turbulence,' *Journal of Fluid Mechanics*, vol. 619, no. 1, p. 95, 2009.

[18] X. Guo and L. Shen, 'Interaction of a deformable free surface with statistically steady homogeneous turbulence,' *Journal of fluid mechanics*, vol. 658, p. 33, 2010.

[19] H. Lamb, *Hydrodynamics*. Cambridge university press, 1993.

[20] P. A. Davidson, *Turbulence: an introduction for scientists and engineers*. Oxford university press, 2015.

[21] I. Morse, *Pm and feshbach, h., methods of theoretical physics*, 1953.

[22] V. Oseledets, 'On a new way of writing the navier-stokes equation. the hamiltonian formalism,' *Russ. Math. Surveys*, vol. 44, pp. 210–211, 1989.

[23] A. Qi, 'Three dimensional vortex methods for the analysis of wave propagation on vortex filaments,' 1991.

[24] K. F. Riley, *Mathematical methods for physics and engineering*. Cambridge: Cambridge University Press, 2006, ISBN: 9780521861533.

[25] E. Branlard, 'Spherical geometry models: Flow about a sphere and hill's vortex,' in *Wind Turbine Aerodynamics and Vorticity-Based Methods*, Springer, 2017, pp. 407–417.

[26] O. M. Phillips, *The dynamics of the upper ocean*. Cambridge university press, 1966.

[27] V. F. Zaitsev and A. D. Polyanin, *Handbook of exact solutions for ordinary differential equations*. CRC press, 2002.

[28] F. Mandl, *Statistical physics*. London New York: Wiley, 1971, ISBN: 0471566586.

[29] T. Liu and C.-J. Kim, 'Contact angle measurement of small capillary length liquid in superrepelled state,' *Scientific reports*, vol. 7, 2017.

[30] M. Brocchini and D. Peregrine, 'The dynamics of strong turbulence at free surfaces. part 1. description,' *Journal of Fluid Mechanics*, vol. 449, p. 225, 2001.

[31] E. Kreyszig, *Advanced engineering mathematics*. Hoboken, NJ: John Wiley, 2006, ISBN: 0471488852.

[32] C. Cao, S. Ibrahim, K. Nakanishi and E. S. Titi, 'Finite-time blowup for the inviscid primitive equations of oceanic and atmospheric dynamics,' *Communications in Mathematical Physics*, vol. 337, no. 2, pp. 473–482, 2015.

[33] O. Ibe, *Fundamentals of applied probability and random processes*. Academic Press, 2014.

[34] MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[35] M. Betancourt, 'A conceptual introduction to hamiltonian monte carlo,' *arXiv preprint arXiv:1701.02434*, 2017.

[36] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, 'Equation of state calculations by fast computing machines,' *The journal of chemical physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[37] M. Bédard, 'Hierarchical models and tuning of random walk metropolis algorithms,' *Journal of Probability and Statistics*, vol. 2019, 2019.

[38] A. Gelman, G. O. Roberts, W. R. Gilks *et al.*, 'Efficient metropolis jumping rules,' *Bayesian statistics*, vol. 5, no. 599-608, p. 42, 1996.

[39] P. Müller, *A generic approach to posterior integration and Gibbs sampling*. Purdue University, Department of Statistics, 1991.

[40] W. C. Kallenberg, T. Ledwina and E. Rafajlowicz, 'Testing bivariate independence and normality,' *Sankhyā: The Indian Journal of Statistics, Series A*, pp. 42–59, 1997.

[41] R. E. Walpole, R. H. Myers, S. L. Myers and K. Ye, *Probability and statistics for engineers and scientists*. Macmillan New York, 1993, vol. 5.

[42] R. M. Neal *et al.*, 'Mcmc using hamiltonian dynamics,' *Handbook of markov chain monte carlo*, vol. 2, no. 11, p. 2, 2011.

[43] R. I. McLachlan and P. Atela, 'The accuracy of symplectic integrators,' *Nonlinearity*, vol. 5, no. 2, p. 541, 1992.

[44] H. Nyquist, 'Certain topics in telegraph transmission theory,' *Transactions of the American Institute of Electrical Engineers*, vol. 47, no. 2, pp. 617–644, 1928.

[45] M. Dingemans, *Water wave propagation over uneven bottoms*. River Edge, NJ: World Scientific Pub, 1997, ISBN: 9789810204273.

# Appendix A

# Matlab Code

## A.1 Classes

```
classdef Eddy %Not pass-by-reference for computing time reasons

    properties(SetAccess = private) %Read only
        loc(1,3) double = [nan,nan,nan] %Changed with updateLoc or updateState
        imp(1,3) double = [nan,nan,nan] %Changed with updateLoc or updateState
        massInertia (1,1) double {Eddy.mustBePositiveOrNan(massInertia)} = nan
        inducedVel(1,3) double = [0,0,0] %Changed with updateInducedVel
        inducedImp(1,3) double = [0,0,0] %Changed with updateInducedImp
        updatedInducedVel (1,1) logical = false
        updatedInducedImp (1,1) logical = false
    end

    properties
        Surface SurfaceSection = SurfaceSection.empty %Default value is empty object
    end

    properties(Dependent) %Updated with get methods when called
        eddy2surfaceVecs cell
        selfEnergy(1,1){mustBeNonnegative}
        interactionEnergy(1,1)
        SURFVEL_n cell
        P_SELF_n double
        P_INTVEL_n double
        P_INTIMP_n double
        ZETA_SELF_n double %Surface elevation due to self motion DVP
        ZETA_KIN_n double %Surface elevation due to kinetic DVP w/o other eddies
    end
```

```
28
29      methods
30
31          % Constructor
32          function obj = Eddy(loc,imp,M,Surface)
33
34              switch nargin
35                  case 0
36                      obj.loc=[nan,nan,nan];
37                      obj.imp=[nan,nan,nan];
38                      obj.massInertia = nan;
39                      obj.Surface = SurfaceSection.empty;
40                  case 3
41                      obj.loc=loc;
42                      obj.imp=imp;
43                      obj.massInertia = M;
44                      obj.Surface = SurfaceSection.empty;
45                  case 4
46                      obj.loc=loc;
47                      obj.imp=imp;
48                      obj.massInertia = M;
49                      obj.Surface = Surface;
50                  otherwise
51                      error("Invalid number of input arguments");
52              end
53
54          end
55
56          % Get functions
57          function eddy2surfaceVecs = get.eddy2surfaceVecs(obj) %Cell of r^np vectors
58              gp = obj.Surface.gridPoints;
59              eddy2surfaceVecs = cell(gp);
60              for xNode=1:gp
61                  for yNode=1:gp
62                      eddy2surfaceVecs{xNode,yNode} = [obj.Surface.X(xNode,yNode)-obj.
                            loc(1), obj.Surface.Y(xNode,yNode)-obj.loc(2), -obj.loc(3)];
63                  end
64              end
65          end
66
67          function selfEnergy = get.selfEnergy(obj)
68              selfEnergy = dot(obj.imp,obj.imp)/(2*obj.massInertia);
69          end
70
```

```
71          function intEnergy = get.interactionEnergy(obj) %Requires updated inducedVel
72              if ~obj.updatedInducedVel
73                  warning("InducedVel  not updated -> Outdated info")
74              end
75              intEnergy = 0.5.*dot(obj.imp,obj.inducedVel);
76          end
77
78          function SURFVEL_n = get.SURFVEL_n(obj) %Assumes no smoothingFunciton
79              SURFVEL_n = cellfun(@(r_np) (C_point(r_np)*(obj.imp'))', obj.
                    eddy2surfaceVecs, 'UniformOutput',false);
80          end
81
82          function P_SELF_n = get.P_SELF_n(obj)
83              P_SELF_n = cellfun(@(r_np) (1/obj.massInertia)*dot(obj.imp,C_point(r_np)
                    *obj.imp') , obj.eddy2surfaceVecs);
84          end
85
86          function P_INTVEL_n = get.P_INTVEL_n(obj) %Requires updated inducedVel
87              if obj.updatedInducedVel
88                  P_INTVEL_n = cellfun(@(r_np) dot(obj.inducedVel,C_point(r_np)*obj.
                        imp') , obj.eddy2surfaceVecs);
89              else
90                  P_INTVEL_n= zeros(obj.Surface.gridPoints);
91                  warning("Induced velocity not updated, setting all elements to zero
                        ");
92              end
93          end
94
95          function P_INTIMP_n = get.P_INTIMP_n(obj) %Requires updted inducedImp
96              if obj.updatedInducedImp
97                  P_INTIMP_n= cellfun(@(r_np) -dot(obj.inducedImp,B_point(r_np)) , obj
                        .eddy2surfaceVecs);
98              else
99                  P_INTIMP_n= zeros(obj.Surface.gridPoints);
100                 warning("Induced impulse not updated, setting all elements to zero")
                        ;
101             end
102         end
103
104         % Updating functions
105         function newObj = updateLoc(obj,loc)
106             newObj = obj;
107             newObj.loc = loc;
108             newObj.updatedInducedVel = false;
```

```
109         end
110
111         function obj = updateImp(obj,imp)
112             obj.imp = imp;
113             obj.updatedInducedImp = false;
114         end
115
116         function obj = updateState(obj,loc,imp) %Updates both location and impulse
117             obj.loc = loc;
118             obj.imp = imp;
119             obj.updatedInducedVel = false;
120             obj.updatedInducedImp = false;
121         end
122
123         function newObj = updateInducedVel(obj,interactionEddies,smoothingFunc,R)
124             if ~obj.updatedInducedVel
125                 obj.inducedVel = obj.calculateInducedVel(interactionEddies,
                        smoothingFunc,R);
126                 obj.updatedInducedVel = true;
127                 newObj = obj;
128             else
129                 warning("InducedVel already updated");
130             end
131         end
132
133         function obj = updateInducedImp(obj,interactionEddies,smoothingFunc,R)
134             if ~obj.updatedInducedImp
135                 obj.inducedImp = obj.calculateInducedImp(interactionEddies,
                        smoothingFunc,R);
136                 obj.updatedInducedImp = true;
137             else
138                 warning("InducedImp already updated");
139             end
140         end
141
142         function newObj = updateInducedProperties(obj,interactionEddies,
                smoothingFunc,R)
143             newObj = obj.updateInducedVel(interactionEddies,smoothingFunc,R);
144             newObj = newObj.updateInducedImp(interactionEddies,smoothingFunc,R);
145         end
146
147         function updatedObj = updateInducedState(obj,otherEddies)
148             obj = calculateInducedImp(obj,otherEddies);
149             obj = calculateInducedVel(obj,otherEddies);
```

```
150             updatedObj = obj;
151         end
152 
153         function newObj = resetUpdatedInducedProperties(obj)
154             obj.updatedInducedVel = false;
155             obj.updatedInducedImp = false;
156             newObj = obj;
157         end
158 
159         % Other functions
160         function inducedVel = calculateInducedVel(obj,interactionEddies,
                smoothingFunc,R)
161             inducedVel = [0;0;0];
162             x_n =obj.loc;
163             for m=1:length(interactionEddies)
164                 eddy_m = interactionEddies(m);
165                 x_m = eddy_m.loc;
166                 L_m = eddy_m.imp;
167                 inducedVel = (inducedVel + C_point_smoothed(x_m-x_n,R,smoothingFunc)
                        *L_m');
168             end
169         end
170 
171         function inducedImp = calculateInducedImp(obj,interactionEddies,
                smoothingFunc,R)
172             x_n = obj.loc;
173             DL_n = zeros(3);
174             for m=1:length(interactionEddies)
175                 eddy_m = interactionEddies(m);
176                 x_m = eddy_m.loc;
177                 L_m = eddy_m.imp;
178                 D_nm = D_point_smoothed(x_n-x_m,R,smoothingFunc);
179                 DL_n = DL_n + (D_nm(:,:,1).*L_m(1) + D_nm(:,:,2).*L_m(2) + D_nm
                        (:,:,3).*L_m(3));
180             end
181             inducedImp = -0.5.*DL_n*obj.imp';
182         end
183 
184         function P_KIN_n = calculateKineticPressure(obj,SURFVEL_NOTSELF) %P_KIN_n
                given total surface velocity field from all eddies
185             if nargin == 1
186                 SURFVEL_NOTSELF = cell(obj.Surface.gridPoints);
187                 SURFVEL_NOTSELF(:,:) = {[0,0,0]};
188             end
```

```
189             SURFVEL_SELF = obj.SURFVEL_n;
190             P_KIN_n = cellfun(@(v_n,v_not_n) 0.5*(dot(v_n,v_not_n) + dot(v_n,v_n)),
                    SURFVEL_SELF,SURFVEL_NOTSELF);
191         end
192
193         function P_TOT_n = calculateTotalPressure(obj,SURFVEL_NOTSELF)
194             if nargin == 1
195                 SURFVEL_NOTSELF = cell(obj.Surface.gridPoints);
196                 SURFVEL_NOTSELF(:,:) = {[0,0,0]};
197                 warning("Other velocity contributions ignored because no SURFVEL_tot
                        was provided");
198             end
199             eddySurfaceVels = obj.eddy2surfaceVecs; %Only calculate once
200             pSELF_n = cellfun(@(r_np) (1/obj.massInertia)*dot(obj.imp,C_point(r_np)*
                    obj.imp') , eddySurfaceVels);
201             pINTVEL_n = cellfun(@(r_np) dot(obj.inducedVel,C_point(r_np)*obj.imp') ,
                     eddySurfaceVels);
202             pINTIMP_n= cellfun(@(r_np) -dot(obj.inducedImp,B_point(r_np)) ,
                    eddySurfaceVels);
203             pKIN_n = calculateKineticPressure(obj,SURFVEL_NOTSELF);
204
205             P_TOT_n = pSELF_n + pINTVEL_n + pINTIMP_n + pKIN_n;
206         end
207
208         function ZETA_SELF_n = get.ZETA_SELF_n(obj)
209             ZETA_SELF_n = ifft2( obj.Surface.TRANSFER.*fft2(obj.P_SELF_n));
210         end
211
212         function ZETA_KIN_n = get.ZETA_KIN_n(obj)
213             ZETA_KIN_n = ifft2( obj.Surface.TRANSFER.*fft2(obj.
                    calculateKineticPressure()));
214         end
215
216         function ZETA_n = calculateSurfaceElevation(obj,SURFVEL_NOTSELF)
217             if nargin == 1
218                 SURFVEL_NOTSELF = cell(obj.Surface.gridPoints);
219                 SURFVEL_NOTSELF(:,:) = {[0,0,0]};
220             end
221             P_TOT_n = obj.calculateTotalPressure(SURFVEL_NOTSELF);
222             ZETA_n = obj.Surface.calculateZetaFromPressureDistribution(P_TOT_n);
223         end
224
225         function E_def_n = calculateDeformationEnergy(obj,SURFVEL_NOTSELF)
226             if nargin == 1
```

```
227             SURFVEL_NOTSELF = cell(obj.Surface.gridPoints);
228             SURFVEL_NOTSELF(:,:) = {[0,0,0]};
229         end
230         ZETA_n = obj.calculateSurfaceElevation(SURFVEL_NOTSELF);
231         E_def_n = obj.Surface.calculateDeformationEnergy(ZETA_n);
232     end
233
234     function mEddy = mirror(obj) %Makes a mirror eddy
235         mEddy = obj;
236         mEddy.loc = [obj.loc(1), obj.loc(2), -obj.loc(3)];
237         mEddy.imp = [obj.imp(1), obj.imp(2), -obj.imp(3)];
238     end
239
240     function bool = isInsideDomain(obj,domain)
241         bool = (obj.loc(1) >= domain(1) && obj.loc(1) <= domain(2)...
242             && obj.loc(2) >= domain(3) && obj.loc(2) <= domain(4)...
243             && obj.loc(3) >= domain(5) && obj.loc(3) <= domain(6));
244     end
245
246     function [newPos,newImp] = integrateOverTimestep(obj,dt) %Time integration
        for isolated eddies
247         newPos = [0;0;0];
248         newImp = [0;0;0];
249         for i=1:3
250             xDot = @(t,x) obj.inducedVel(i) + obj.imp(i)/obj.massInertia;
251             lDot = @(t,x) obj.inducedImp(i);
252             [~,xOverTimeStep] = ode45(xDot,[0 dt],obj.loc(i));
253             [~,lOverTimeStep] = ode45(lDot,[0 dt],obj.imp(i));
254             newPos(i) = xOverTimeStep(end); %Update location
255             newImp(i) = lOverTimeStep(end); %Update momentum
256         end
257     end
258
259     function bool = isInFarfield(obj,otherEddies,delta) %Boolean value
        specifying if eddy is further away than delta from other eddies
260         x_n =obj.loc;
261         bool = true;
262         for m=1:length(otherEddies)
263             eddy_m = otherEddies(m);
264             x_m = eddy_m.loc;
265             if norm(x_n-x_m) < delta
266                 bool = false;
267                 break;
268             end
```

```
269             end
270         end
271 
272         function bool = isEmpty(obj) %Boolean value specifying if object is empty
273             bool = any(isnan(obj.loc)); %Checks if any of the loc properties are nan
274         end
275 
276         function otherEddies = extractFromVec(obj,eddyVec) %Extract eddy from vec
277             isSame = arrayfun(@(other) all(obj.loc == other.loc),eddyVec );
278             otherEddies = eddyVec(~isSame);
279         end
280 
281         % Plotting functions (requires cbrewer)
282         function plotSurfaceVelocities(obj,col)
283             U = cellfun(@(v_p) v_p(1), obj.SURFVEL_n);
284             V = cellfun(@(v_p) v_p(2), obj.SURFVEL_n);
285             plotEddies(obj,2,'ImpulseColor','r');
286             quiver(obj.Surface.X,obj.Surface.Y,U,V,0,'color',col);
287         end
288 
289         function plotPressureTerms(obj,contourLevels,SURFVEL_NOTSELF)
290             if nargin <3
291                 SURFVEL_NOTSELF = cell(obj.Surface.gridPoints);
292                 SURFVEL_NOTSELF(:,:) = {[0,0,0]};
293             end
294             cMap = flip(cbrewer('seq','Reds',contourLevels,'PCHIP'));
295             subplot(2,2,1);
296             contourf(obj.Surface.X,obj.Surface.Y,obj.P_SELF_n,contourLevels);
297             title("$P_{self}^n$");colormap(cMap);colorbar;axis equal;
298             subplot(2,2,2);
299             contourf(obj.Surface.X,obj.Surface.Y,calculateKineticPressure(obj,
                    SURFVEL_NOTSELF),contourLevels);
300             colormap(cMap);colorbar;title("$P_{kin}^n$");axis equal;
301             subplot(2,2,3);
302             contourf(obj.Surface.X,obj.Surface.Y,obj.P_INTVEL_n,contourLevels);
303             colormap(cMap);colorbar;title("$P_{int,vel}^n$");axis equal;
304             subplot(2,2,4);
305             contourf(obj.Surface.X,obj.Surface.Y,obj.P_INTIMP_n,contourLevels);
306             colormap(cMap);colorbar;title("$P_{int,imp}^n$");axis equal;
307         end
308 
309         function plotTotalPressure(obj,contourLevels,SURFVEL_NOTSELF)
310             if nargin <3
311                 SURFVEL_NOTSELF = cell(obj.Surface.gridPoints);
```

```
            SURFVEL_NOTSELF(:,:) = {[0,0,0]};
        end
        P_tot = obj.calculateTotalPressure(SURFVEL_NOTSELF);
        cMap = flip(cbrewer('seq','Reds',contourLevels,'PCHIP'));
        figure;
        contourf(obj.Surface.X,obj.Surface.Y,P_tot,contourLevels);
        title("$P_{self}^n$");colormap(cMap);colorbar;axis equal;

    end

    function plotSurfaceElevationTerms(obj,contourLevels,SURFVEL_NOTSELF)
        if nargin <3
            SURFVEL_NOTSELF = cell(obj.Surface.gridPoints);
            SURFVEL_NOTSELF(:,:) = {[0,0,0]};
        end
        pKIN_n = calculateKineticPressure(obj,SURFVEL_NOTSELF);
        ZETA_self_n = calculateZetaFromPressureDistribution(obj.P_SELF_n);
        ZETA_kin_n = calculateZetaFromPressureDistribution(pKIN_n);
        ZETA_intvel_n = calculateZetaFromPressureDistribution(obj.P_INTVEL_n);
        ZETA_intimp_n = calculateZetaFromPressureDistribution(obj.P_INTIMP_n);

        cMap = cbrewer('seq','Blues',contourLevels,'PCHIP');
        figure;
        sgtitle(sprintf("$$Bo_{\\lambda}$$ = %.1f",obj.Surface.Bo));
        subplot(2,2,1);
        contourf(obj.Surface.X,obj.Surface.Y,ZETA_self_n,contourLevels);
        title("$\zeta_{self}^n$");colormap(cMap);colorbar;axis equal;
        subplot(2,2,2);
        contourf(obj.Surface.X,obj.Surface.Y,ZETA_kin_n,contourLevels);
        colormap(cMap);colorbar;title("$\zeta_{kin}^n$");axis equal;
        subplot(2,2,3);
        contourf(obj.Surface.X,obj.Surface.Y,ZETA_intvel_n,contourLevels);
        colormap(cMap);colorbar;title("$\zeta_{int,vel}^n$");axis equal;
        subplot(2,2,4);
        contourf(obj.Surface.X,obj.Surface.Y,ZETA_intimp_n,contourLevels);
        colormap(cMap);colorbar;title("$\zeta_{int,imp}^n$");axis equal;
    end

    function plotSurfaceElevation(obj,contourLevels,SURFVEL_NOTSELF)
        if nargin < 3
            SURFVEL_NOTSELF = cell(obj.Surface.gridPoints);
            SURFVEL_NOTSELF(:,:) = {[0,0,0]};
        end
        ZETA_n = obj.calculateSurfaceElevation(SURFVEL_NOTSELF);
```

```
356             cMap = cbrewer('seq','Blues',contourLevels,'PCHIP');
357             contourf(obj.Surface.X,obj.Surface.Y,ZETA_n,contourLevels);hold on;
358             colormap(cMap);colorbar;
359         end
360 
361         function plotSurfaceElevationWithMirror(obj,contourLevels,SURFVEL_NOTSELF)
362             if nargin < 3
363                 SURFVEL_NOTSELF = cell(obj.Surface.gridPoints);
364                 SURFVEL_NOTSELF(:,:) = {[0,0,0]};
365             end
366             ZETA_n = obj.calculateSurfaceElevation(SURFVEL_NOTSELF) + obj.mirror().
                    calculateSurfaceElevation(SURFVEL_NOTSELF);
367             cMap = cbrewer('seq','Blues',contourLevels,'PCHIP');
368             contourf(obj.Surface.X,obj.Surface.Y,ZETA_n,contourLevels);hold on;
369             colormap(cMap);colorbar;
370         end
371 
372     end
373 
374     % Static methods
375     methods(Static)
376         function mustBePositiveOrNan(var) %Function for property validation
377             if ~(var>0 || isnan(var))
378                 error('Must be positive or nan');
379             end
380         end
381     end
382 
383 end
```

```
1  classdef Turbulence
2      %Collection of Eddy objects and their respective periodic and mirror
           counterparts
3      properties
4          RealCenterEddies Eddy %Real Eddy objects in domain of interest
5          RealPeriodicEddies Eddy %Real Eddy objects around with periodic BCs. Columns
               : Different eddies, Rows: copies in order C->E->NE->N->NW->W->SW->S->SE
6          MirrorCenterEddies Eddy %Mirrors of RealCenterEddies
7          MirrorPeriodicEddies Eddy %Mirrors of RealPeriodicEddies
8          AllEddies Eddy %[RCE; RPE; MCE; MPE]
9 
10         smoothingFunc (1,1) single {mustBeMember(smoothingFunc,[0,1,2,3,4])} = 0
11         R (1,1) double {Turbulence.mustBePositiveOrNan(R)} = nan %Smoothing radius
```

```
12          maxInteractionLength (1,1) double {Turbulence.mustBePositiveOrNan(
                maxInteractionLength)} = nan %Radius of SOI
13          N_main (1,1) single {mustBeInteger,mustBeNonnegative} = 0
14          N_all (1,1) single {mustBeInteger,mustBeNonnegative} = 0 %Total number of
                eddies
15
16          domain (1,6) double {mustBeReal} %[xMin,xMax,yMin,yMax,zMin,zMax]
17          Surface SurfaceSection %Surface grid and parameters
18      end
19
20      properties (Dependent)
21          P_SELF
22          P_INTVEL
23          P_INTIMP
24          P_KIN
25          SURFVEL
26          P_TOT
27          ZETA
28          ValidEddies %All eddies to evaluate pressure terms for
29          selfEnergy (1,1) {mustBeNonnegative}
30          interactionEnergy (1,1) %Can be negative
31          deformationEnergy (1,1) {mustBeNonnegative}
32      end
33
34      methods
35
36          %Constructor
37          function obj = Turbulence(RealCenterEddies,domain,Surface,smoothingFunc,
                maxInteractionLength)
38              if nargin<2
39                  error("Turbulence must have RealCenterEddies and domain");
40              else
41                  if ~all(arrayfun(@(objct)isInsideDomain(objct,domain),
                        RealCenterEddies)) %Check that all eddies are within domain
42                      error("All Real Center Eddies must be within domain")
43                  end
44                  obj.RealCenterEddies = RealCenterEddies;
45                  obj.domain = domain;
46                  obj.N_main = length(RealCenterEddies);
47                  obj.N_all = 18*obj.N_main;
48                  %Copying
49                  RPE(8,obj.N_main) = Eddy(); %Empty 8*N array
50                  obj.RealPeriodicEddies = RPE;
51                  for n = 1:obj.N_main
```

```
52                  obj.RealPeriodicEddies(:,n) = makePeriodicCopies(
                        RealCenterEddies(n),domain(2));
53              end
54              obj.MirrorCenterEddies = arrayfun(@(o) mirror(o), obj.
                    RealCenterEddies);
55              obj.MirrorPeriodicEddies = arrayfun(@(o) mirror(o), obj.
                    RealPeriodicEddies);
56              obj.AllEddies = [RealCenterEddies;obj.RealPeriodicEddies;obj.
                    MirrorCenterEddies;obj.MirrorPeriodicEddies];
57              switch nargin
58                  case 2
59                      obj.Surface = SurfaceSection.empty;
60                      obj.smoothingFunc = 0;
61                  case 3
62                      obj.Surface = Surface;
63                      obj.smoothingFunc = 0;
64                  case 4
65                      obj.Surface = Surface;
66                      obj.smoothingFunc = smoothingFunc;
67                  case 5
68                      obj.Surface = Surface;
69                      obj.smoothingFunc = smoothingFunc;
70                      obj.maxInteractionLength = maxInteractionLength;
71                  otherwise
72                      error("Invalid number of arguments")
73              end
74              if nargin < 5; obj.maxInteractionLength = (3/sqrt(2))*(domain(2)-
                    domain(1)); end %Interaction with all eddies
75              if nargin < 4
76                  switch smoothingFunc
77                      case 0
78                          obj.R = nan;
79                      case {1,2}
80                          obj.R = (1/(2*pi))^(1/3);
81                      case {3,4}
82                          obj.R = (5/(4*pi))^(1/3);
83                      otherwise
84                          error("smoothingFunc must be integer in range [0,4]")
85                  end
86              end
87
88          end
89      end
90
```

```
91          % Get functions
92
93          function P_SELF = get.P_SELF(obj)
94              P_SELF = sum(cat(3,obj.ValidEddies.P_SELF_n), 3); %Cocatinate P_SELF
                    arrays for all Eddy objects in ValidEddies in 3rd dimension and sum
                    over 3rd dimension
95          end
96
97          function P_INTVEL = get.P_INTVEL(obj)
98              P_INTVEL = sum(cat(3,obj.ValidEddies.P_INTVEL_n), 3);
99          end
100
101         function P_INTIMP = get.P_INTIMP(obj)
102             P_INTIMP = sum(cat(3,obj.ValidEddies.P_INTIMP_n), 3 );
103         end
104
105         function SURFVEL = get.SURFVEL(obj) %u^tot cell from all ValidEddies
106             gp = obj.Surface.gridPoints;
107             SURFVELs = cat(3,obj.ValidEddies.SURFVEL_n); % Cocatinating SURF_VEL
                    cell arrays in third dimension
108             U = sum( cellfun(@(v_p) v_p(1),SURFVELs), 3); %Break down cell array to
                    U,V and W components and sum in third dimension
109             V = sum( cellfun(@(v_p) v_p(2),SURFVELs), 3);
110             W = sum( cellfun(@(v_p) v_p(3),SURFVELs), 3); %Not 0 for eddy w/o mirror
111             SURFVEL = cell(gp);
112             for xNode = 1:gp
113                 for yNode = 1:gp
114                     SURFVEL{xNode,yNode} = [U(xNode,yNode), V(xNode,yNode), W(xNode,
                            yNode)];
115                 end
116             end
117         end
118
119         function P_KIN = get.P_KIN(obj)
120             P_KIN = cellfun(@(v_p) 0.5*dot(v_p,v_p),obj.SURFVEL);
121         end
122
123         function P_TOT = get.P_TOT(obj) %Requires updated imp and vel
124             if all([obj.ValidEddies.updatedInducedVel,obj.ValidEddies.
                    updatedInducedImp])
125               SURFVEL_TOT = obj.SURFVEL;
126               N_valid = length(obj.ValidEddies);
127               P_TOT_ns = zeros(obj.Surface.gridPoints,obj.Surface.gridPoints,N_valid
                      );
```

```
            for n=1:N_valid
                P_TOT_ns(:,:,n) = obj.ValidEddies(n).calculateTotalPressure(
                    SURFVEL_TOT);
            end
            P_TOT = sum(P_TOT_ns,3); %Sum up pressure over 3rd dimension
        else
            error("Induced properties not updated")
        end
    end

    function ZETA = get.ZETA(obj)
        ZETA = obj.Surface.calculateZetaFromPressureDistribution(obj.P_TOT);
    end %Total surfdef from all ValidEddies

    function ValidEddies = get.ValidEddies(obj)
        ValidEddies = [obj.RealCenterEddies,obj.MirrorCenterEddies];
    end

    function E_def = get.deformationEnergy(obj)
        E_def = obj.Surface.calculateDeformationEnergy(obj.ZETA);
    end

    function K_self = get.selfEnergy(obj)
        K_self = sum([obj.RealCenterEddies.selfEnergy]);
    end

    function K_int = get.interactionEnergy(obj) %Requires updated induced props
        K_int = sum([obj.RealCenterEddies.interactionEnergy]);
    end

    %Other functions
    function IE = InteractionEddies(obj,n) %Eddies within SOI of eddy n
        AllEds = obj.returnAllOtherEddies(n); %Do not include itself
        interactionBool = arrayfun(@(ed) norm(obj.ValidEddies(n).loc - ed.loc) <
            obj.maxInteractionLength, AllEds); %Bools with eddies inside SOI
        IE = AllEds(interactionBool); %Extracting eddies from interactionBool
    end

    function allOtherEddies = returnAllOtherEddies(obj,n) %Returns 1D vector
        with all eddies except eddy n
        AllEddiesVec = reshape(obj.AllEddies',1,obj.N_all); %1D vec with correct
            indexing
        allOtherEddies = AllEddiesVec(setdiff(1:end,n)); %Excluding eddy n
    end
```

```
168
169         function newObj = changeSingleEddyCopies(obj,n,eddy_n) %Changes all copies
                of real eddy with index n
170             newObj = obj;
171             newObj.RealCenterEddies(n) = eddy_n;
172             newObj.MirrorCenterEddies = arrayfun(@(ed) mirror(ed), obj.
                    RealCenterEddies);
173             newObj.RealPeriodicEddies(:,n) = makePeriodicCopies(eddy_n,obj.domain(2)
                    );
174             newObj.MirrorPeriodicEddies(:,n) = arrayfun(@(ed) mirror(ed),newObj.
                    RealPeriodicEddies(:,n)); %Only mirror updated eddies
175             newObj.AllEddies(:,n) = [newObj.RealCenterEddies(:,n);newObj.
                    RealPeriodicEddies(:,n);newObj.MirrorCenterEddies(:,n);newObj.
                    MirrorPeriodicEddies(:,n)];
176         end
177
178         function newObj = updateInteractionProperties(obj) %Updates interaction
                properties of all eddies
179             newMainEddies(1,length(obj.RealCenterEddies))=Eddy();
180             for n=1:obj.N_main
181                 eddy_n = obj.RealCenterEddies(n);
182                 interactionEddies = obj.InteractionEddies(n);
183                 eddy_n = eddy_n.updateInducedVel(interactionEddies,obj.smoothingFunc
                        ,obj.R); %Calculating x_dot_t and L_dot_t
184                 eddy_n = eddy_n.updateInducedImp(interactionEddies,obj.smoothingFunc
                        ,obj.R);
185                 newMainEddies(n) = eddy_n;
186             end
187             newObj = Turbulence(newMainEddies,obj.domain,obj.Surface,obj.
                    smoothingFunc,obj.maxInteractionLength);
188         end
189
190         function plotTurbulence(obj,dim,arrowScale) %Plots the location and impulses
                 of all eddies in either 2 or 3 dimensions
191             plotEddies([obj.RealCenterEddies'; obj.RealPeriodicEddies(:)],dim,'
                    ArrowScale',arrowScale);
192             xline(obj.domain(1));xline(obj.domain(2));
193             yline(obj.domain(3));yline(obj.domain(4));
194             axis equal;
195             xlim(3.*obj.domain(1:2));ylim(3.*obj.domain(3:4));
196         end
197
198         function plotTurbulenceAndHighlightEddyN(obj,arrowScale,n) %Plots loc/imp
                vec of all eddies and highlights eddy n + SOI
```

```
199             obj.RealCenterEddies(n).plotSurfaceElevation(20);hold on;
200             plotEddies(obj.RealCenterEddies,2,'ArrowScale',arrowScale);
201             plotEddies(obj.InteractionEddies(n),2,'ImpulseColor','c','ArrowScale',
                    arrowScale); %Plot eddies in interaction range in cyan
202             plotEddies(obj.RealCenterEddies(n),2,'ImpulseColor','r','ArrowScale',
                    arrowScale);
203             for i=1:9
204                 plot(obj.AllEddies(i,n).loc(1) + obj.maxInteractionLength.*cosd
                        (0:360),...
205                     obj.AllEddies(i,n).loc(2) + obj.maxInteractionLength.*sind
                            (0:360),'c','LineWidth',2); %Circles with
                            maxInteractionlength
206             end
207             axis equal;axis tight;
208             xlim(obj.domain(1:2));ylim(obj.domain(3:4));
209             hold off;
210             zVal = obj.RealCenterEddies(n).loc(3);
211             ax=gca; ax.Title.String = sprintf('$$z_n = %.2f$$',zVal);
212         end
213
214         function plotSurfaceVelocities(obj,col) %Vector plot of u^tot
215             U = cellfun(@(v_p) v_p(1), obj.SURF_VEL);
216             V = cellfun(@(v_p) v_p(2), obj.SURF_VEL);
217             plotEddies(obj.RealCenterEddies,2);
218             quiver(obj.Surface.X,obj.Surface.Y,U,V,'color',col);
219         end
220
221         function plotPressureTerms(obj,contourLevels,plotEds)
222             cMap = flip(cbrewer('seq','Reds',contourLevels,'PCHIP'));
223             subplot(2,2,1);
224                 contourf(obj.Surface.X,obj.Surface.Y,obj.P_SELF,contourLevels);
225                 title("$P_{self}^n$");colormap(cMap);colorbar;axis equal;
226             subplot(2,2,2);
227                 contourf(obj.Surface.X,obj.Surface.Y,obj.P_KIN,contourLevels);
228                 colormap(cMap);colorbar;title("$P_{kin}^n$");axis equal;
229             subplot(2,2,3);
230                 contourf(obj.Surface.X,obj.Surface.Y,obj.P_INTVEL,contourLevels);
231                 colormap(cMap);colorbar;title("$P_{int,vel}^n$");axis equal;
232             subplot(2,2,4);
233                 contourf(obj.Surface.X,obj.Surface.Y,obj.P_INTIMP,contourLevels);
234                 colormap(cMap);colorbar;title("$P_{int,imp}^n$");axis equal;
235             if plotEds
236                 subplot(2,2,1); hold on; plotEddies(obj.RealCenterEddies,2,'
                        ImpulseColor','c'); axis(obj.domain(1:4));
```

```
                subplot(2,2,2); hold on; plotEddies(obj.RealCenterEddies,2,'
                    ImpulseColor','c'); axis(obj.domain(1:4));
                subplot(2,2,3); hold on; plotEddies(obj.RealCenterEddies,2,'
                    ImpulseColor','c'); axis(obj.domain(1:4));
                subplot(2,2,4); hold on; plotEddies(obj.RealCenterEddies,2,'
                    ImpulseColor','c'); axis(obj.domain(1:4));
            end
        end

    end

    methods(Static)

        function mustBePositiveOrNan(var) %Function for property validation
            if ~(var>0 || isnan(var))
                error('Must be positive or nan');
            end
        end
    end

end
```

```
classdef SurfaceSection < handle

    properties
        xVec
        kVec
        X %Real space meshgrid
        Y
        KX %Wavenumber space meshgrid
        KY
        K % Length of kVec over grid
        gridPoints (1,1) single {mustBeInteger}
        dx (1,1) double {mustBeNonnegative} %GridSpacing
        Bo  double {mustBeNonnegative} %Bond number
        Fr double {mustBeNonnegative} %Turbulent Froude number
        width
    end

    properties(Dependent)
        TRANSFER %Non dimensional transfer function k/omega^2(k). Not evaluated
            directly to avoid nan at origin
        OMEGA %Non dimensional dispersion relation
```

```
21          kSpaceAxis %k-space limits in imagesc plots
22          midNode %Node corresponding to middle of domain
23      end
24
25      methods
26
27          %% Constructor
28          function obj = SurfaceSection(width,gridPoints,Bo,Fr)
29              if nargin < 2
30                  error("Grid parameters must be provided")
31              end
32              obj.gridPoints = gridPoints;
33              obj.xVec = linspace(-width/2,width/2,gridPoints);
34              obj.dx = width/(gridPoints-1);
35              obj.width = width;
36              [obj.X,obj.Y] = ndgrid(obj.xVec);
37              [obj.KX,obj.KY] = build_k_mesh(gridPoints,gridPoints,obj.dx);
38              obj.kVec = sort(obj.KX(1,:));
39              obj.K = sqrt(obj.KX.^2 + obj.KY.^2);
40              if nargin == 4
41                  obj.Bo = Bo;
42                  obj.Fr = Fr;
43              else
44                  obj.Bo = [];
45                  obj.Fr = [];
46              end
47          end
48
49          %% Get-functions
50
51          function TRANSFER = get.TRANSFER(obj)
52              TRANSFER = -obj.Fr^2./(1 + (4*pi^2/obj.Bo).*obj.K.^2);
53          end
54
55          function kSpaceAxis = get.kSpaceAxis(obj)
56              kSpaceAxis = [min(min(obj.KX)),max(max(obj.KX)),min(min(obj.KY)),max(max
                    (obj.KY))];
57          end
58
59          function OMEGA = get.OMEGA(obj)
60              OMEGA = obj.Fr^(-1).*sqrt((2*pi)^(-1).*obj.K + (2*pi/(obj.Bo)).*obj.K
                    .^3);
61          end
62
```

```
63          function midNode = get.midNode(obj)
64              midNode = ceil(length(obj.xVec)/2);
65          end
66
67          %% Other functions
68
69          function ZETA = calculateZetaFromPressureDistribution(obj,DVP)
70              ZETA = ifft2( obj.TRANSFER.*(fft2(DVP)));
71          end
72
73          function gradZETAmax = calculateMaxSurfaceGradientMagnitude(obj,ZETA)
74              [gradZETAx,gradZETAy] = gradient(ZETA,obj.dx);
75              gradZETAmag = sqrt(gradZETAx.^2 + gradZETAy.^2);
76              gradZETAmax = max(gradZETAmag,[],'all');
77          end
78
79          function E_ST = calculateSurfaceTensionEnergy(obj,ZETA)
80              [gradZETAx,gradZETAy] = gradient(ZETA,obj.dx); %2D gradient
81              eps_ST = 0.5*obj.We^(-1).*((gradZETAx.^2 + gradZETAy.^2) );
82              E_ST = trapz(obj.xVec,trapz(obj.xVec,eps_ST,2)); %Integrating surface
                    energy density over whole domain
83          end
84
85          function E_pot = calculateSurfacePotentialEnergy(obj,ZETA)
86              [gradZETAx,gradZETAy] = gradient(ZETA,obj.dx); %2D gradient with spacing
                     dx
87              eps_pot = 0.5*obj.We^(-1).*(obj.Bo.*ZETA.^2 + (gradZETAx.^2 + gradZETAy
                    .^2) );
88              E_pot = trapz(obj.xVec,trapz(obj.xVec,eps_pot,2));
89          end
90
91          function E_def = calculateDeformationEnergy(obj,ZETA)
92              [gradZETAx,gradZETAy] = gradient(ZETA,obj.dx); %2D gradient
93              eps_ST = 0.5*obj.Fr^(-2).*(obj.Bo^(-1).*(gradZETAx.^2 + gradZETAy.^2) );
94              eps_pot = 0.5*obj.Fr^(-2).*(ZETA.^2);
95              eps_def_n = eps_ST + eps_pot;
96              E_def = trapz(obj.xVec,trapz(obj.xVec,eps_def_n,2));
97          end
98
99          function ZETA_FINE = refineSurfaceElevation(obj,ZETA,GPfine) %Interpolate a
                refined version of input matrix ZETA
100             xVecFine = linspace(obj.xVec(1),obj.xVec(end),GPfine); %Refine xVec
101             [Xfine,Yfine] = ndgrid(xVecFine); %Refine grid
102             ZETA_FINE = griddata(obj.X,obj.Y,ZETA,Xfine,Yfine);
```

```
        end

        function [Xfine,Yfine] = finerMesh(obj,gp) %Function for refining mesh
            [Xfine,Yfine] = ndgrid(linspace(obj.xVec(1),obj.xVec(end),gp));
        end

        function E_def = calculateDeformationEnergyWithRefinedSurface(obj,ZETA)
            gpFine = size(ZETA,1);
            dxFine = obj.width/(gpFine-1);
            xVecFine = linspace(-obj.width/2,obj.width/2,gpFine);
            [gradZETAx,gradZETAy] = gradient(ZETA,dxFine); %2D gradient
            eps_pot_n = 0.5*obj.Fr^(-2).*(ZETA.^2 + obj.Bo.*(gradZETAx.^2 +
                gradZETAy.^2) );
            E_def = trapz(xVecFine,trapz(xVecFine,eps_pot_n,2));
        end
    end

end
```

## A.2 RWM script

```
close all
clear all
clc

saveSim = false;
Bo = 10;
Fr = 0.1;
IT = 100;
simName = sprintf("MCMCruns/RWM_gen7_%d",simNum);
simNum = 9;

%Global parameters
w = 14; %Width of domain
h = 1; %Height of domain
gridPoints = 81;
smoothingFunc = 2; %Set to 0 for no smoothing

%Eddy parameters
M = 1;
N_real = 1;
beta = 1;
```

```
22
23 %Monte Carlo parameters
24 sigmaLc = sqrt(M/beta); %Std of L components
25 stdJump_x = 0.5*h;
26 stdJump_L = 0.5*sigmaLc; %Normalized wrt sigmaL
27
28 if smoothingFunc == 1 || smoothingFunc == 2
29     R = (M/(2*pi))^(1/3);
30 elseif smoothingFunc == 3 || smoothingFunc == 4
31     R = (5*M/(4*pi))^(1/3); %One eddy on top of mirror
32 elseif smoothingFunc == 0
33     R=nan;
34     minSep = (M/(2*pi))^(1/3); %Allowable separation distance during MCMC
35 end
36
37 %Distributing initial eddies
38 domain = [-w/2 w/2 -w/2 w/2 -h 0];
39 spawnDomain = [domain(1:5) -0.2]; %Fist iteration away from surface
40 SS = SurfaceSection(w,gridPoints,Bo,Fr); %Constructing Surface Section object
41
42 %Pre allocating
43 dK_int = zeros(1,IT);
44 N = 1;
45 K_self = zeros(1,IT); K_int = zeros(1,IT); dE_defs = zeros(1,IT);
46 L_mags = zeros(N,IT); Lx = zeros(N,IT); Ly = zeros(N,IT); Lz = zeros(N,IT);
47 x = zeros(N,IT); y = zeros(N,IT); z = zeros(N,IT);
48 dH_props = zeros(1,IT);
49 alphas = zeros(N,1); avgAlphas = zeros(1,IT); alpha = 0;
50 inducedVel_mag=zeros(N,IT); inducedVel_mag_prop=zeros(N,IT);
51 accepted = 0; rejected = 0;
52 acceptanceRatio = zeros(1,IT);
53 dE_defs = zeros(N,IT);
54
55 % Make jumping distributions
56 pdJump_x = makedist('Normal','mu',0,'sigma',stdJump_x);
57 pdJump_L = makedist('Normal','mu',0,'sigma',stdJump_L);
58
59 %Make initial eddy from prior, mirror and update induced properties
60 eddy_n = distributeEddies(N_real,spawnDomain,beta,M,SS,0);
61 eddy_n = eddy_n.updateLoc([0,0,eddy_n.loc(3)]);
62 mirror_n = eddy_n.mirror();
63 eddy_n = eddy_n.updateInducedVel(mirror_n,smoothingFunc,R);
64 SURFVEL_NOTSELF = mirror_n.SURFVEL_n;
65 E_def_init = eddy_n.calculateDeformationEnergy(SURFVEL_NOTSELF);
```

```
66
67  fprintf("Sim number %d: Bo=%1.f, Fr=%.1f \n",simNum,Bo,Fr);
68  disp(simName)
69  %% Loop
70  for it = 1:IT
71
72      %Drawing from jumping distribution
73      dRvec_prop = [0,0,pdJump_x.random()];
74      dLvec_prop = [pdJump_L.random(),pdJump_L.random(),pdJump_L.random()];
75
76      %Attributing new values to proposed new eddy
77      eddy_prop = eddy_n;
78      eddy_prop = eddy_prop.updateState(eddy_n.loc + dRvec_prop, eddy_n.imp +
            dLvec_prop); %->UpdatedInducedVel/UpdatedInducedImp = false
79
80      if eddy_prop.isInsideDomain(domain) %Check for allowed move
81          mirror_prop = eddy_prop.mirror();
82          eddy_prop = eddy_prop.updateInducedVel(mirror_prop,smoothingFunc,R);
83          SURFVEL_NOTSELF_n = mirror_n.SURFVEL_n;
84          SURFVEL_NOTSELF_prop = mirror_prop.SURFVEL_n;
85
86          % Calculate change in energy
87          dK_self = eddy_prop.selfEnergy - eddy_n.selfEnergy;
88          dK_int = eddy_prop.interactionEnergy - eddy_n.interactionEnergy;
89          dE_def = eddy_prop.calculateDeformationEnergy(SURFVEL_NOTSELF_prop) - eddy_n
                .calculateDeformationEnergy(SURFVEL_NOTSELF_n);
90          dH = dK_self + dK_int + dE_def;
91
92          alpha = exp(-beta*dH); %Probability of proposed state
93          q = unifrnd(0,1);
94
95          if alpha>q %Accept or reject based on acceptance probability
96              eddy_n = eddy_prop;
97              mirror_n = eddy_n.mirror();
98              dE_defs(it) = dE_def;
99              accepted = accepted+1;
100         else
101             rejected = rejected+1;
102         end
103
104     else
105         alpha = 0;
106         dH = 0;
107         rejected = rejected+1;
```

```
        dE_def = 0;
    end
    dH_props(it) = dH;

    % Store values for post-processing
    acceptanceRatio(it) = accepted/(accepted+rejected);
    impVecs = eddy_n.imp'; %Matrix with eddy impulse vecs
    locVecs = eddy_n.loc';
    L_mags(:,it) = norm(eddy_n.imp);
    Lx(:,it) = impVecs(1,:); Ly(:,it) = impVecs(2,:); Lz(:,it) = impVecs(3,:);
    x(:,it) = locVecs(1,:); y(:,it) = locVecs(2,:); z(:,it) = locVecs(3,:);
    K_self(it) = eddy_n.selfEnergy;
    K_int(it) = eddy_n.interactionEnergy;
    avgAlphas(it) = alpha;

    if floor(it*1000/IT) == it*1000/IT %Printing progress
        % Compute correlations for preliminary convergence evaluation
        itVecCondensed = 1:250:it; % Containing only every 250th iteration in order
            to forget about the autocorrelation
        LxCondensed = Lx(itVecCondensed);
        LyCondensed = Ly(itVecCondensed);
        try
            [corrXY,pValue] = corr(LxCondensed',LyCondensed','Type','Spearman');
        catch
            warning("Could not compute correlation, assigning nan");
            corrXY = nan;
            pValue = nan;
        end
        fprintf("Completed %d/%d iterations. AR = %.2f, Corr(L_x,L_y) = %.2f, p-
            value = %.2f \n",it,IT,acceptanceRatio(it),corrXY,pValue);
    end
end
E_def = E_def_init + cumsum(sum(dE_defs,1)); %Sum up cumulative changes in E_def
if saveSim
    save(simName);
end
fprintf("Computation complete \n")
```

## A.3 SDS script

```
clear all
close all
```

```
clc
set(0,'defaultAxesFontSize',20);
set(0,'defaultTextFontSize',20);
set(0,'defaultTextInterpreter','latex');
set(0,'defaultLegendInterpreter','latex');
set(0,'DefaultFigureWindowStyle','normal');

%Loading simulation

N_real_posterior = 20; %Number of real eddies per ensemble
ENS = 5; %Number of ensembles
histBins = 300; %Number of histogram bins hist(zeta)
minSepDist = (3/(2*pi))^(1/3);
saveSim = false;
interaction = true;
simNamePosterior = "SurfaceStatistics_runs/gen_7"; %File name of SDS simulations
thresholdDepth = -(8*pi)^(1/3); %s_eq
maxInteractionLength = 3.17; %r_c
smoothingFuncPosterior = 0;

gp = 101; %Overwrite gridpoint values from MCMC calculations
w = 14; h=1;
domain = [-w/2,w/2,-w/2,w/2,-h,0];

%Creating a cell array of the relevant MCMC simulations to draw posterior from
gen = 7;
filenms = {sprintf('MCMCruns/RWM_gen%d_1',gen),sprintf('MCMCruns/RWM_gen%d_4',gen),
    sprintf('MCMCruns/RWM_gen%d_7',gen),...
    sprintf('MCMCruns/RWM_gen%d_2',gen),sprintf('MCMCruns/RWM_gen%d_5',gen),sprintf(
        'MCMCruns/RWM_gen%d_8',gen),...
    sprintf('MCMCruns/RWM_gen%d_3',gen),sprintf('MCMCruns/RWM_gen%d_6',gen),sprintf(
        'MCMCruns/RWM_gen%d_9',gen)};

%Pre-allocating
zeta_mean = zeros(1,9); zeta_variance = zeros(1,9); zeta_skewness = zeros(1,9);
    zeta_kurtosis = zeros(1,9); %Vectors with surface statistics
maxGradZeta = zeros(1,9);
gradZETAmag = cell(1,9);
ZETA_ens_f = cell(1,9);
for f=1:length(filenms)
    load(filenms{f});

    SSposterior = SurfaceSection(w,gp,Bo,Fr); %Overwrite SS object from MCMC sims to
         include more gridPoints
```

```
42      ZETA_ens = zeros(SSposterior.gridPoints,SSposterior.gridPoints,ENS); %3D array
            with ens as third dimension
43
44      parfor ens=1:ENS %Running for-loops in parallel
45
46          itVec = 1:IT;
47          itsAboveThreshold = itVec(z>thresholdDepth); %Extract indecies of iterations
                 with z>threshold
48          it_indecies = randi(length(itsAboveThreshold),1,N_real_posterior); %Drawing
                N_real random indices
49          zVals_ens = z(it_indecies); %Using z-value from p(z)
50          LxVals_ens = Lx(it_indecies); %Using L-values from p(L|z)
51          LyVals_ens = Ly(it_indecies);
52          LzVals_ens = Lz(it_indecies);
53
54          %Spawning eddies
55          RCE = Eddy.empty; %Empty Eddy vector
56          for n = 1:N_real_posterior
57              eddy_n_imp = [LxVals_ens(n),LyVals_ens(n),LzVals_ens(n)];
58              locOK = false; %Enter while loop
59              while ~locOK
60                  eddy_n_loc = [random('uniform',domain(1),domain(2)),random('uniform'
                        ,domain(3),domain(4)),zVals_ens(n)]; %Assign x-y coords
61                  RCEnonEmpty = RCE(~(arrayfun(@(ed) isEmpty(ed),RCE))); %Extracting
                        only non-empty Eddy objects
62                  locOK = all(arrayfun(@(ed) norm(eddy_n_loc - ed.loc) > minSepDist,
                        RCEnonEmpty)); %Spawning location OK if all eddies further away
                        than delta
63              end
64              RCE = [RCE,Eddy(eddy_n_loc,eddy_n_imp,1,SSposterior)]; %Make Eddy object
                     and append RCE
65          end
66
67          TURB = Turbulence(RCE,domain,SSposterior,smoothingFuncPosterior,
                maxInteractionLength); %Construct Turbulence object
68          TURBupdated = TURB.updateInteractionProperties(); %Update interaction props
69          ZETA_ens(:,:,ens) = TURBupdated.ZETA; %Calculate total E_surf
70      end
71      ZETA_ens_f{f} = ZETA_ens; %Save all ZETA ensembles for sim f to cell array
72      ZETAavg = mean(ZETA_ens,3); %Mean surface deformation matrix
73
74      fprintf("Sims completed: %d \n",f);
75  end
76
```

```
77 if saveSim
78     save(simNamePosterior);
79 end
```

## A.4 Supporting functions

```
1 function B_np = B_point(r_np)
2     r_length = norm(r_np);
3     B_np = -(1/(4*pi* (r_length^3)) ).*r_np;
4 end
```

```
1 function C_np = C_point(r_np)
2
3     r_length = norm(r_np);
4     outer = [[r_np(1)*r_np(1), r_np(1)*r_np(2), r_np(1)*r_np(3)];...
5              [r_np(2)*r_np(1), r_np(2)*r_np(2), r_np(2)*r_np(3)];...
6              [r_np(3)*r_np(1), r_np(3)*r_np(2), r_np(3)*r_np(3)]]; %Outer product of
                   r_np with itself
7     C_np = (1/ ( 4*pi*( r_length^5) ) ).* (3.*outer - eye(3).*(r_length^2));
8
9 end
```

```
1  function D_np = D_point(r_np) %Function returning propogator C_np, i.e C_n(r_n)
       evaluated in point p
2  %Initializing D as 3x3x3 array
3  D_np = zeros(3,3,3);
4  r_length = norm(r_np);
5  for i=1:3
6      for j=1:3
7          for k=1:3
8              delta_ij = (i==j); %Kroeniker delta
9              delta_jk = (j==k);
10             delta_ik = (i==k);
11             D_np(i,j,k) = (3/(4*pi*r_length^7)) *( ((r_np(i)*delta_jk + r_np(j)*
                   delta_ik + r_np(k)*delta_ij))*r_length^2 - (5*r_np(i)*r_np(j)*r_np(k
                   )) );
12         end
13     end
14 end
```

```

end
```

```
function C_np = C_point_smoothed(r_np,R,coreFunctionNumber)

      if norm(r_np)==0
          r_np=1e-10*[1;1;1]; %Avoiding Nan at origin
      end

    switch coreFunctionNumber
        case 0
            f = @(r) 1;
            dfdr = @(r) 0;
            R = 1;
        case 1
            f = @(r) 1 - exp(-r.^3);
            dfdr = @(r) 3.*(r.^2)*exp(-r.^3);
        case 2
            f = @(r) tanh(r.^3);
            dfdr = @(r) 3*r.^2*(1-tanh(r.^3));
        case 3
            f = @(r) 1 + exp(-r.^3).*(1.5.*r.^3 - 1);
            dfdr = @(r) -3*r.^2*exp(-r.^3)*((3*r.^3)/2 - 1) + (9.*r.^2*exp(-r.^3))
                ./2;
        case 4
            f = @(r) tanh(r.^3)+ 1.5.*(sech(r.^3))^2.*r.^3;
            dfdr = @(r) 3*r^2*(1 - tanh(r^3)^2) + (9*r^2*sech(r^3)^2)/2 - 9*r^5*sech
                (r^3)^2*tanh(r^3);
        otherwise
            error("Input valid coreFunction number")
    end
    r = norm(r_np);
    r_star = r/R;

    C_np =  R^(-3)* ( (f(r_star) - (r_star*dfdr(r_star)/(3)) ).*C_point(r_np./R)  +
        ( (dfdr(r_star)) /(6*pi*r_star^2)).*eye(3));
end
```

```
function D_np = D_point_smoothed(r_np,R,coreFunctionNumber)
      if norm(r_np)==0
          r_np=1e-10*[1;1;1]; %Avoiding Nan at origin
      end
```

```
5
6       switch coreFunctionNumber
7           case 0
8               f = @(r) 1;
9               dfdr = @(r) 0;
10              ddfdr = @(r) 0;
11              R = 1;
12          case 1
13              f = @(r) 1 - exp(-r.^3);
14              dfdr = @(r) 3.*(r.^2)*exp(-r.^3);
15              ddfdr = @(r) 6*r*exp(-r^3) - 9*r^4*exp(-r^3); %Second derivative
16          case 2
17              f = @(r) tanh(r.^3);
18              dfdr = @(r) 3*r.^2*(1-tanh(r.^3));
19              ddfdr = @(r) 6*r*(1 - tanh(r^3)^2) - 18*r^4*tanh(r^3)*(1 - tanh(r^3)^2);
20          case 3
21              f = @(r) 1 + exp(-r.^3).*(1.5.*r.^3 - 1);
22              dfdr = @(r) -3*r.^2*exp(-r.^3)*((3*r.^3)/2 - 1) + (9.*r.^2*exp(-r.^3))
                    ./2;
23              ddfdr = @(r) -6*r*exp(-r^3)*((3*r^3)/2 - 1) + 9*r^4*exp(-r^3)*((3*r^3)/2
                     - 1) - 27*r^4*exp(-r^3) + 9*r*exp(-r^3);
24          case 4
25              f = @(r) tanh(r.^3)+ 1.5.*(sech(r.^3))^2.*r.^3;
26              dfdr = @(r) 3*r^2*(1 - tanh(r^3)^2) + (9*r^2*sech(r^3)^2)/2 - 9*r^5*sech
                    (r^3)^2*tanh(r^3);
27              ddfdr = @(r) 6*r*(1 - tanh(r^3)^2) - 18*r^4*tanh(r^3)*(1 - tanh(r^3)^2)
                    + 9*r*sech(r^3)^2 - 72*r^4*sech(r^3)^2*tanh(r^3) ...
28                  + 54*r^7*sech(r^3)^2*tanh(r^3)^2 - 27*r^7*sech(r^3)^2*(1 - tanh(r^3)
                        ^2);
29          otherwise
30              error("Input valid coreFunction number")
31      end
32      r = norm(r_np);
33      r_star = r/R;
34
35      g = @(r) f(r) - (r/3)*dfdr(r);
36      dgdr = @(r) (1/3)*(2*dfdr(r) - r*ddfdr(r));
37      D_np = R^(-4).*( D_point(r_np./R).*g(r_star) + dgdr(r_star)*F_point(r_np./R) ) ;
38  end
```

```
1   function F_np = F_point(r_np)
2   %Initializing F as 3x3 array
3   F_np = zeros(3,3,3);
```

```
r = norm(r_np);
for i=1:3
    for j=1:3
        for k=1:3
            delta_jk = (j==k);
            F_np(i,j,k) = (3/(4*pi*(r^6))) * (r_np(i)*r_np(j)*r_np(k) - (r_np(i)*
                delta_jk*(r^2)) );
        end
    end
end
end
```

```
function PeriodicEddies = makePeriodicCopies(eddy_n,R_xy)
east = eddy_n; west = eddy_n; north = eddy_n; south = eddy_n;
northEast = eddy_n; northWest = eddy_n; southEast = eddy_n; southWest = eddy_n;

east = east.updateLoc(east.loc + [2*R_xy,0,0]);
west = west.updateLoc(west.loc - [2*R_xy,0,0]);
north = north.updateLoc(north.loc + [0,2*R_xy,0]);
south = south.updateLoc(south.loc - [0,2*R_xy,0]);

northEast = northEast.updateLoc(northEast.loc + [2*R_xy,2*R_xy,0]);
northWest = northWest.updateLoc(northWest.loc + [-2*R_xy,2*R_xy,0]);
southEast = southEast.updateLoc(southEast.loc + [2*R_xy,-2*R_xy,0]);
southWest = southWest.updateLoc(southWest.loc + [-2*R_xy,-2*R_xy,0]);

PeriodicEddies = [east;northEast;north;northWest;west;southWest;south;southEast];
end
```

Trym Haddal

# Modelling Near Surface Turbulence with Helmholtz-Onsager Vortices

NTNU
Kunnskap for en bedre verden