Langmuir Turbulence in the Ocean Surface Boundary Layer

Towards a Sub-grid Statistical Climate Process Model

BROWN

Kara Hartig

Department of Physics

Brown University

Senior thesis submitted for

Sc.B in Physics

May 3, 2018

Abstract

Langmuir turbulence, formed when steady winds blow over the ocean, helps to mix the ocean surface boundary layer that regulates the exchange of heat and gas between the stable ocean and the volatile atmosphere. Due to their size and complexity, global climate models struggle to resolve turbulence on this scale, leading to persistent uncertainties in ocean surface temperatures and the mixed layer depth. As part of the effort to reduce climate model uncertainties and better understand ocean-atmosphere dynamics, this thesis involved the construction and analysis of a Direct Statistical Simulation (DSS) model of Langmuir turbulence. Simulations designed with two different approximations, the quasilinear and generalized quasilinear, were compared to a fully non-linear simulation to judge the ability of DSS to capture critical dynamics at minimal computational cost. I found that both approximations could produce Langmuir turbulence but the generalized quasilinear approximation does so with much greater accuracy.

Contents

List of Figures

Chapter 1

Introduction & Background

Scientists and policymakers alike rely on good climate models to illuminate the past, describe the present, and project into the future of the Earth system. With the rapid onset of anthropogenic climate change, an imperative and a time limit have emerged in the field of climate science. If we want to put precise numbers on sea level rise, average global temperature increase, desertification, and more, then we need our climate models to be fast, accurate, and informed by our best science on climate dynamics.

1.1 Direct Numerical Simulation

Climate models today are based on Direct Numerical Simulation (DNS), which takes a physically intuitive approach to climate modeling. It starts with an Earth System Model, a computerized representation of the Earth that stores important climate variables: air temperature, pressure, and humidity; solar radiation; ocean temperature, density, salinity, and gas content; landmass albedo and plant coverage; the list goes on to encompass as many climate-relevant variables as we can fit. Each variable is stored as a series of discrete values on a three-dimensional grid, ultimately requiring billions of variables. The Earth System Model is then allowed to evolve over time using known physical laws that mimic the conditions in the real climate. Chains of equations and sophisticated mathematical solvers allow solar radiation to heat the air and ocean, pressure differences to drive air currents, and storms to form and rage and die away. Climate statistics, which describe resilient patterns in the climate like storm tracks, average rainfall, and the likelihood of extreme weather events, are collected by observing the Earth System Model as it runs over years of simulated time. The DNS approach to climate modeling is ubiquitous, employed by climate researchers from university labs and research centers all the way up to the National Center for

Atmospheric Research (NCAR) and the Intergovernmental Panel on Climate Change (IPCC). A particularly striking example can be seen in figure 1.1, the result of a global ocean surface current simulation carried out by MIT's global circulation model.

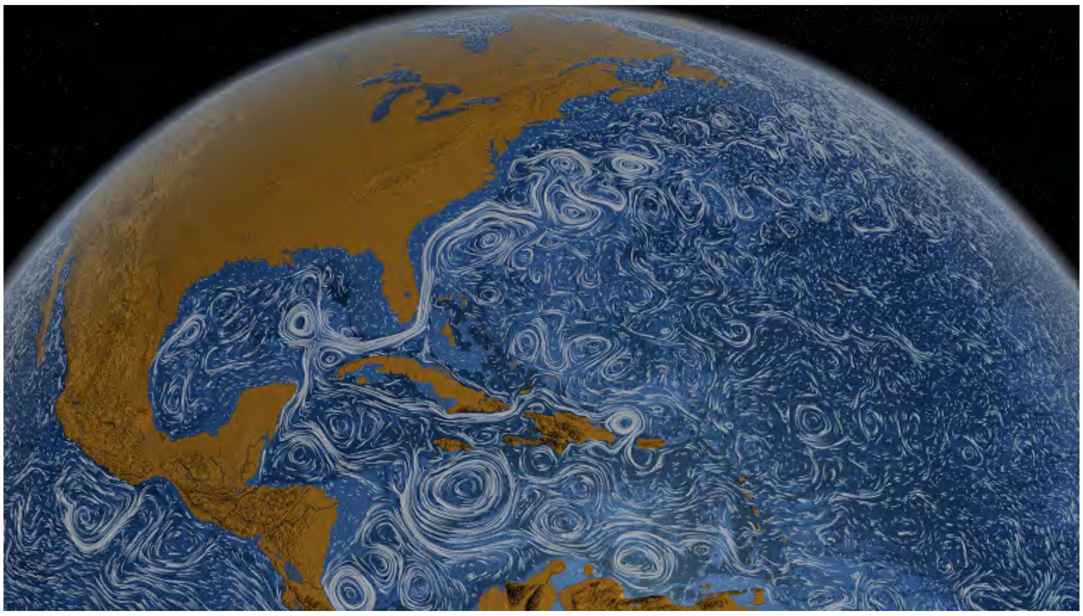


Figure 1.1: Visualization of the Gulf Stream using model data from ECCO2, a joint project between MIT and the Jet Propulsion Laboratory that uses the MIT global circulation model to study ocean currents [1].

For all that DNS climate models have accomplished in describing and predicting changes in the climate system, they have well-known limitations. Storing billions of variables takes a lot of computing power, and the higher you push the grid resolution the more memory and procesing power you require: if I have a grid point every 10 kilometers and I want to improve the resolution to 5 kilometers, I have multiplied the number of grid points not by 2 but by 8 since I am halving the grid spacing along not just one but all three spatial dimensions ($2^3 = 8$). I also have to cut the time step in half to ensure numerical convergence once I start to solve the equations of motion, so the multiplier on computing resources needed is actually $2^4 = 16$. When there are hundreds or thousands of climate variables defined at each grid point, the computing load adds up fast. Climate dynamics are complicated, requiring many equations and interactions that take even supercomputers a long time to calculate. Unsurprisingly, DNS models take ages to run, easily weeks or months. The burden of billions of variables and computationally intensive equations leads to long run times and requires huge supercomputers, tying up computing resources and introducing a significant time lag after any change or advance in modeling methods.

In order to reduce the computing burden, DNS models have to sacrifice some accuracy. The grid resolution is generally the easiest to cut, so many DNS models must accept a spatial resolution of hundreds of kilometers. This resolution is surprisingly effective for large-scale features like air or ocean currents and solar radiation intake, but it fails to represent small scale, or sub-grid, features. There are work-arounds that sneak these features into climate models without representing them explicitly, such as tuning physical parameters to preserve the effects of sub-grid mechanisms on a larger scale or assuming homogeneity or isotropy. But these approaches, while providing useful climate model results, do not give scientists a fundamental understanding of the climate dynamics at work. If we want to improve climate models while preserving sub-grid mechanisms, we need to consider alternatives to DNS that provide a deeper understanding of climate dynamics.

1.2 A Brief Introduction to Direct Statistical Simulation

Direct Statistical Simulation (DSS) presents an alternative to DNS. Developed by Professor Brad Marston of Brown University and colleagues, DSS represents each climate variable as the sum of a mean field and a fluctuation. The quasilinear approximation is then applied to throw out any interactions between fluctuation terms, simplifying the equations of motion to speed up computation time while preserving larger-scale behavior [9, 10]. Ultimately, DSS could produce a full sub-grid climate process model capable of representing climate features that are not resolved in DNS models due to resolution constraints.

Before DSS can be implemented in a full climate model, we must confirm that it accurately represents key climate features. Since completing the theoretical framework for DSS, Marston and colleagues have successfully applied it two test cases in fluid dynamics, zonal jets [10] and convection [11]. The final test case, and the focus of this thesis, is Langmuir turbulence.

1.3 Langmuir Turbulence

When a steady wind blows over the ocean and exerts a force on the surface of the water, it generates ocean surface gravity waves, which have a period of 1 to 10 seconds and a wavelength of a few to a few hundred meters [12]. Figure 1.2 shows the frequency distribution and sources of the many kinds of waves that appear in the ocean; wind-driven surface gravity waves appear at the far right.

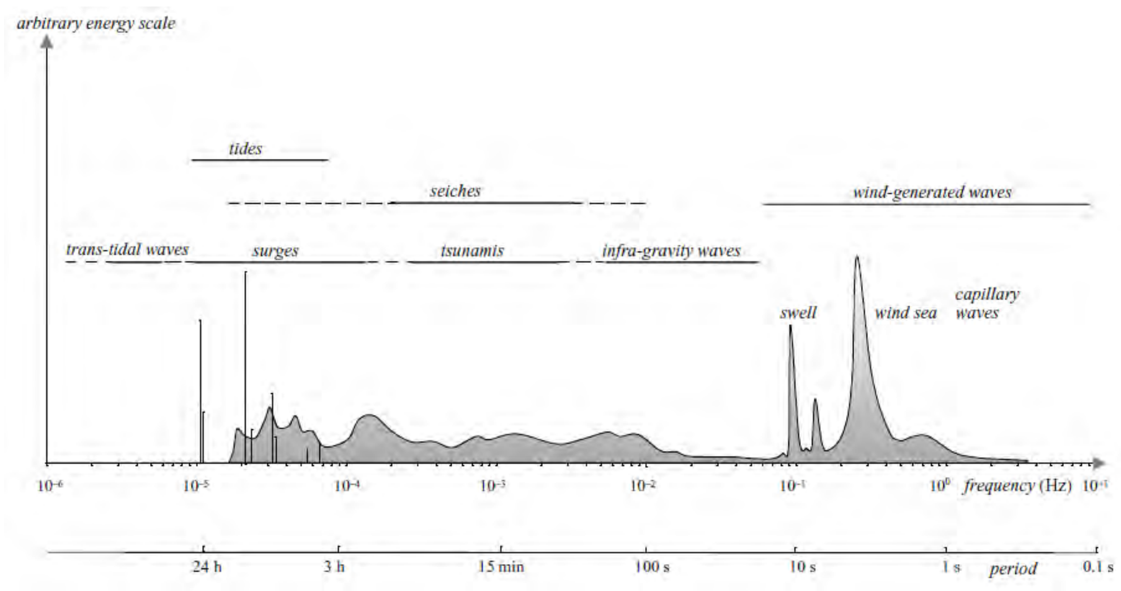


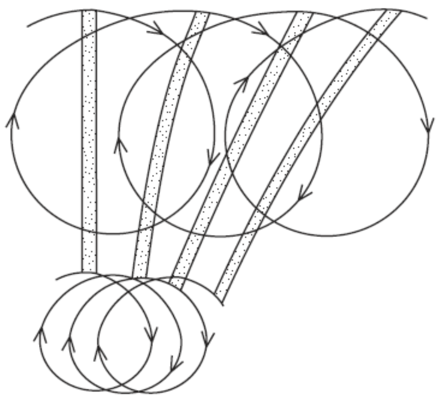
Figure 1.2: Figure 1.1 of Holthuijsen [2]. Frequencies and periods of ocean waves. Surface gravity waves appear at the far right, under “wind-generated waves”.

If we watch a parcel of water under the influence of a gravity wave, it follows the path illustrated in figure 1.3a, rising and moving forward as the wave crest passes and falling and sliding part of the way back once the crest has passed. This produces the cyclical pattern in the figure, but there is a net motion of water in the direction of wave propagation. Water near the surface tends to move the fastest, while viscous friction causes the water velocity to decrease exponentially with depth. The Stokes drift $\boldsymbol{u_s}$ describes the average velocity of a parcel of water as it travels with surface waves propagating in the $\hat{x}$ direction:

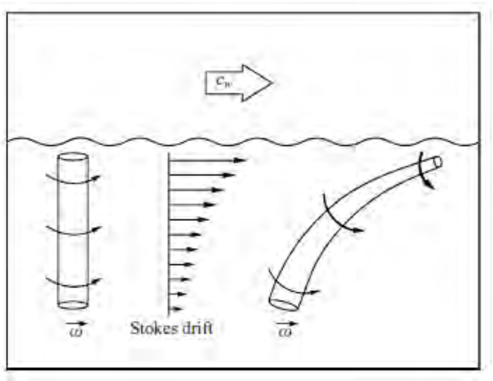
$$\boldsymbol{u_s}(z) = a_s \mathrm{e}^{2k_s z}\hat{x} \tag{1.1}$$

where z is depth, a_s is the drift velocity at the surface ($z = 0$), and k_s is the wavenumber.

Langmuir turbulence is a circulation pattern that forms in the surface layer of the ocean. Under steady wind and waves, the Stokes drift causes local vortices, vertical columns of rotating water which form and [dissolve] spontaneously due to the somewhat random motion of water, to tilt and stretch along the surface of the ocean, as illustrated in figure 1.3b. These vortices align roughly with the direction of wave propagation to form Langmuir cells, the characteristic feature of Langmuir turbulence:



(a) Figure 1.5b of Thorpe (2007) [13]. Path of a parcel of water under the influence of the Stokes force and tilting and stretching of a vertical column of water (dotted) as a result.



(b) Figure 8 of Teixeiras & Belcher (2002)[14]. Tilting and stretching of a vertical vortex of water under the influence of the Stokes force.

Figure 1.3: Illustrations of the Stokes drift in action on a column or parcel (a) or vertical vortex tube (b) of water near the ocean surface.

long counter-rotating rolls of water that stretch along the surface of the ocean, as illustrated in 1.4. Like gears, adjacent Langmuir cells rotate in opposite directions, producing alternating upwells and downwells in between cells. The downwells produce convergence zones along the surface of the ocean that collect surface material into long lines called "windrows" as the water to either side flows towards the middle and then downward. Windrows formed by Langmuir turbulence is often observed on the surface of the ocean and large lakes as foam or buoyant algae collects above the downwells; oil spills provide a particularly striking illustration of Langmuir windrows 1.5.

Although small relative to the other length scales of ocean dynamics, Langmuir circulation plays a major role in ocean circulation and the interaction between the ocean and the atmosphere. Relatively speaking, Langmuir cells are quite small, typically a few meters to a kilometer wide and three to ten times that in length [3]. However, their structure gives them an outsized influence on ocean dynamics. Recall the downwelling jets that form between adjacent Langmuir cells, illustrated in figure 1.6. The scale depth of the Stokes drift, which *causes* Langmuir turbulence, is limited by exponential decay from viscous friction to about $\delta_S = \frac{1}{2k} \approx 5$ m [5, 15]. The downwelling jets caused by Langmuir turbulence, however, can penetrate much deeper into the ocean, on the order of 100 m [5]. The difference in scale depth between the Stokes drift and the jets caused by Langmuir turbulence tells us that Langmuir circulation

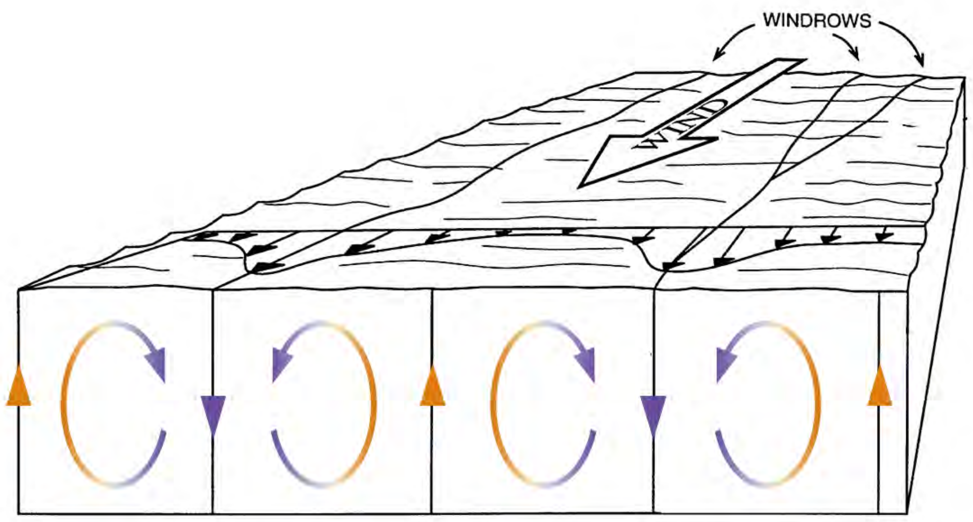


Figure 1.4: Adapted from Figure 1 of Thorpe (2004) [3]. A cross-section of Langmuir turbulence cells near the ocean surface, showing the counter-rotation and alternating upwells (red) and downwells (blue).

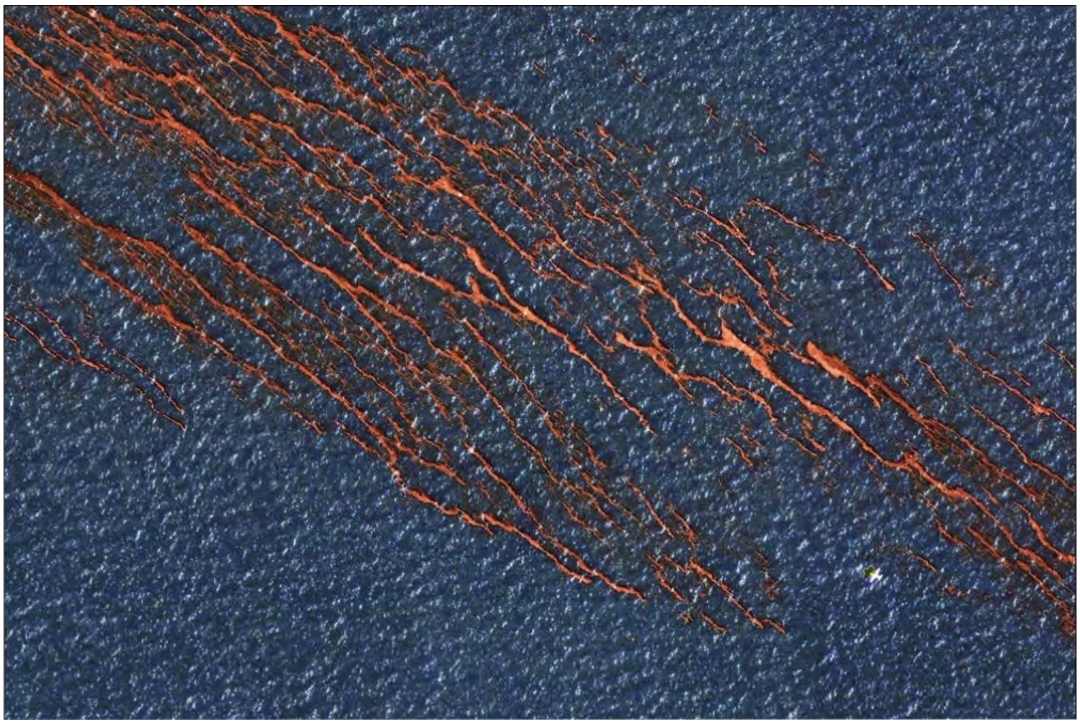


Figure 1.5: An aerial photograph from npr.org [4] taken just after the Deepwater Horizon oil spill in 2010 shows oil collecting along convergence zones to form Langmuir windrows. For a sense of scale: the small white spot in the lower-right is an airplane.

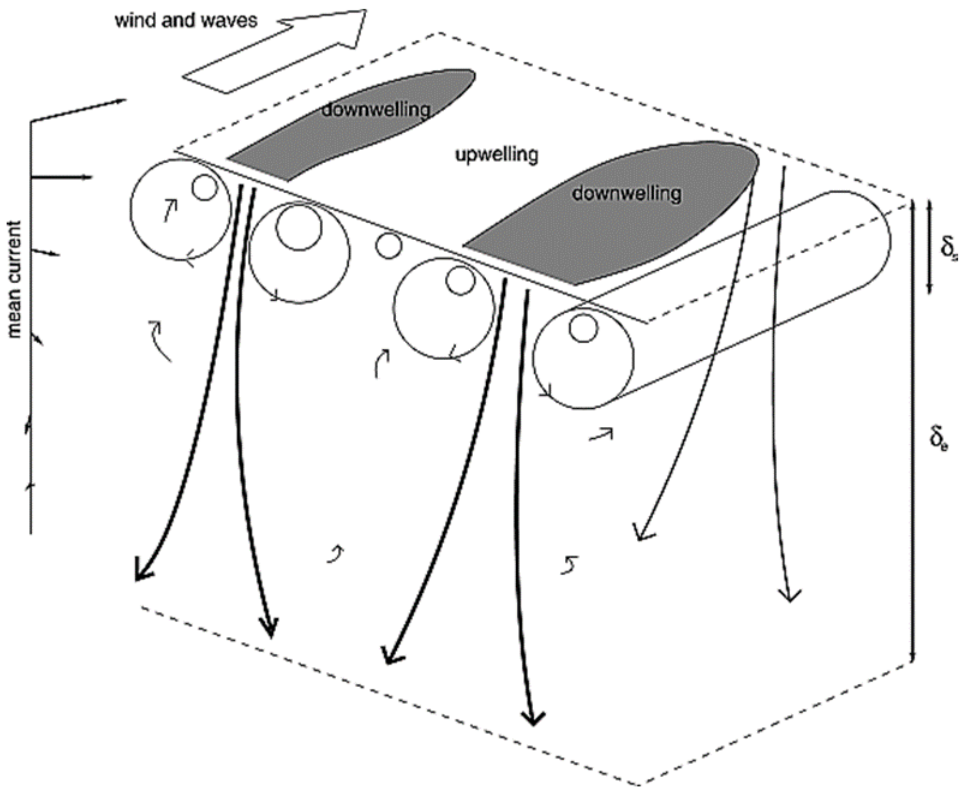


Figure 1.6: Figure 7 from Polton & Belcher (2007) [5]. The Stokes depth scale δ_S is compared to the Ekman depth scale δ_e on the far right, with the Langmuir cells oriented from the bottom left to the top right near the surface of the water. As illustrated, the downwelling jets can penetrate much deeper than the Stokes drift or Langmuir cells alone.

acts as an amplifier, translating wave forcing in the top $\frac{1}{2k} \approx 5$ m of the ocean into turbulent features that reach 100 m in depth.

The climate implications of Langmuir turbulence and the amplification effects mentioned above appear in the "mixed layer" or the "ocean surface boundary layer" (OSBL). The OSBL is the region composed of the top 100 m or so of the ocean that is more or less fully mixed, with constant temperature, salinity, and dissolved gas concentration throughout. Below the OSBL, the ocean is highly stratified: water is warmest near the bottom of the OSBL and gets monotonically colder with depth. Temperature stratification eliminates the mixing that would otherwise be driven by buoyancy imbalances, while distance from the surface minimizes forcing by wind, waves, or solar radiation. As a result, the "deep ocean" is highly stable and responsible for the long-term storage of trace gases and chemicals. The mixed layer serves as an

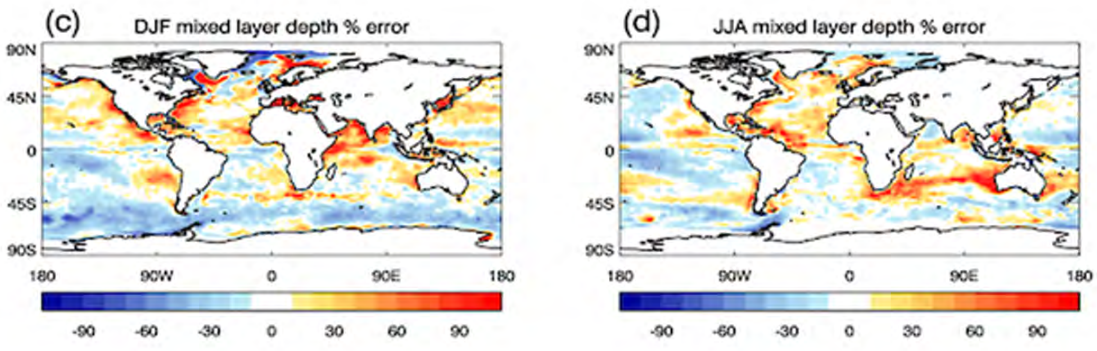


Figure 1.7: Figures 1c and 1d from Belcher (2012) [6]. Percent error between simulated and measured depths of the mixed layer for Dec-Jan-Feb (left) and Jun-Jul-Aug (right). Simulated data came from averaging HadGEM3 model data over 20 years, measured depths from Argo float data.

intermediary between the atmosphere, which varies on a time scale of days to weeks, and the deep ocean, which varies on a time scale of centuries or even longer. As a result, the depth of the mixed layer is a crucial climate variable. Langmuir turbulence helps to set this depth through the turbulent mixing driven by downwelling jets, so it stands to reason that including it in climate models is a good idea.

Unfortunately, climate modeling is not so simple. Recall that computing constraints restrict current climate models to a horizontal resolution of hundreds of kilometers, which is far too sparse to resolve Langmuir cells. There is evidence that climate model accuracy suffers as a result; figure 1.7 shows errors in the mixed layer depth based on current climate models as compared to data taken by floats [6]. In some regions, particularly in the June-July-August Southern Ocean, errors are as much as 100% of the calculated mixed layer, with corresponding errors in calculated sea surface temperature of 3-4° C [6].

There are many ways that ocean surface gravity waves can influence ocean mixing and the global climate, but Langmuir turbulence is increasingly considered one of the most important to incorporate into climate models [6, 7, 16]. D'Asaro et al. found that including Langmuir turbulence in a global model of boundary layer mixing increased the depth of the mixed layer by up to 40% (see figure 1.8). In the Southern Ocean, where models have historically suffered from a persistent shallow bias in the depth of the mixed layer, the inclusion of Langmuir turbulence could have a particularly powerful effect. By finding a more computationally manageable

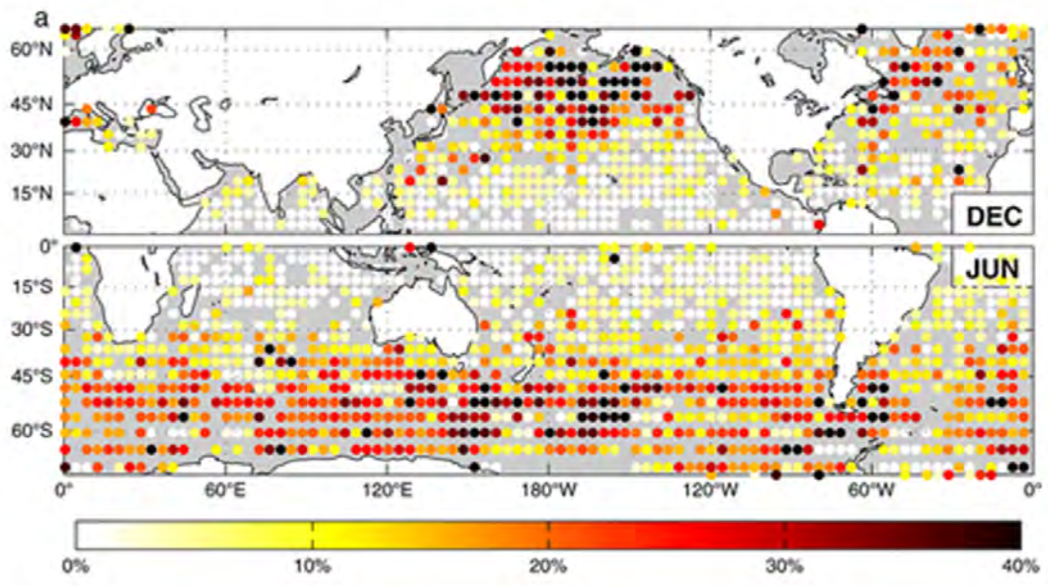


Figure 1.8: Adapted from Figure 3 of D'Asaro (2013) [7]. Percent increase in (winter) simulated mixed layer depth with wave forcing relative to no wave forcing; Langmuir turbulence is included through parametrization rather than explicit simulation.

way to simulate sub-grid Langmuir turbulence, we may be able to explicitly incorporate Langmuir turbulence into global climate models and improve some of the errors currently encountered in the mixed layer.

1.3.1 Better Than Your Average Test Case

Fortunately for us, Langmuir turbulence also has practical advantages as a test case. While the detailed dynamics of Langmuir circulation are complicated and remain an open area of research [3], the equations of motion were identified by Craik and Leibovich in 1976 [17]. The Craik-Leibovich equations can be written in various forms (see Suzuki and Fox-Kemper (2016) [18]), but for the simulations that follow the relevant form is shown in equation 1.2:

$$\boldsymbol{\nabla}\cdot\boldsymbol{u}=0 \tag{1.2a}$$

$$\begin{aligned}\partial_t\boldsymbol{u}+(\boldsymbol{u}\cdot\boldsymbol{\nabla})\,\boldsymbol{u}+\boldsymbol{f}\times(\boldsymbol{u}+\boldsymbol{u_s}) =& -\frac{1}{\rho_0}\boldsymbol{\nabla}\left(p+\frac{\rho_0}{2}\left(|\boldsymbol{u}+\boldsymbol{u_s}|^2-|\boldsymbol{u}|^2\right)\right)\\ &+\nu_h\nabla_h^2\boldsymbol{u}+\nu_z\nabla_z^2\boldsymbol{u}\\ &+\boldsymbol{u_s}\times\boldsymbol{\nabla}\times\boldsymbol{u}\\ &+\frac{\boldsymbol{\tau}}{\rho_0}\\ &+\boldsymbol{g}\alpha\,(T-T_0)\end{aligned} \tag{1.2b}$$

$$\partial_t T+((\boldsymbol{u}+\boldsymbol{u_s})\cdot\boldsymbol{\nabla})\,T=\frac{q}{c_p\rho_0}+\kappa_h\nabla_h^2T+\kappa_z\nabla_z^2T \tag{1.2c}$$

where $\boldsymbol{u}=\{u,v,w\}$ is the three-dimensional velocity vector, $\boldsymbol{u_s}$ is the Stokes drift defined in equation 1.1, $\boldsymbol{f}=f\hat{\boldsymbol{x}}$ is the Coriolis parameter, p is the three-dimensional pressure, $\boldsymbol{\tau}=\tau\hat{\boldsymbol{x}}$ is the surface wind stress, $\boldsymbol{g}=g\hat{\boldsymbol{z}}$ is the gravitational acceleration, T is the three-dimensional temperature, and all other constant parameters are given in table 2.1 in section 2. ∇_h^2 and ∇_v^2 are the horizontal and vertical components of the vector Laplacian.

A lot is happening in equation 1.2, so we will break down the terms one-by-one. The first equation, 1.2a, is the continuity equation and enforces conservation of mass when density is constant [19]. We are operating under the Boussinesq approximation, a common approach to fluid dynamics that takes the density to be constant in all terms except for buoyancy ($\boldsymbol{g}\alpha\,(T-T_0)$ in equation 1.2b). In practice, Boussinesq hydrodynamics gives an incompressible flow (a close approximation for most liquids, incompressibility mainly excludes the formation of sound waves) while still allowing for buoyancy [19].

The second equation, 1.2, is derived from the Navier-Stokes equations, which are a set of three momentum-balance equations that underpin much of the theory of fluid dynamics. The terms $\partial_t\boldsymbol{u}+(\boldsymbol{u}\cdot\boldsymbol{\nabla})\,\boldsymbol{u}$ (total acceleration), $-\frac{1}{\rho_0}\boldsymbol{\nabla}p$ (pressure forces), $\nu_h\nabla_h^2\boldsymbol{u}+\nu_z\nabla_z^2$ (viscous forces), and $\boldsymbol{g}\alpha\,(T-T_0)$ (buoyancy) follow directly from the Navier-Stokes equations [19]. Additional terms are added to account for the effects of Stokes drift and the Coriolis force: $\boldsymbol{f}\times(\boldsymbol{u}+\boldsymbol{u_s})$ is the Coriolis effect; the correction to the pressure $-\frac{1}{\rho_0}\boldsymbol{\nabla}\left(\frac{\rho_0}{2}\left(|\boldsymbol{u}+\boldsymbol{u_s}|^2-|\boldsymbol{u}|^2\right)\right)$ accounts for the Stokes drift; $\boldsymbol{u_s}\times\boldsymbol{\nabla}\times\boldsymbol{u}$ is the wave-influenced vortex force [18]; $\frac{\boldsymbol{\tau}}{\rho_0}$ is the forcing caused by wind along the surface of the water.

The final equation, 1.2c, describes heat conservation [3]. The LHS, $\partial_t T+((\boldsymbol{u}+\boldsymbol{u_s})\cdot\boldsymbol{\nabla})\,T$, is analogous to the first two terms on the LHS of the momentum equation and gives the time rate of change of temperature T for a parcel of fluid moving with the velocity field $\boldsymbol{u}$. Moving on to the RHS, $\frac{q}{c_p\rho_0}$ describes the rate of heat loss at the water's surface. The final two terms $\kappa_h\nabla_h^2 T + \kappa_z\nabla_z^2 T$ describe heat loss due to thermal diffusion.

Two of the terms in equation 1.2, wind forcing $\frac{\tau}{\rho_0}$ and radiative heat loss $\frac{q}{c_p\rho_0}$, only act at the surface of the water. When implementing the equations of motion in the simulation, those terms appear as boundary conditions on the x-velocity u and the z-derivative of temperature $\frac{dT}{dz}$ rather than as terms in the equations.

The Craik-Leibovich equations are far from trivial, but they have proven to be robust and continue to be used for studies of Langmuir turbulence today. The familiarity of the equations of motion and the extensive literature on its peculiarities greatly simplifies the computational task ahead.

The complex dynamics of Langmuir turbulence also present a test of the flexibility of DSS. Langmuir circulation relies on the coupling of mixed turbulence scales: small-scale eddies (swirls of water, like the vortex tubes in figure 1.3b) interact with larger-scale flows like the Stokes drift in order to produce the overall pattern. The quasilinear approximation required by DSS, described in more detail in section 1.4.1, decouples eddy scales from one another. Eddies can interact with mean fields or with the subset of other eddies that produce mean fields, but they cannot interact with other eddies to produce more eddies (called eddy-eddy scattering). Langmuir turbulence will therefore provide a useful test of whether DSS can still capture features with complex eddy interactions.

1.4 A Lengthier Introduction to Direct Statistical Simulation

We now turn to the meat of Direct Statistical Simulation. In numerical simulations, the climate is a sum of interacting but distinct physical processes: we feed sub-grid models into global models, couple atmospheric models to oceanic models, and collect climate statistics by tracking the aggregate behavior of small scale processes over long periods of time. Intuitive and useful, but not necessarily instructive. The theoretical development of DSS emerged as an alternative to the numerical approach. By reimagining the mathematical framework of climate models, DSS attempts to answer fundamental questions about climate as a distinct dynamical system, not merely a sum of interacting physical processes. In numerical simulations, the variables are descriptive physical features – temperature, pressure, trace gas or chemical content –

and the outputs are the climate system at various points in time. Climate statistics, the outputs we are actually interested in, are calculated indirectly based on the behavior of the climate system over time. In DSS, the variables *are* the climate statistics and the governing equations tell us how those statistics evolve over time and interact with each other directly. Instead of tracking air temperature and solar radiation absorption and then crunching out trends and patterns, DSS follows storm tracks, average rainfall, and ocean currents. With this approach, we believe that a more fundamental understanding of climate can emerge. By directly [observing] how each climate statistic affects the others, DSS should provide a more intuitive perspective on the climate system as a whole.

In order to test the efficacy of DSS on Langmuir turbulence, we need to compare a model that uses DSS to a control. The control in this case is a fully nonlinear (NL) simulation, which directly integrates the time derivatives in the equations of motion (see equation 1.2). The NL simulation is like a high-resolution DNS model, reproducing what we believe to be the actual physical conditions of Langmuir turbulence. It pays for that accuracy by being computationally expensive, hence the need for DSS in the first place. The NL simulation will be compared to two different versions of DSS. One uses the quasilinear (QL) approximation, which is a common approach to computational fluid dynamics that essentially decouples small-scale from large-scale eddies. The other uses the generalized quasilinear approximation (GQL), a variation on QL developed by Marston and colleagues that allows more interaction between eddy scales with only a small projected increase in computation time. We expect to find that Langmuir turbulence simulated with the QL approximation is fairly close to the NL simulation results but that the GQL approximation performs even better, an argument for this modified version of QL to be put to use in future implementations of DSS.

1.4.1 Quasilinear Approximation

Mathematically, the quasilinear approximation involves decomposing each state variable into two parts, a mean and a fluctuation. State variables for our purposes include temperature T, pressure p, and the three components of velocity u, v, and w (along the x-, y-, and z-axes). We can describe the QL decomposition for a generalized state variable q as follows:

$$q = \overline{q} + q' \tag{1.3}$$

where $\overline{q}$ is the mean field while q' is the fluctuation. The decomposition in equation 1.3 is known as a Reynolds decomposition, and it comes with three key properties:

$$\overline{q'} = 0 \tag{1.4}$$

$$\overline{\overline{q}} = \overline{q} \tag{1.5}$$

$$\overline{a\,q + b} = a\,\overline{q} + b \tag{1.6}$$

where a and b can either be scalars or vary over a direction not involved in the mean $\overline{q}$.

One of the impressive flexibilities of DSS is in its definition of the mean; in general, $\overline{q}$ can be a temporal mean, a spatial mean, or an ensemble mean [20]. In this case, however, we are studying spatial means, so $\overline{q}$ is the average over the horizontal plane. In the context of our Langmuir turbulence simulation, the state variable decomposition in equation 1.3 means that the water temperature, for example, is stored in two parts: $\overline{T}$ is the average of the temperature in the xy-plane (which varies only with depth z) while T' contains the deviations from those averages.

Each equation of motion is then split into two: what was once $\mathrm{dt}(q)$ is now two equations, one for $\mathrm{dt}(\overline{q})$ and the other for $\mathrm{dt}(q')$. In order to determine which terms belong in which equation, we need to distinguish between linear and non-linear terms. A linear term, denoted in general form by $\mathcal{L}[q]$, involves only one state variable or its derivatives. A non-linear term, denoted by $\mathcal{Q}[q, q]$, involves the product of two state variables, their derivatives, or a state variable with its own derivative. Below are the equations of motion for Langmuir turbulence with non-linear terms picked out in red and linear terms in regular type, to provide a more concrete example:

$$\boldsymbol{\nabla} \cdot \boldsymbol{u} = 0 \tag{1.7a}$$

$$\begin{aligned}\partial_t \boldsymbol{u} + (\boldsymbol{u} \cdot \boldsymbol{\nabla})\,\boldsymbol{u} + \boldsymbol{f} \times (\boldsymbol{u} + \boldsymbol{u_s}) = &-\frac{1}{\rho_0}\boldsymbol{\nabla}\left(p + \frac{\rho_0}{2}\left(|\boldsymbol{u} + \boldsymbol{u_s}|^2 - |\boldsymbol{u}|^2\right)\right) \\ &+ \nu_h \nabla_h^2 \boldsymbol{u} + \nu_z \nabla_z^2 \boldsymbol{u} \\ &+ \boldsymbol{u_s} \times \boldsymbol{\nabla} \times \boldsymbol{u} \\ &+ \frac{\boldsymbol{\tau}}{\rho_0} \\ &+ \boldsymbol{g}\alpha\,(T - T_0)\end{aligned} \tag{1.7b}$$

$$\partial_t T + ((\boldsymbol{u} + \boldsymbol{u_s}) \cdot \boldsymbol{\nabla})\,T = \frac{q}{c_p \rho_0} + \kappa_h \nabla_h^2 T + \kappa_z \nabla_z^2 T \tag{1.7c}$$

The quasilinear approximation dictates how we treat the non-linear terms, which represent interactions between and among mean fields $\overline{q}$ and eddies or fluctuations q'. We will approximate the large-scale behavior of the system by discarding any non-linear terms that involve eddy-eddy interactions that produce more eddies (as opposed to two eddies that interact just right to produce a mean field). Discarding eddy-eddy scattering terms, which is the real substance of the quasilinear approximation, is described in more detail just after equation 1.13, below.

Now that we have identified the linear and non-linear terms, we can figure out how to split each equation into $\mathrm{dt}(\overline{q})$ and $\mathrm{dt}(q')$. There are a few ways to explain this; I will start with an algebraic approach, which will give a generalized form for the two time derivatives. Once we have the equations, I will take a more intuitive approach by showing how to sort the terms of one of the Langmuir equations of motion between $\mathrm{dt}(\overline{q})$ and $\mathrm{dt}(q')$.

For a generic equation of the form

$$\mathrm{dt}(q) = \mathcal{L}[q] + \mathcal{Q}[q, q] \tag{1.8}$$

we can apply a horizontal average over both sides

$$\overline{\mathrm{dt}(q)} = \overline{\mathcal{L}[q]} + \overline{\mathcal{Q}[q, q]} \tag{1.9}$$

For linear operators, a horizontal average can slip inside the operator. For non-linear operators, on the other hand, we have to expand over the possible interactions by plugging in $q = \overline{q} + q'$:

$$\begin{aligned} \mathrm{dt}(\overline{q}) &= \mathcal{L}[\overline{q}] + \overline{\mathcal{Q}[\overline{q} + q', \overline{q} + q']} \\ &= \mathcal{L}[\overline{q}] + \overline{\mathcal{Q}[\overline{q}, \overline{q}]} + \overline{\mathcal{Q}[\overline{q}, q']} + \overline{\mathcal{Q}[q', \overline{q}]} + \overline{\mathcal{Q}[q', q']} \end{aligned} \tag{1.10}$$

The middle two non-linear terms, $\overline{\mathcal{Q}[\overline{q}, q']}$ and $\overline{\mathcal{Q}[q', \overline{q}]}$, are subject to the scalar rule for Reynolds decomposition (equation 1.6), where we can treat $\overline{q}$ itself as scalar-like because it varies only over a direction orthogonal to the horizontal average:

$$\begin{aligned} \overline{\mathcal{Q}[\overline{q}, q']} &=> \overline{\overline{q} \cdot q'} \\ &= \overline{\overline{q}} \cdot \overline{q'} \\ &= \overline{q} \cdot \overline{q'} \\ &= 0 \end{aligned}$$

where we used $\overline{\overline{q}} = \overline{q}$ (equation 1.5) and $\overline{q'} = 0$ (equation 1.4). The operator $\cdot$ is meant to represent a generalized interaction; the non-linear term could involve derivatives rather than straight products, but the same reduction steps would apply regardless. This leaves us with the full generalized expression for $\mathrm{dt}(\overline{q})$:

$$\mathrm{dt}(\overline{q}) = \mathcal{L}[\overline{q}] + \mathcal{Q}[\overline{q}, \overline{q}] + \overline{\mathcal{Q}[q', q']} \tag{1.11}$$

Now that we have $\mathrm{dt}(\overline{q})$, we can actually solve for $\mathrm{dt}(q')$ through a straightforward substitution. Start by taking a time-derivative of the Reynolds decomposition:

$$\begin{aligned} q &= \overline{q} + q' \\ \mathrm{dt}(q) &= \mathrm{dt}(\overline{q}) + \mathrm{dt}(q') \\ \mathrm{dt}(q') &= \mathrm{dt}(q) - \mathrm{dt}(\overline{q}) \end{aligned}$$

We can then substitute the generalized expression for $\mathrm{dt}(q)$ (equation 1.8), plugging in $q = \overline{q} + q'$, and the one we just derived for $\mathrm{dt}(\overline{q})$ (equation 1.11):

$$\begin{aligned} \mathrm{dt}(q') &= \mathcal{L}[q] + \mathcal{Q}[q, q] - \Big(\mathcal{L}[\overline{q}] + \mathcal{Q}[\overline{q}, \overline{q}] + \overline{\mathcal{Q}[q', q']}\Big) \\ &= \mathcal{L}[\overline{q} + q'] + \mathcal{Q}[\overline{q} + q', \overline{q} + q'] - \mathcal{L}[\overline{q}] - \mathcal{Q}[\overline{q}, \overline{q}] - \overline{\mathcal{Q}[q', q']} \\ &= \mathcal{L}[\overline{q}] + \mathcal{L}[q'] + \mathcal{Q}[\overline{q}, \overline{q}] + \mathcal{Q}[\overline{q}, q'] + \mathcal{Q}[q', \overline{q}] + \mathcal{Q}[q', q'] \\ &\quad - \mathcal{L}[\overline{q}] - \mathcal{Q}[\overline{q}, \overline{q}] - \overline{\mathcal{Q}[q', q']} \end{aligned} \tag{1.12}$$

expanding $\mathcal{Q}[\overline{q} + q', \overline{q} + q']$ just as we did in equation 1.10. Now cancel matching terms:

$$\mathrm{dt}(q') = \mathcal{L}[q'] + +\mathcal{Q}[\overline{q}, q'] + \mathcal{Q}[q', \overline{q}] + \mathcal{Q}[q', q'] - \overline{\mathcal{Q}[q', q']} \tag{1.13}$$

Everything we have done so far is "real" math, but this next step is where the quasilinear approximation actually happens. The expression $\mathrm{dt}(q')$ describes the time-evolution of eddies (fluctuations q'), so the right-hand side gives all of the interactions that can influence the eddy dynamics. The magic of the quasilinear approximation comes from killing the last two terms, $\mathcal{Q}[q', q']$ and $-\overline{\mathcal{Q}[q', q']}$. Physically, this prevents eddy-eddy interactions ($\mathcal{Q}[q', q']$) from affecting other eddies ($\mathrm{dt}(q')$), which we call eddy-eddy scattering. Instead, the only interactions that can affect $\mathrm{dt}(q')$ are between an eddy and a mean field ($\overline{q}, q'$).

Once we've crossed out the eddy-eddy scattering terms, we end up with our generalized expression for $\mathrm{dt}(q')$, displayed below along with $\mathrm{dt}(\overline{q})$ so that we have them both in one place:

$$\mathrm{dt}(q') = \mathcal{L}[q'] + \mathcal{Q}[\overline{q}, q'] + \mathcal{Q}[q', \overline{q}] \tag{1.14}$$
$$\mathrm{dt}(\overline{q}) = \mathcal{L}[\overline{q}] + \mathcal{Q}[\overline{q}, \overline{q}] + \overline{\mathcal{Q}[q', q']} \tag{1.15}$$

Ok, so we have the generalized equations of motion. What does this look like in practice? We'll use one of the equations of motion for Langmuir turbulence as an example of how to split and sort non-linear terms. Recall that the momentum balance equation 1.2b for Langmuir turbulence is actually three equations, since $\boldsymbol{u} = \{u, v, w\}$ is a vector of state variables. We will look at the x-component of this equation, $\mathrm{dt}(u)$. The un-expanded form is

$$\begin{aligned}\partial_t u + (\boldsymbol{u} \cdot \boldsymbol{\nabla_x})\, u + (\boldsymbol{f} \times (\boldsymbol{u} + \boldsymbol{u_s}))_x = &-\frac{1}{\rho_0}\boldsymbol{\nabla_x}\left(p + \frac{\rho_0}{2}\left(|\boldsymbol{u} + \boldsymbol{u_s}|^2 - |\boldsymbol{u}|^2\right)\right) \\ &+ \nu_h \nabla_h^2 u \\ &+ (\boldsymbol{u_s} \times \boldsymbol{\nabla} \times \boldsymbol{u})_x \end{aligned} \tag{1.16}$$

where the subscript x indicates the x-component of an operator or expression and we have dropped the $\nu_z \nabla_z^2 \boldsymbol{u}$ and $\boldsymbol{g}\alpha\,(T - T_0)$ because they act only in the vertical direction and the $\frac{\tau}{\rho_0}$ because it is incorporated into our problem as a boundary condition. Expanding out the dot products and cross products and applying the derivative operator ∇_x and the Laplacian ∇_x^2, we have for the x-direction (still in the fully non-linear case, here):

$$\begin{aligned}\mathrm{dt}(u) + \frac{1}{\rho_0}\mathrm{dx}(p) - \nu_h(\mathrm{dx}^2(u) + \mathrm{dy}^2(u)) - \nu_z \mathrm{dz}^2(u) - fv + \mathrm{dx}(uu_s) \\ = -u\mathrm{dx}(u) - v\mathrm{dy}(u) - w\mathrm{dz}(u)\end{aligned} \tag{1.17}$$

where I have placed the linear terms on the left-hand side and the non-linear terms on the right. I could just plug in $q = \overline{q} + q'$ and start expanding each term, but that would get messy very quickly. Instead, recall that the linear terms are divided easily between $\mathrm{dt}(\overline{q})$ and $\mathrm{dt}(q')$: $\mathcal{L}[\overline{q}]$ go with $\mathrm{dt}(\overline{q})$ and $\mathcal{L}[q']$ go with $\mathrm{dt}(q')$. It is only the non-linear term that must be expanded by hand: $\mathcal{Q}[\overline{q} + q', \overline{q} + q'] = \mathcal{Q}[\overline{q}, \overline{q}] + \mathcal{Q}[\overline{q}, q'] + \mathcal{Q}[q', \overline{q}] + \mathcal{Q}[q', q']$. Applying this expansion to the non-linear terms on the right-hand side of equation 1.17:

$$\begin{aligned}-u\mathrm{dx}(u) - v\mathrm{dy}(u) - w\mathrm{dz}(u) = &- \overline{u}\mathrm{dx}(\overline{u}) - \overline{v}\mathrm{dy}(\overline{u}) - \overline{w}\mathrm{dz}(\overline{u}) \\ &- \overline{u}\mathrm{dx}(u') - \overline{v}\mathrm{dy}(u') - \overline{w}\mathrm{dz}(u') \\ &- u'\mathrm{dx}(\overline{u}) - v'\mathrm{dy}(\overline{u}) - w'\mathrm{dz}(\overline{u}) \\ &- u'\mathrm{dx}(u') - v'\mathrm{dy}(u') - w'\mathrm{dz}(u') \end{aligned} \tag{1.18}$$

In this form, it is not so hard to sort terms: as per equations 1.11 and 1.14, any $\mathcal{Q}[\overline{q}, \overline{q}]$ or $\mathcal{Q}[q', q']$ term goes in $\mathrm{dt}(\overline{q})$, leaving $\mathcal{Q}[\overline{q}, q']$ and $\mathcal{Q}[q', \overline{q}]$ for $\mathrm{dt}(q')$. Then it is just a matter of actually writing out the quasilinear equations of motion, sorting the terms as described above:

$$\begin{aligned}\mathrm{dt}(\overline{u}) &+ \frac{1}{\rho_0}\mathrm{dx}(\overline{p}) - \nu_h(\mathrm{dx}^2(\overline{u}) + \mathrm{dy}^2(\overline{u})) \\ &- \nu_z\mathrm{dz}^2(\overline{u}) - f\overline{v} + \mathrm{dx}(\overline{u}u_s) \\ &= -\overline{u}\mathrm{dx}(\overline{u}) - \overline{v}\mathrm{dy}(\overline{u}) - \overline{w}\mathrm{dz}(\overline{u}) \\ &\quad - u'\mathrm{dx}(u') - v'\mathrm{dy}(u') - w'\mathrm{dz}(u')\end{aligned} \tag{1.19}$$

$$\begin{aligned}\mathrm{dt}(u') &+ \frac{1}{\rho_0}\mathrm{dx}(p') - \nu_h(\mathrm{dx}^2(u') + \mathrm{dy}^2(u')) \\ &- \nu_z\mathrm{dz}^2(u') - fv' + \mathrm{dx}(u'u_s) \\ &= -\overline{u}\mathrm{dx}(u') - \overline{v}\mathrm{dy}(u') - \overline{w}\mathrm{dz}(u') \\ &\quad - u'\mathrm{dx}(\overline{u}) - v'\mathrm{dy}(\overline{u}) - w'\mathrm{dz}(\overline{u})\end{aligned} \tag{1.20}$$

That gives us one of the five equations of motion for Langmuir turbulence in QL form. Conversion for all equations follows the same procedure outlined above, so I will not walk through the rest of them.

Discarding eddy-eddy scattering as dictated by the quasilinear approximation may seem extreme, but it has proven effective at representing physical systems with strong mean flows. The simplest form of DSS, which proceeds with the method outlined above, has successfully described plasmas on giant planets [21], wall-bounded shear-flow turbulence [22, 23], the growth of the dry atmospheric convective boundary layer [24], and magnetorotational instability in accretion discs [25].

As physical systems are driven further from equilibrium, however, the limitations of the quasilinear approximation quickly become apparent. Tobias et al. found good agreement of DSS with DNS for the problem of driving β-plane jets but the approach failed as the system was driven further from equilibrium [26, 10]. If DSS is to succeed across a large variety of physical systems far from equilibrium, as it must to represent the intricacies of climate, a new approach is in order.

1.4.2 Generalized Quasilinear Approximation

Marston and colleagues have developed a modified approach to DSS that allows some but not all eddy-eddy scattering rather than discarding them entirely as the quasi-linear approximation does. The generalized quasilinear (GQL) approximation takes a more lenient approach to the state variable decomposition and eddy-eddy interactions. Rather than strictly defining the variable q as the sum of a mean field and a fluctuation, the GQL approximation splits q by wavenumber: the sum of all components of q with a wavenumber below some threshold Λ form $\overline{q}$ (which oscillates slowly in the horizontal plane) while all wavenumbers above Λ go into q' (which oscillates rapidly). The equations of motion are still split into $\mathrm{d}(\overline{q})$ and $\mathrm{d}(q')$, but when cancelling non-linear terms we now allow some extra eddy-eddy interactions. QL only allows eddy-eddy into mean field scattering, which means two components with non-zero wavenumbers interact to form something with a wavenumber of zero. GQL, on the other hand, includes any eddy-eddy scattering that produces a wavenumber below Λ, so two components of with wavenumbers higher than Λ can form *either* something with a wavenumber of zero *or* another eddy so long as it has a wavenumber below Λ. By retaining or discarding terms in this fashion, we allow some interactions between high-wavenumber eddies to form other eddies rather than just mean fields.

With the math behind us, it is important to emphasize some critical features of GQL [10]. First, GQL interpolates between QL and NL through the wavenumber threshold Λ. If we set $\Lambda = 0$, then $\overline{q}$ becomes the mean field, q' is the fluctuation, and we retrieve the QL case. If we allow $\Lambda \to \infty$, then $\overline{q} \to q$, q' empties out, and we have the fully non-linear case again. By adjusting Λ, we can choose how to trade off between exactness (NL) and computational speed and simplicity (QL). Second, by retaining the extra interaction terms that are left out of QL, GQL preserves linear and quadratic conservation laws including conservation of energy, angular momentum, and enstrophy (the integral of the vorticity, an important quantity in fluid dynamics). Third, since some eddy-eddy scattering is now allowed, it is possible for energy to flow between the small and large turbulent scales. By forbidding eddy-eddy interactions in QL, we prevent small-scale eddies and large-scale eddies from interacting at all, whereas GQL softens this barrier with Λ. Finally, and this is a more theoretical point, QL and GQL are not fully statistical. They are actually a half-way step to the purely statistical approach, the 2nd order (generalized) cumulant expansions CE2 and GCE2 (similar to Stochastic Structural Stability Theory (S3T) [23, 27]). By "purely statistical", I mean that the only dynamic variables are the first and second cumulants, or statistical moments. In general, CE2 and GCE2 are approximations that involve truncating the infinite set of cumulant expansion equations at 2nd order. However, for the approximations used in QL and GQL, CE2 and GCE2 are exact in the sense that they require no other approximations in order to achieve closure of the equations of motion. Since CE2 and GCE2 are more difficult to implement, it is easier to work with QL and GQL as was done for this project.

1.5 Dedalus and Spectral Methods

We have a physical system and its equations of motion, but to carry out the simulations and analyze their results we need a computational fluid dynamics software package. For this project, we chose Dedalus, an open-source program written in Python that solves systems of partial differential equations in spectral (Fourier) space [28]. Dedalus has a variety of useful features, including:

- Spectral space representation
- Both periodic and non-periodic boundary conditions
- Plain-text entry of equations
  - for example, $\boldsymbol{\nabla} \cdot \boldsymbol{u} = 0$ is entered as `‘‘dx(u) + dy(v) + dz(w) = 0’’`
- MPI parallelization
- Flexible analysis tools
  - Integration, differentiation, interpolation, basic functions
- HDF5 file format
  - Organizes output files into a folder-like structure that includes metadata

Spectral space representation, as opposed to real space, makes Dedalus particularly well-suited to DSS analysis. When variables are represented in real space, they are stored as a three-dimensional array of values, where each element of the array is the value of the variable at that point in space; this should sound familiar from our discussion of DNS climate modeling in section 1.1 In spectral space, Fourier coefficients are stored instead. Fourier transformations allows us to convert data back and forth with ease, but some problems are much easier to solve in one representation than the other. In the case of DSS, the QL and GQL approximations are spectral statements: they dictate which Fourier modes (i.e. mean fields with a wavenumber of zero vs fluctuations with a non-zero wavenumber) can interact with each other. Particularly for GQL, where we want to set a threshold wavenumber Λ and use different equations of motion depending on which Fourier mode of the state variable we are considering, the equations of motion are much easier to manage in software that uses spectral space.

The other key feature of Dedalus is its treatment of boundary conditions. For a problem in three dimensions like Langmuir turbulence, three bases must be chosen on which to represent the variables and equations of motion, one for each direction. The vertical z direction is not periodic, as there is a distinct physical difference between the top of the simulated domain (the surface of the ocean on which wind and waves act) and the bottom (the deep ocean). By contrast, the two horizontal directions are periodic; there is nothing special about the sides of our domain, they just continue out into more water, so we can choose a periodic basis to represent the x and y directions. Dedalus supports up to one non-periodic basis for any given problem, so we are able to represent the vertical direction in our simulation using (non-periodic) Chebyshev polynomials while the horizontal directions use a (periodic) Fourier basis.

To actually solve the equations of motion, Dedalus uses an approach called the pseudo-spectral algorithm. It begins by choosing a set of trial functions to form a basis; as mentioned above, for our problem this is a Fourier basis for the horizontal directions and a Chebyshev polynomial basis for the vertical. Each basis type defines a set of "collocation points" at which the equations of motion will actually be solved; these collocation points are not evenly spaced in general and vary by basis type. At each collocation point, the equations of motion and the initial conditions are Fourier transformed into spectral space and solved exactly. The solutions at each point are made up of a finite sum of weighted basis functions while the solution in between points is interpolated. To advance the simulation in time, Dedalus applies numerical integration in spectral space. By repeating this process over and over with small time steps for each iteration, a time series of the motion is built up.

1.6 This Project, in Brief

Ultimately, this project compares NL, QL, and GQL simulations of Langmuir turbulence in the ocean surface boundary layer. We will evaluate the accuracy of QL and GQL approaches to Langmuir turbulence simulation by comparing them to the NL case, which serves as a direct numerical simulation (DNS) benchmark. To minimize the influence of random numerical variations between simulations, we will average the results across an ensemble of 10 simulations per case (NL, QL, and GQL). Each ensemble member will be identical to the others except for the random number seed used to generate the initial conditions. For each simulation type, we will confirm the presence and dimensions of Langmuir cells by studying horizontal midplanes of the vertical velocity. We will then go on to compare vertical profiles of the mean fields, momentum flux, and turbulent temperature transport to track the evolution of the mixed layer and evaluate the relative performance of the QL and GQL approaches.

We expect that QL will produce a rough facsimile of Langmuir turbulence but that GQL will be even more accurate. The success of both, even in producing a rough approximation, will validate the ability of DSS to represent a variety of physical systems. If GQL can produce an accurate model of Langmuir turbulence, we will be one step closer to deploying DSS in a full-scale sub-grid climate model.

Chapter 2

Method

In order to set up the Langmuir turbulence simulations, we need to establish six components: domain, equations of motion, parameters, boundary conditions, initial conditions, and computational settings. All of these components are communicated to Dedalus for each simulation through a Python script, which is available for reference in Appendices A, B, and C along with annotations. The simulation conditions are based on McWilliams et al (1997) [15] as modified in Ref. [8], although the domain size and resolution have been reduced in this work.

Instead of simulating the entire ocean, which would be overkill given our stated purpose of studying a subgrid phenomenon that itself is only a few hundred meters in size, we run the Langmuir turbulence simulations over a finite domain. Since Langmuir turbulence is an ocean surface phenomenon, the vertical dimension of the domain is constrained by the mixed layer depth of the ocean. The mixed layer can be anywhere from 10 m to 150 m deep depending on the location and season, but for these simulations the initial mixed layer is 32.1 m deep for consistency with McWilliams et al. We set our vertical domain dimension at 64.8 m, approximately double the initial mixed layer depth, since we are interested in studying how Langmuir turbulence deepens the mixed layer over time. The horizontal dimensions, which need to be large enough to capture multiple Langmuir cells, are 288 m by 288 m for consistency with Ref. [8]. For the spectral representation of the domain, there are 72 modes in the vertical direction and 96 modes in each of the horizontal directions.

A schematic of the domain can be found in figure 2.1. At the top of the domain, which represents the interface between the ocean and the atmosphere, there are three major physical processes: surface cooling, wind forcing, and wave forcing. The surface cooling, set by $q = -5$ W/m^2, is included for consistency with McWilliams et al. and helps to destabilize the flow and kick-start turbulence. Wind forcing, set by $\tau = 0.037$ N/m^2, is incorporated as a boundary condition on the velocity of the

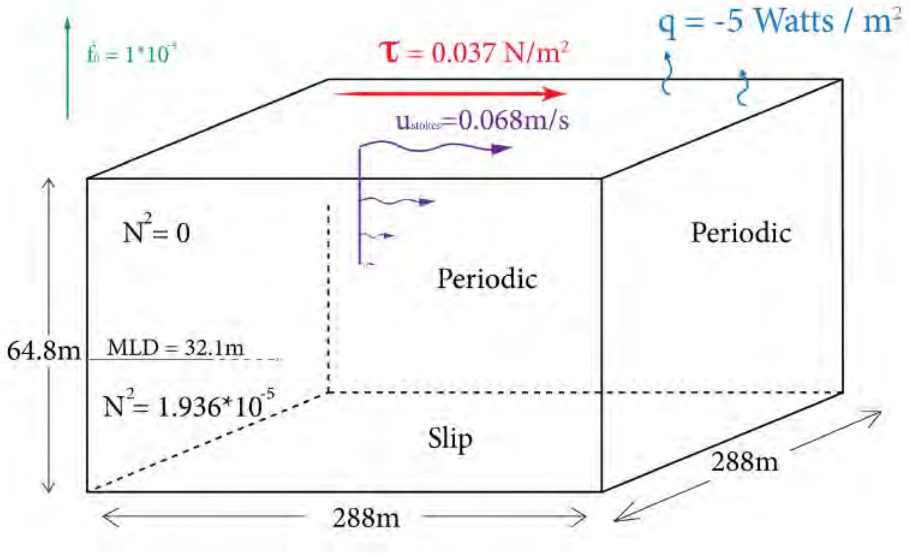


Figure 2.1: Figure 5 of Ref. [8]. Diagram of the Langmuir turbulence domain, with wind stress τ and surface cooling q at the top. The bounds are non-periodic in the vertical direction and periodic in both horizontal directions.

water at the surface. Wave forcing, represented by the Stokes drift $\boldsymbol{u_s}(z) = a_s e^{2k_s z}\hat{x}$, exponentially decays with depth and is incorporated into the equations of motion in equation 1.2.

The equations of motion for Langmuir turbulence come from the phase-averaged Craik-Leibovich equations, which can be found in section 1.3.1. The physical parameters, including viscosity, density, and wave speed, are based on McWilliams et al. [15] and Ref. [8] and are given in table 2.1.

A few parameters appear in table 2.1 that are not from the Craik-Leibovich equations (equation 1.2), specifically N^2, a_s, k_s, and MLD. The square of the Brunt-Väisälä frequency $N^2 = g\alpha\frac{\partial T}{\partial z}$ is a constant characterizing the linear temperature stratification of the deep ocean, below the mixed layer. The next two, a_s and k_s, come from the definition of the Stokes drift in equation 1.1. Finally, MLD is the initial mixed layer depth above which the temperature is uniformly equal to the background temperature T_0 and below which there is linear temperature stratification as described by the Brunt-Väisälä frequency N.

Most parameters have roughly their values as measured for the actual ocean with a few notable exceptions. The surface cooling $q = -5$ W/m^2 is chosen for consistency with McWilliams et al. [15], where it is included to destabilize and kick-start the flow. q is quite small, but this is appropriate considering that we do not want thermal convection to dominate the dynamics. The Coriolis parameter varies with latitude,

Parameter Symbol	Name	Value
f	Coriolis parameter	$1 \times 10^{-4}\ \mathrm{s}^{-1}$
ρ_0	background density	$1035\ \mathrm{kg\,m}^{-3}$
ν_h	horizontal harmonic kinematic viscosity	$2 \times 10^{-3}\ \mathrm{m}^2\,\mathrm{s}^{-1}$
ν_z	vertical harmonic kinematic viscosity	$1 \times 10^{-3}\ \mathrm{m}^2\,\mathrm{s}^{-1}$
τ	surface wind stress	$0.037\ \mathrm{N\,m}^{-2}$
g	gravitational acceleration	$9.81\ \mathrm{m\,s}^{-2}$
α	coefficient of thermal expansion	$2 \times 10^{-4}\ \mathrm{C}^{-1}$
T_0	background temperature	20 C
q	surface heat flux	$-5\ \mathrm{W\,m}^{-2}$
c_p	specific heat of sea water	$3994\ \mathrm{J\,kg}^{-1}\,\mathrm{C}^{-1}$
κ_h	horizontal thermal diffusivity	$2 \times 10^{-3}\ \mathrm{m}^2\,\mathrm{s}^{-1}$
κ_z	vertical thermal diffusivity	$1 \times 10^{-3}\ \mathrm{m}^2\,\mathrm{s}^{-1}$
N^2	(squared) Brunt-Väisälä frequency	$1.936 \times 10^{-5}\ \mathrm{s}^{-2}$
a_s	surface Stokes drift velocity	$0.068\ \mathrm{m\,s}^{-1}$
k_s	Stokes wavenumber	$0.105\ \mathrm{m}^{-1}$
MLD	mixed layer depth	32.1 m

Table 2.1: Constant parameters used in the equations of motion (equation 1.2), the Stokes drift (equation 1.1), and to describe the domain (figure 2.1). Based on Ref.s [15, 8].

but the given value of $f = 1 \times 10^{-4}$ is appropriate for mid-latitudes around 45°, where $f = 1.03 \times 10^{-4}$. The kinematic viscosity ν and thermal diffusivity κ, which essentially control energy loss to friction and heat diffusion, are much higher than their physical values, which are on the order of 1×10^{-6} to 1×10^{-7} m^2/s [29]. This is done to account for the discrete nature of the grid on which all physical variables are represented. As the grid spacing increases, so too must viscosity and diffusivity to account for the separation between grid points. This is the same reason that we distinguish different values of ν and κ for the horizontal and vertical directions: the grid spacing in the z direction is different from that in the x and y directions, so it follows that ν and κ should also be different. Incidentally, the viscosity of the ocean in many global climate simulations is many times greater than that of molasses to account for grid spacing and the drastically different time scales of the ocean and the atmosphere.

The boundary conditions, giving periodic boundaries in the horizontal directions, a slip condition at the bottom of the domain, and both surface cooling and wind stress at the top, are as follows:

$$\begin{aligned}
\frac{\mathrm{d}T}{\mathrm{d}z} &= 0 \text{ at surface,} \qquad -\frac{q}{\kappa_h c_p \rho_0} \text{ at bottom} && \text{(2.1a)}\\
\frac{\mathrm{d}u}{\mathrm{d}z} &= \frac{\tau}{\nu_h \rho_0} \text{ at surface,} \qquad 0 \text{ at bottom} && \text{(2.1b)}\\
\frac{\mathrm{d}v}{\mathrm{d}z} &= 0 \text{ at surface,} \qquad 0 \text{ at bottom} && \text{(2.1c)}\\
w &= 0 \text{ at surface,} \qquad 0 \text{ at bottom} && \text{(2.1d)}
\end{aligned}$$

The boundary conditions above are joined by a final condition on the pressure field, which replaces the condition $w = 0$ at the bottom of the domain for the nx = 0, ny = 0 Fourier mode:

$$\int_{z_{bottom}}^{z_{surface}} p\, dz = 0 \tag{2.2}$$

The pressure condition, called a pressure gauge choice, is necessary for two reasons. One, for the (0,0) Fourier mode when both nx and ny are 0 (solving for mean fields in both horizontal directions), the continuity equation 1.2a reduces to $\frac{\mathrm{d}w}{\mathrm{d}z} = 0$, which makes one of the boundary conditions in equation 2.1a redundant. This redundant boundary condition must be replaced for that mode in order to provide the proper number of boundary conditions to constrain the problem. Two, pressure p as defined in the equations of motion is gauge-invariant, meaning that only its derivative is fully

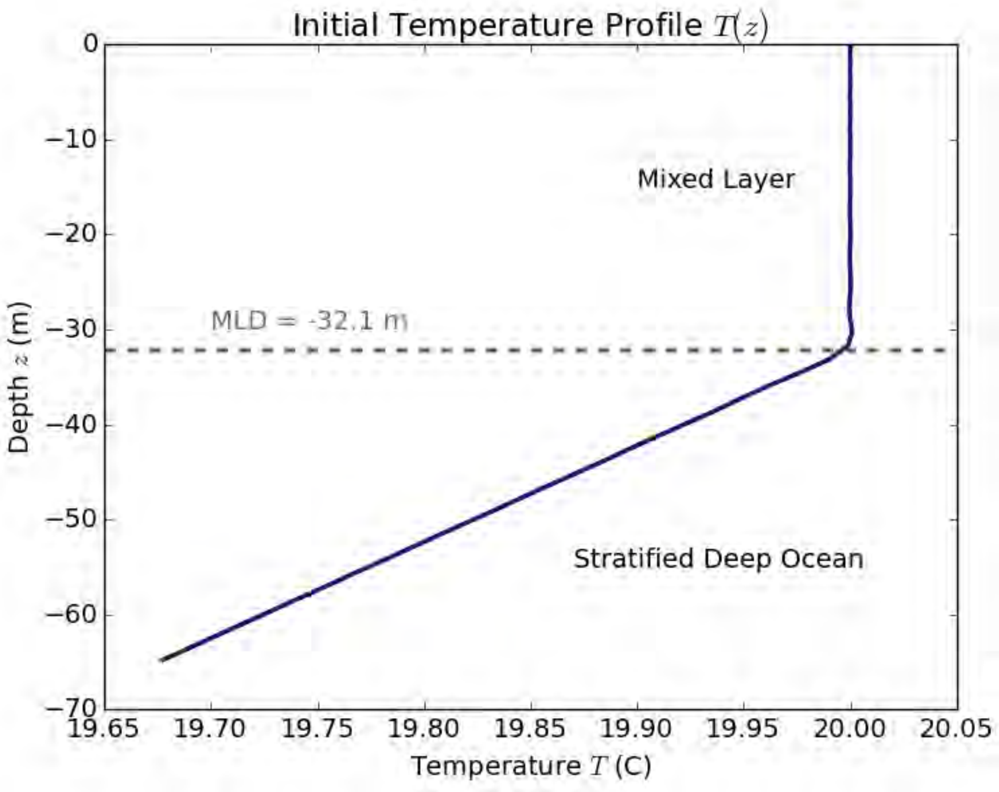


Figure 2.2: Vertical profile of the initial temperature field, labeled with the mixed layer (top 32.1 m) and deep ocean (below mixed layer).

defined while the actual value is arbitrary. We must set this arbitrary value to give the numerical solutions a reference point, so we can address both issues above by replace the redundant boundary condition for the (0,0) mode with a pressure gauge choice, equation 2.2.

The initial condition for our Langmuir turbulence simulation is defined on the temperature field; the initial velocities are set to zero for convenience, as they will be determined by wind, wave, and buoyancy forcing once the simulation starts. To mimic the physical ocean, the initial temperature has two distinct regions: a surface mixed layer and a stratified deep ocean, illustrated in figure 2.2. Above the mixed layer depth of 32.1 m, the temperature is set to a uniform background of $T_0 = 20°$ C plus random noise on the order of 1×10^{-4} in the top 5 m to provide initial instability. Below the mixed layer depth, the ocean is stratified: colder water is more dense, so the temperature decreases linearly with depth from the bottom of the mixed layer. The rate of temperature decrease in the deep ocean is set by the squared Brunt-Väisälä frequency N^2, given in table 2.1.

Each simulation is stopped after completing 78 hours of simulated time. The time step used to numerically integrate the equations of motion is set by the Courant-Friedrichs-Lewy (CFL) condition, which is a numerical constraint based on the velocities $\boldsymbol{u}$ and $\boldsymbol{u_s}$ that ensures convergence of the numerical solutions. Excluding the very beginning of each simulation, which had large time steps due to the slow onset of turbulence, the CFL condition generally gave time steps of between 7 and 10 seconds.

The output of each simulation was written to an external file every 6 hours of simulated time, giving a time-series of 13 snapshots for each simulation. Dedalus's flexible analysis tools were used to process the results of each simulation. For analyzing midplanes, the state variables – temperature T and the three components of velocity u, v, and w – were saved in their entirety, one three-dimensional array per variable. Midplanes were then produced externally in Python by interpolating a two-dimensional slice out of the three-dimensional array for a variable of interest.

Three types of vertical profiles were also saved: mean fields, momentum flux profiles, and turbulent temperature transport profiles. All three of these profiles were ensemble averaged before analysis: profiles were generated for all 10 ensemble members in a given simulation type at each of the 13 snapshots. The profiles of each ensemble member were then averaged together at each snapshot to produce a single set of ensemble averaged profiles, denoted by $\langle\rangle$. Mean fields were calculated by integrating each state variable over x and y (for QL, this just involved saving the variable $\overline{q}$), producing a vertical profile of the horizontal average of each variable as a function of z: $\overline{u}(z) = \frac{1}{\text{area}} \int u \, dx \, dy$. Momentum flux and turbulent temperature transport profiles were each determined by integrating the product of the fluctuating components of a pair of state variables over x and y. For momentum flux, this involved the product of two velocities, for example $\overline{u'v'}(z) = \frac{1}{\text{area}} \int u'v' \, dx \, dy$. For turbulent temperature transport, we take the product of a velocity and the temperature, for example $\overline{u'T'}(z) = \frac{1}{\text{area}} \int u'T' \, dx \, dy$. With three velocities and one temperature, there are three unique ensemble-averaged profiles each for momentum flux $\langle\overline{u'v'}(z)\rangle$, $\langle\overline{u'w'}(z)\rangle$, $\langle\overline{v'w'}(z)\rangle$ and turbulent temperature transport $\langle\overline{u'T'}(z)\rangle$, $\langle\overline{v'T'}(z)\rangle$, $\langle\overline{w'T'}(z)\rangle$.

The simulations were carried out using computing resources from Brown's Center for Computation and Visualization (CCV). Dedalus is, thankfully, MPI parallelized, which means that a single simulation can be divvied up between many processors that run simultaneously, considerably decreasing the computation time. By default, Dedalus will divide up the simulation domain along one of the periodic directions; if I submitted a simulation to 4 processors, for example, the domain might be split into 4 equal-sized chunks along the y-direction, where each chunk is run on a separate processor. This divide-and-conquer process can be made much more efficient by using a mesh, which splits up the domain along both the x- and y-directions. For all of the Langmuir turbulence simulations, I chose a 4 by 8 mesh (i.e. dividing the y-axis into 4 segments and the x-axis into 8), requiring $4 \cdot 8 = 32$ processors; this gave an approximately one-to-one match between simulated time and real time, so all of the simulations finished in less than 100 hours.

Chapter 3

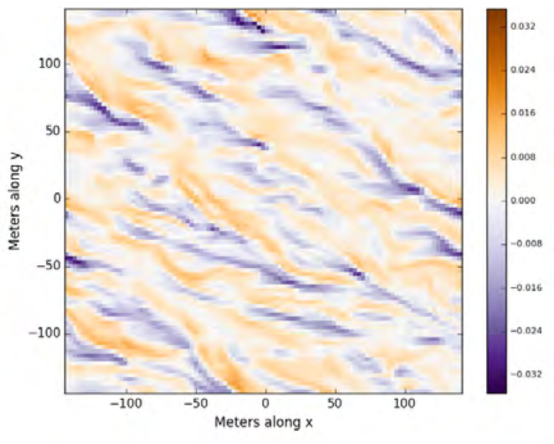
Results

The results that follow cover a 10-simulation ensemble each for the NL, QL, and GQL cases, producing a total of 30 simulations. To avoid confusion, we will use angled brackets to indicate an ensemble average as in $\langle x \rangle$ and an overbar to indicate an average over the horizontal plane $\overline{x} = \frac{1}{\text{area}} \int x \, dxdy$.

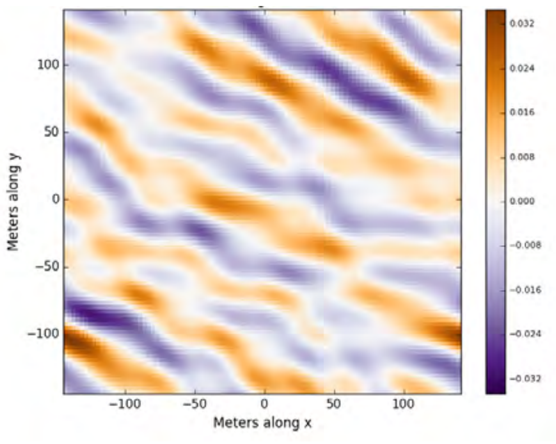
First, we will confirm that all three simulation types have produced the Langmuir turbulence pattern and compare their structural characteristics (Section 3.1). Then we will analyze mean fields, turbulent transport, and momentum flux to determine how well QL and GQL capture the dynamics of the fully non-linear Langmuir turbulence simulation (Sections 3.2 and 3.3).

3.1 Midplanes

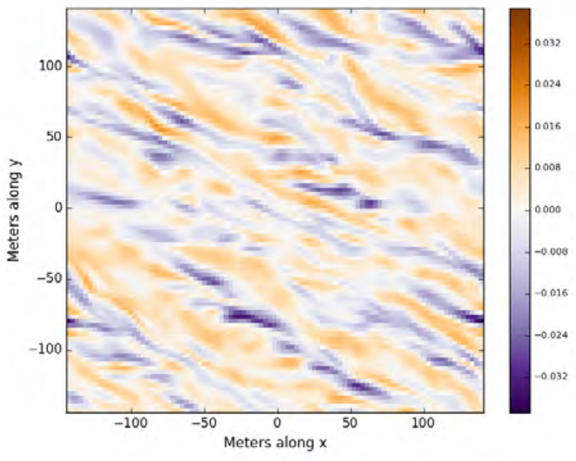
Horizontal midplanes of vertical velocity w are displayed at 18 hours of simulated time for the NL, QL, and GQL simulations in figure 3.1. We are looking from a birds-eye view down on a block of ocean water. The midplanes shown are a slice of that block taken from 8 meters below the ocean's surface. The colors correspond to vertical velocity, so brown indicates water that is moving up or out of the page while purple indicates water that is moving down or into the page. What would we expect Langmuir turbulence to look like in this representation? Recall from figure 1.4 that Langmuir cells are long tubes oriented along the ocean's surface edged by alternating strips of upwells and downwells. In a horizontal midplane of vertical velocity as shown, we would expect to see alternating strips of up-moving water (upwells, in brown) and down-moving water (downwells, in purple). All three simulations in figure 3.1 exhibit this behavior, with the Langmuir cells oriented diagonally from the upper-left to the lower-right of each midplane. As expected, the NL and GQL midplanes look particularly similar, although the GQL cells tend to be slightly thinner and more



(a) Non-linear (NL)

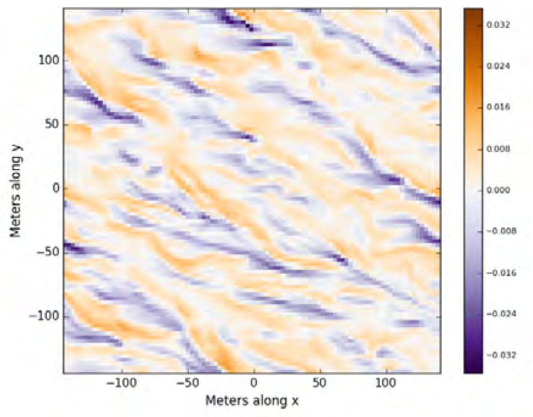


(b) Quasilinear (QL)

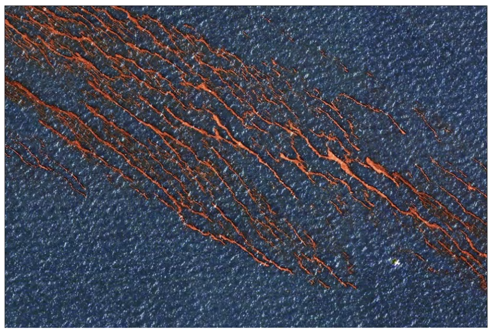


(c) Generalized Quasilinear (GQL)

Figure 3.1: Horizontal midplanes of vertical velocity w at 18 hours. From a birds-eye view: brown indicates water moving up/out of the page, purple for water moving down/into the page. Colorbar units are m/s.



(a) Horizontal midplane of vertical velocity for NL simulation case. Colorbar units are m/s.



(b) An aerial photograph from npr.org [4] taken just after the Deepwater Horizon oil spill in 2010 shows oil collecting along convergence zones to form Langmuir windrows. For a sense of scale: the small white spot in the lower-right is an airplane.

Figure 3.2: Comparing Langmuir windrows in simulation (a) and observation (b).

fragmented. Langmuir cell formation is much more regular in the QL simulation, most likely a result of the structural coherence that arises from the neglect of eddy-eddy scattering.

Magnitudes and length scales also match across simulations. Using the distances marked out on the x- and y-axes, we see that the larger Langmuir cells (each composed of a pair of adjacent upwells and downwells) are on the order of 50 m across in the NL, QL, and GQL midplanes. These length scales are roughly consistent with Langmuir cells “in the wild”, which range from a few to 100 m across. Similar velocity magnitudes are also present across all three simulations, with the colorbars topping out just above 0.032 m/s.

For comparison to Langmuir turbulence as it appears in the actual ocean, in figure 3.2 the NL simulation midplane appears alongside the oil spill image from section 1, where the purple strips of the simulation midplane correspond to the convergence zones along which the oil collects in the photograph.

3.2 Mean Field Profiles

3.2.1 Temperature Profiles

Ensemble averaged vertical temperature profiles from the NL, QL, and GQL simulations at 0 hours (the initial temperature profile) and 60 hours can be found in figure 3.3. By construction, all simulations have the same initial temperature. Even

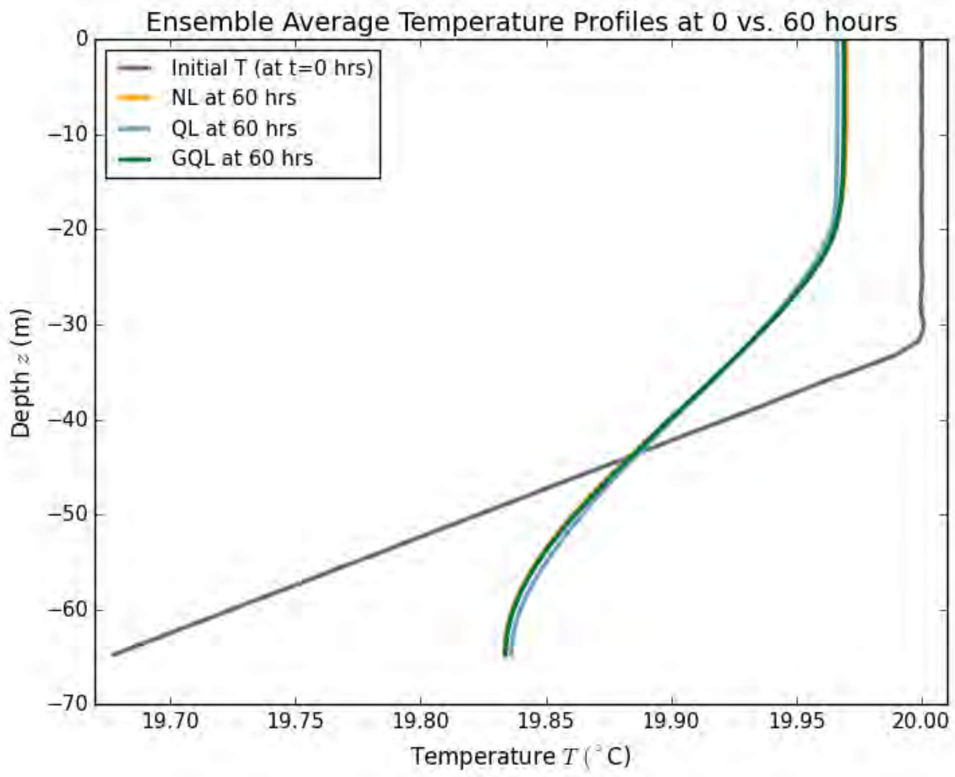


Figure 3.3: Vertical Temperature profiles at 0 and 60 hours for NL, QL, and GQL.

after 60 hours, the three profiles are remarkably similar. The QL temperature profile deviates only slightly from the other two, trending colder at the top and warmer at the bottom, which implies that QL may have enhanced vertical mixing, a possibility that is explored in the discussion in section 4.2.

A time-series of the temperature profile for the NL case, taken to be representative of all three simulation types, is shown in figure 3.4. As the simulation progresses, we observe a somewhat unexpected smoothing trend rather than a deepening one. If the mixed layer was deepening consistently over time, the vertical portion of the profile near the surface would extend deeper and deeper over the course of the simulation. Instead, we see the mixed layer actually shrink to about 20 m deep from its initial value of 32.1 m, but the deep ocean is also warming and becoming less stratified. If the simulations had continued, say for 100 hours instead of the 72 hours shown in figure 3.4, we might eventually see the two trends converge to form a mixed layer the spans the entire vertical domain. This is a significant result that will be explored in the Discussion: instead of pushing the mixed layer down from above, Langmuir turbulence appears to reach into the stratified deep ocean, cooling the mixed layer,

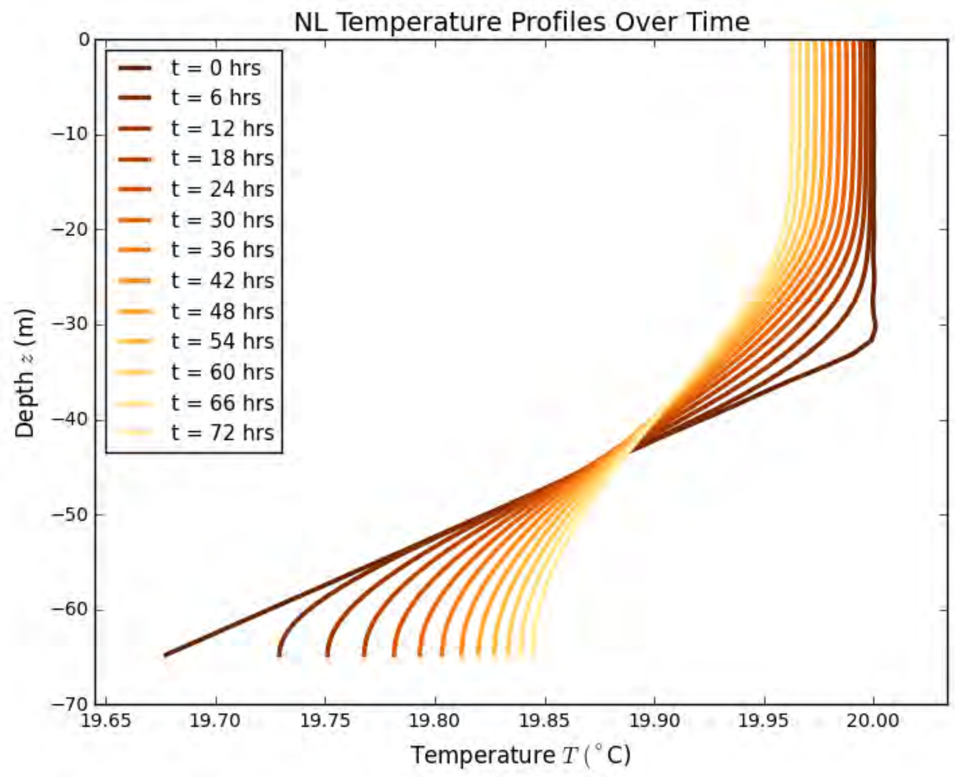


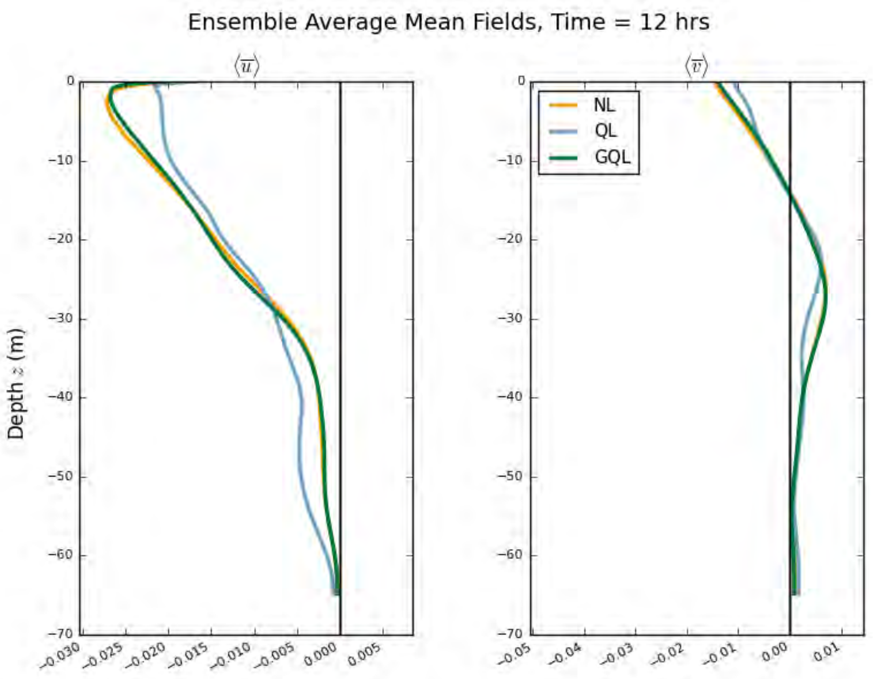
Figure 3.4: Time-series of ensemble averaged temperature profiles for the NL simulation.

warming the deep ocean, and reducing the gradient between the two to eventually form a deeper mixed layer.

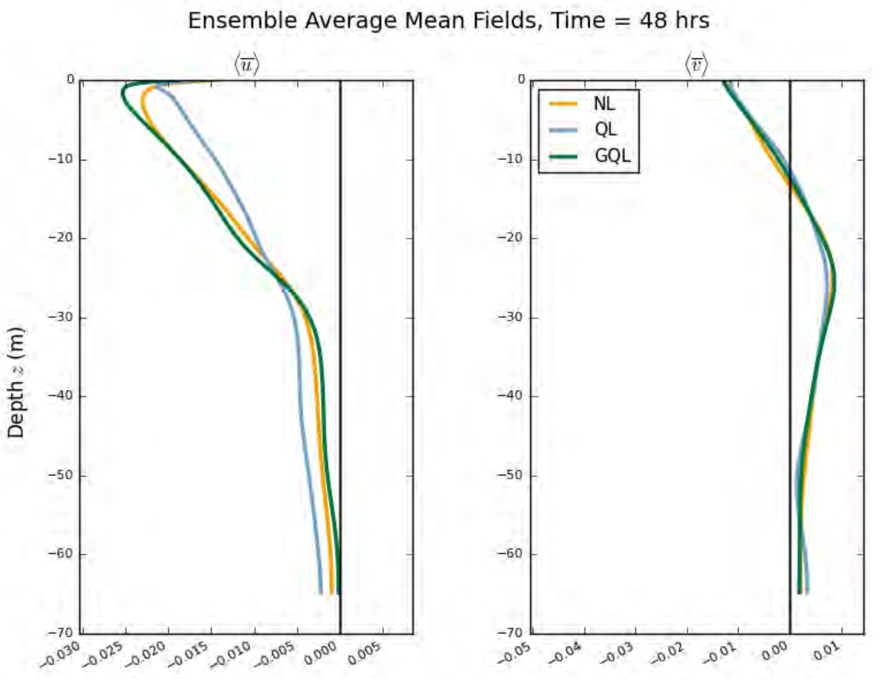
3.2.2 Velocity Mean Fields

Mean fields for velocity were constructed just like the temperature profiles by averaging each state variable in the horizontal plane and plotting those averages as a function of depth. Due to the boundary conditions on w given in equation 2.1d, which prevent flow in or out of the domain along the z-direction, the mean field $\langle \overline{w} \rangle$ is always identically zero up to numerical truncation error, so it is not shown in the following plots. Mean fields of x-velocity u and y-velocity v at 12 hours and 48 hours are shown in figure 3.5, with NL, QL, and GQL fields shown on the same plots for comparison.

We can begin with the general observation that GQL mean fields tend to track NL closely throughout the course of the simulations. Deviations are small and do not tend to be concentrated at greater depths, which is good news for consistency across



(a) 12 hours



(b) 48 hours

Figure 3.5: Vertical profiles of ensemble averaged mean velocities for NL, QL, and GQL simulations.

simulation types in the evolution of the mixed layer depth. QL mean fields tend to deviate further than GQL fields from the NL case, particularly for $\langle \overline{u} \rangle$. Overall, the QL mean fields indicate a smoothing effect, under-predicting the maxima and over-predicting the minima of the NL mean fields, the cause of which is unclear.

3.3 Correlation Profiles

Physically, there are two types of second-order correlation profiles, which refers to the horizontal average of a product of two state variables. One is momentum flux, which involves a horizontal average of the product of two velocity fluctuations $\langle \overline{u_i' u_j'} \rangle$ and describes the transfer of momentum between orthogonal directions. The other is turbulent temperature transport, which involves the horizontal average of the product of the temperature fluctuation and a velocity fluctuation $\langle \overline{u_i' T'} \rangle$ and describes the transport of (heat) energy in the direction specified by the velocity u_i.

3.3.1 Turbulent Transport

Turbulent transport describes how a flow property like temperature is moved around in the domain by the velocity field. By looking at the turbulent transport of temperature T, we can study how energy, in this case thermal energy, is distributed through convection as opposed to, for example, thermal diffusion. Convection is the dominant form of heat transport in most fluids, so it is important to understand how Langmuir turbulence facilitates the transport and mixing of heat energy in the upper ocean. The climate implications are also significant, as one of the primary processes carried out in the mixed layer and a strong motivation for improving its representation in climate models is the transport of heat between the atmosphere and the deep ocean.

A snapshot of turbulent temperature transport at 54 hours is shown in figure 3.6. I chose to display only a single time because the behavior of the QL profiles is so erratic and the fit of QL to the NL profile so consistently poor that there is not much to say about either consistencies or changes over time. As usual, the QL profiles have the worst fit, often fluctuating wildly off of the NL case and no more inclined to behave well near the bottom of the domain than they are near the surface. While GQL generally does much better, it, too, has significant deviations in some snapshots, particularly for $\langle \overline{v'T'} \rangle$, where it tends to stick much closer to zero than the NL case.

The poor performance of the QL and occasionally GQL approximations could indicate a lack of convergence among the ensemble members. If $\overline{w'T'}$, for example, tends to vary a lot from one ensemble member to another, then a 10-member ensemble

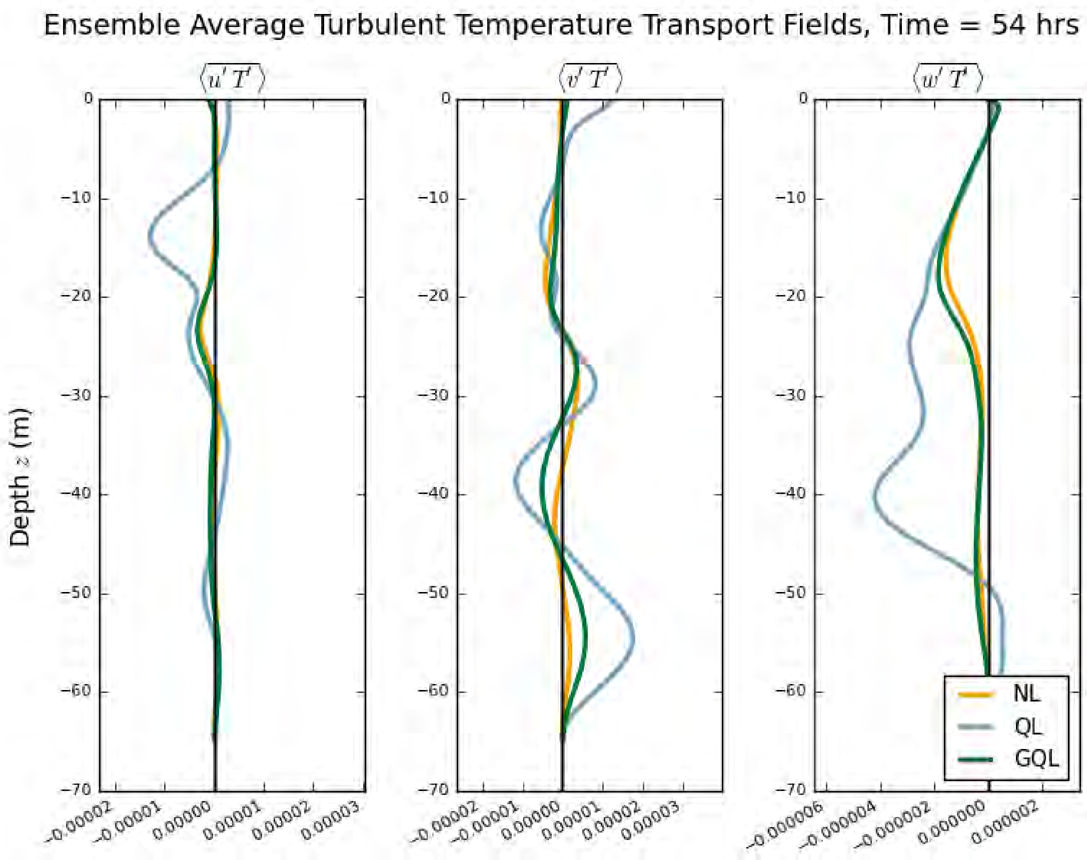


Figure 3.6: Vertical profiles of ensemble averaged turbulent temperature transport for NL, QL, and GQL simulations.

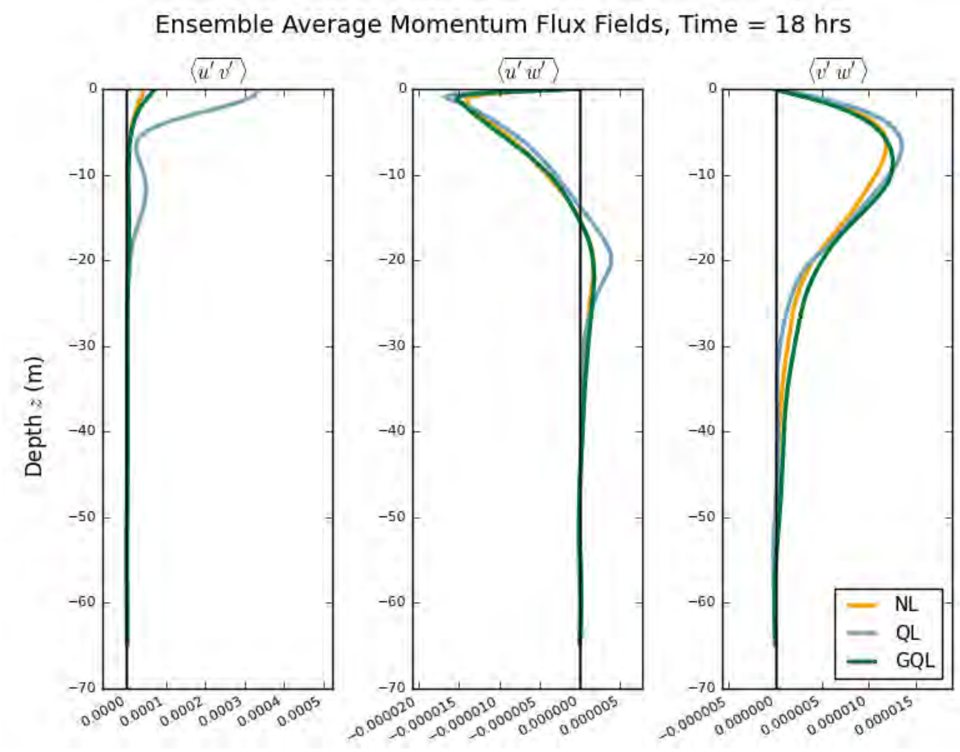
may not be enough to collect reliable statistics. The possibility of poor convergence and its implications for DSS will be explored in section 4.3.

Comparing strengths, we see that $\langle\overline{w'T'}\rangle$ is an order of magnitude less than either $\langle\overline{u'T'}\rangle$ or $\langle\overline{v'T'}\rangle$. Evidently, w' and T' are less correlated, which indicates that warmer water does not have a strong preference for moving up as opposed to down. Physically, this tells us that buoyancy, introduced in the equations of motion as the term $g\alpha(T - T_0)$ (see equation 1.2b), is weak compared to the other forces at work in Langmuir turbulence.

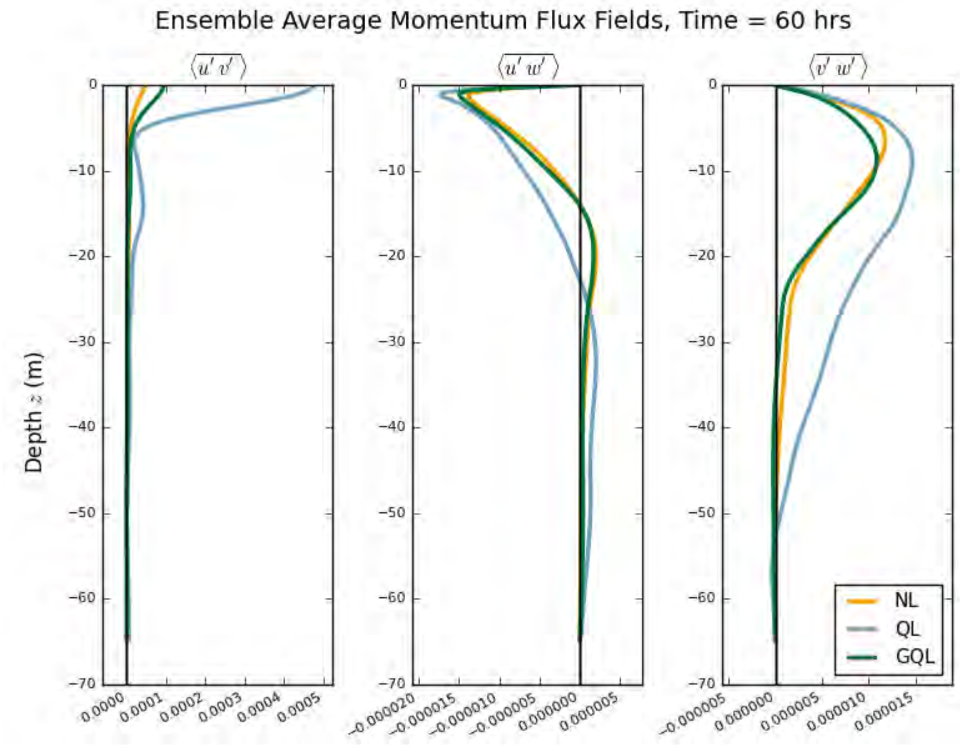
3.3.2 Momentum Flux

There are three unique momentum fluxes determined by the product $u'_i u'_j$ where $i \neq j$: $\langle\overline{u'v'}\rangle$, $\langle\overline{u'w'}\rangle$, and $\langle\overline{v'w'}\rangle$. The three momentum fluxes for the NL, QL, and GQL cases are displayed at 18 hours in figure 3.7a and at 60 hours in figure 3.7b.

Momentum flux is an interesting case where the profiles for NL, QL, and GQL tend to be fairly stable over time, with the main exception being the QL $\langle\overline{v'w'}\rangle$ profile, which develops a significant deep-water tail by 60 hours. Stability does not, however, imply goodness of fit. The GQL profiles show deviations from the NL case near the surface in $\langle\overline{u'v'}\rangle$ and in the length of the deep-water tail in $\langle\overline{v'w'}\rangle$. For $\langle\overline{u'v'}\rangle$, the NL and GQL profiles are fairly consistent, although near the surface the QL is many times the value of either of the others. The agreement between the three on $\langle\overline{u'w'}\rangle$ is the best of the bunch, although even there the QL profile deviates further from the NL and GQL as time passes (see figure 3.7b).



(a) 18 hours



(b) 60 hours

Figure 3.7: Vertical profiles of ensemble averaged momentum flux for NL, QL, and GQL simulations.

Chapter 4

Discussion

It is easy to get lost in the minutiae of figures and approximations, so let us take a step back and review some of the takeaways so far:

1. In most cases, GQL mean fields and transports are a better fit to their NL counterparts than the QL results.

2. The mixed layer actually gets shallower over the course of the simulation, although the deep ocean also becomes less stratified. Given enough time, the two regions may converge to form a deeper mixed layer.

3. While temperature profiles were remarkably similar across all three simulations, the QL profile was slightly cooler near the surface and warmer near the bottom of the domain, which could indicate enhanced vertical mixing.

4. Vertical profiles of both momentum and temperature transport for QL and (to a lesser extent) GQL occasionally fluctuated unexpectedly from snapshot to snapshot, indicating possible issues with convergence of the ensemble members.

Item 1 is the good news: as we expected, allowing for more eddy-eddy scattering generally improves computational representations of turbulence. This is a point in the favor of DSS, particularly given the issues encountered with the QL approximation when trying to model processes without sufficiently strong mean fields [10] (and turbulence is a poster child for things not moving in a uniform direction).

Items 2-4 are more mixed. The mixed layer does not appear to deepen monotonically, although this may be attributable to the type of deep mixing driven by the downwelling jets formed in between Langmuir cells. The QL vertical temperature profile in figure 3.3 seemed to reduce the gradient between the mixed layer and the deep ocean a bit faster than the NL or GQL cases, but it is difficult to say with any certainty what may have contributed to it. Finally, fluctuations in some vertical profiles hinted at an ensemble convergence issue, but at least that can be (and is, in section 4.3) probed more closely. In the sections that follow, we will explore what the figures in the previous section can and cannot tell us about the phenomenon above: the evolution of the mixed layer (section 4.1), the smoothing of the QL temperature profile (section 4.2), and convergence testing of QL and GQL (section 4.3).

4.1 Mixed Layer Evolution

Deepening of the mixed layer is one of the most important features to capture in Langmuir turbulence climate process models. As explained in section 1.3, the mixed layer serves as the dynamic interface between the volatile atmosphere and the stable deep ocean. One of the primary purposes of integrating Langmuir turbulence into global climate models is to improve the bias that currently exists, particularly in the Southern Ocean, on the mixed layer depth (see figures 1.7), which in turn affects the quantity of heat and trace gases stored in the ocean. So we had better make sure that QL and GQL are getting it right.

The temperature profiles in figures 3.3 and 3.4 indicate that the initial mixed layer tends to get cooler and shallower over time while the stratification of the deep ocean lessens. If we imagine the downwelling jets of Langmuir cells to be pushing down on the mixed layer, then this result is not reassuring. However, recall that the downwelling jets can penetrate 100 m or more, while our simulation domain is only about 65 m deep. Instead of pushing down, Langmuir turbulence may be injecting warmer water from the surface into the deep ocean. This would gradually warm the deep ocean rather than merely pushing it deeper. But how could we determine that this is actually happening? In theory, our turbulent temperature transports should tell us how temperature is being shifted around in the domain; after all, $\langle\overline{w'T'}\rangle$ measures the transport of T in the vertical direction. However, those plots (figure 3.6) are so variable over time as to be difficult to trust. One small point in the favor of my speculation is that we do see non-zero values of $\langle\overline{w'T'}\rangle$ all the way down to the bottom of the domain. This holds true even for the very first snapshot at 6 hours, so there is vertical temperature transport well into the deep ocean region beginning early on.

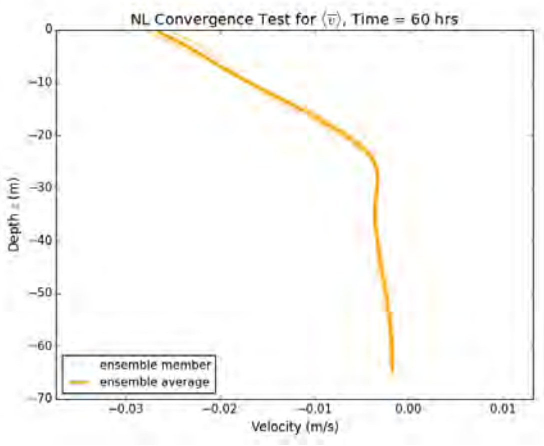
4.2 QL Temperature Smoothing

While the QL and GQL temperature profiles both track NL quite closely throughout the simulation, by 60 hours the QL profile exhibits some extra cooling near the surface and warming near the bottom of the domain (see figure 3.3). Again, I can only speculate what might cause such a trend. It is possible that the lack of eddy-eddy scattering prevents coherent dynamical structures from breaking down, since their small and large turbulence scales do not interact in the quasilinear approximation. Certainly the horizontal midplanes in figure 3.1 show that Langmuir cells in the QL case are much more regular than those for either NL or GQL. The QL vertical turbulent temperature transport $\langle\overline{w'T'}\rangle$ is highly variable from one snapshot to the next, but that variability sometimes produces much higher transport magnitudes than appear for either NL or GQL, even near the base of the domain. It is possible that QL simulations tend to build up strong but transient structures that lead to greater mixing due to their neglect of eddy-eddy scattering. Unfortunately, performing the energy pathway analysis that might support such a conjecture is beyond the scope of this project.

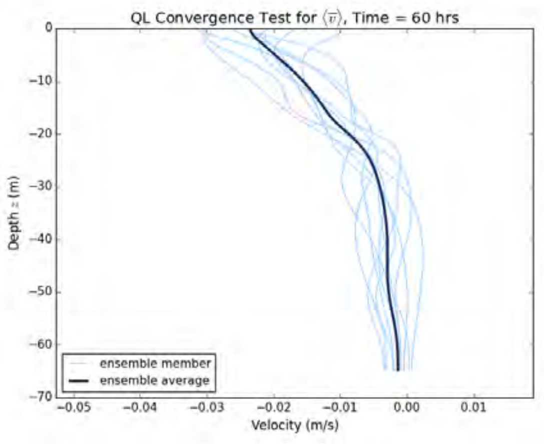
4.3 Convergence Analysis

Over the course of evaluating some of the mean fields, momentum flux profiles, and turbulent temperature transports, I noticed that the QL profiles and even occasionally the GQL could fluctuate dramatically as a function of depth or from one snapshot to the next. A particularly striking example is in the turbulent temperature transports $\langle\overline{v'T'}\rangle$ and $\langle\overline{w'T'}\rangle$ in figure 3.6, where QL waves about erratically. I speculated that the variations could arise from poor convergence among the ensemble members; if each ensemble member was drastically different from the others and tended to vary widely over time, we might expect the ensemble average to also swing about. In that vein, I produced plots that show the ensemble average on top of all 10 individual ensemble member profiles; representative snapshots at 60 hours for $\langle\overline{v}\rangle$ (figure 4.1) and $\langle\overline{v'w'}\rangle$ (figure 4.2).

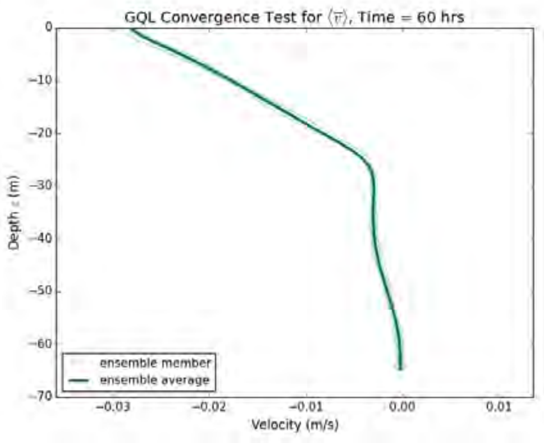
It is both gratifying and disheartening that we see poor convergence reflected in the mixed ensemble and individual plots. The mean fields tended to perform better, with the ensemble members of both NL and GQL closely packed around the ensemble average but much more variation in QL. The ensemble averaged momentum flux was reassuringly stable and well-defined for NL but more variable for GQL and



(a) NL

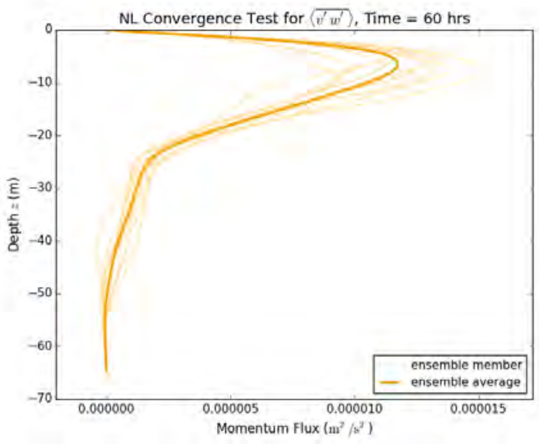


(b) QL

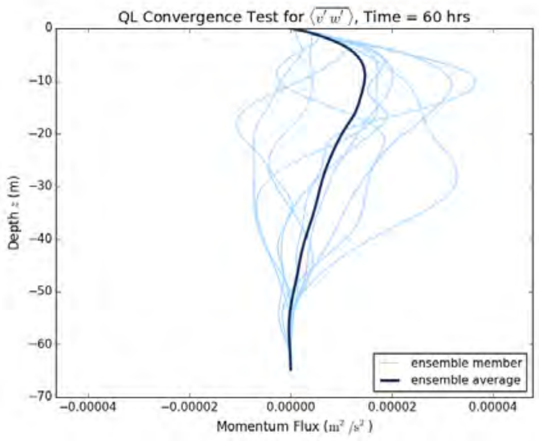


(c) GQL

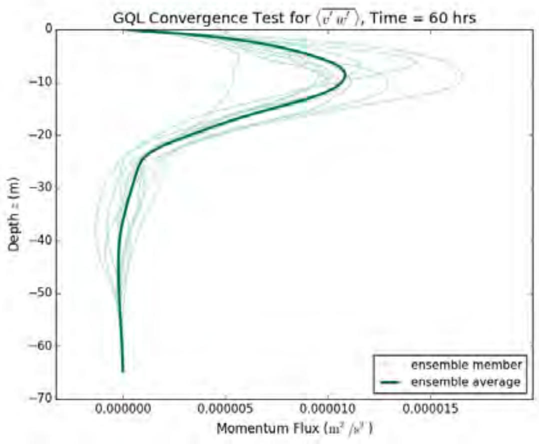
Figure 4.1: Horizontal velocity mean field $\overline{v}$ ensemble averages (dark) and ensemble members (light) for NL, QL, and GQL.



(a) NL



(b) QL



(c) GQL

Figure 4.2: Momentum flux $\overline{v'w'}$ ensemble averages (dark) and ensemble members (light) for NL, QL, and GQL.

particularly poor for QL. It is not uncommon to see some ensemble members far into the positive and others far into the negative for the same snapshot, tugging the ensemble average about. One conclusion we might draw is that we simply need many more ensemble members; instead of averaging over 10 simulations, we might try 20 or even 50 and hope that collecting more statistics will make it clearer where the ensemble average truly lies. A different interpretation is that QL and GQL are more sensitive to initial conditions when it comes to certain types of transport. If the structures of QL are indeed more robust because of the neglect of eddy-eddy scattering, for example, the initial conditions might have an outsized influence on which structures arise and persist. Without the time and resources available to try a 50-member ensemble simulation, I am left to speculation.

4.3.1 Future Work

As exciting as it is to show that Direct Statistical Simulation can indeed reproduce complicated turbulent features, there is still work to be done before it becomes usable in sub-grid climate models. The next major thrust in research on Direct Statistical Simulation is dimensional reduction. Recall that the Reynolds decomposition $q = \overline{q} + q'$ used in the quasilinear approximation applies to a generalized average $\overline{q}$, not just the horizontal average that was used in this work. Depending on the symmetry of the problem in question and the averaging operation $\overline{q}$ used, DSS can actually end up trying to solve over *more* dimensions than DNS's four (three space, one time), making it much more computationally expensive than DNS methods [20]. There is evidence that the argument for DSS can be salvaged by using Proper Orthogonal Decomposition to discard the least energetic modes, reducing the dimensionality of the problem and bringing DSS back into competition with DNS [8]. Future research must further explore the possibilities for reduced dimensionality, including problems for which it succeeds or fails and best approaches to choosing how many modes to include.

An area of future research more closely related to this project in particular is testing grid resolution and solution convergence. In comparing these midplanes of Langmuir turbulence, shown in section 3, to those produced in the preliminary stages of Ref [8], we noticed that the length scale of the Langmuir cells between the two projects was inconsistent, with mine approximately 50 m across and theirs more like 100 m, despite identical equations and physical parameters. We discovered that the grid resolution near the top of the domain (ocean surface) seemed to be setting the length scale of the turbulent features; the Chebyshev basis has higher resolution near the top than the bottom while the computational method in Ref [8] has the same grid resolution throughout the domain. This finding implies that one or both of our simulations was not fully converged. In order to ensure that the grid resolution is not having an undue impact on the physical properties of our simulated turbulence, it would be helpful to run a series of simulations with increasingly higher resolution. We could then track the size of the Langmuir cells as a function of grid resolution to determine the optimal resolution, high enough to fully resolve the Langmuir cells and get a converged solution but low enough to be computationally efficient.

In the interests of studying the evolution of the mixed layer, another area of future study would involve working with deeper domains. As mentioned in the analysis of temperature profiles in section 3.2.1 and their implications for the mixed layer in section 4.1, our domain is shallow enough that the downwelling jets produced by Langmuir turbulence appear to hit the bottom and start warming the "deep ocean" early on in the simulation. For a more realistic look at the mixed layer, we would need to work with a deeper domain so that the bottom surface is, at least for the first few hours, unaffected by Langmuir turbulence. In that scenario, we might be able to observe the formation of the deeper mixed layer as the surface cools and the region just below the initial mixed layer depth warms, a trend observed in figure 3.4. We could then compare the rate of deepening and, hopefully, the final equilibrium depth of the mixed layer between the NL, QL, and GQL simulations for further information about how well DSS captures the key climate features of Langmuir turbulence.

Chapter 5

Conclusions

Let us take a step back and review what we have covered so far before attempting a sweeping conclusion. We began by going over the current state of climate modeling: the use of Direct Numerical Simulation and earth system models, their limitations in spatial resolution and computing speed, and the need for Direct Statistical Simulation, which can begin to incorporate sub-grid phenomenon. Then we focused in on Langmuir turbulence, a sub-grid turbulence feature in the upper ocean with a particularly strong influence on the mixed layer depth and related climate variables like sea surface temperature. We dove into Direct Statistical Simulation and the quasilinear and generalized quasilinear approximations to explore methods for simplifying the equations of motion by decomposing variables into mean field and fluctuating components and discarding eddy-eddy scattering. For the experimental setup, I described the dimensions and physical properties of the simulation domain, paying particular attention to the importance of heat flux and wind stress in the boundary conditions and the inclusion of both a mixed layer and a stratified deep ocean in the initial temperature profile. We then reviewed the results, finding evidence for Langmuir cells in horizontal midplanes of vertical velocity, reduction of the temperature gradient over time, and good agreement between NL and GQL cases for mean field and correlation profiles. Finally, I presented key takeaways in the Discussion and evaluated the evolution of the mixed layer, the potential for greater vertical mixing in the QL case, and the issue of ensemble convergence.

Now for the sweeping conclusion. My project presents good evidence that both the QL and GQL approximations can produce Langmuir turbulence, but that GQL does so with much greater accuracy. This is good news for the development of Direct Statistical Simulation, as I have shown that even complex turbulence features like Langmuir turbulence are quite robust to the loss of eddy-eddy scattering. It is more difficult to evaluate the long-term performance of any of the three cases in representing the mixed layer because of the vertical size constraints that were placed on the domain. Given a deeper domain and a longer run time, it would be possible to determine more precisely how the evolution of the mixed layer varies between the three cases. In particular, we would be interested in learning whether the equilibrium state, when the mixed layer becomes stable over time, occurs at the same time or the same depth between the three cases. It is promising for the future of the Southern Ocean shallow bias that all three cases trend towards the development of a deeper mixed layer over time, but for application in climate models the equilibrium depth will be just as important as the fact that the mixed layer is indeed getting deeper.

Acknowledgements

There are a lot of thanks to go around to all of the people who made this thesis possible. Where else could I start but with my research advisor, Prof. Brad Marston, whose knowledge and patience have been invaluable to me this past year. He always seemed to know when I needed to hear an encouraging word and when I needed a prod to try harder. Through this project he also introduced me to climate science, a field that has since captured my imagination in a way that nothing else in science has.

I never would have understood the first thing about my project without the advice and assistance of Joseph Skitka, who shared his notes and expertise with me and was always willing to help me troubleshoot when I was just about ready to throw in the towel and start keyboard mashing. I would also like to thank Han Liang, a Marston alumn, who lent me his code as a learning tool in the formative stages of this project.

Science would be a small and lonely pursuit without collaborations, and luckily my work has been neither small nor lonely. From the Department of Earth, Environmental, and Planetary Sciences I would like to thank Prof. Baylor Fox-Kemper for inviting me to his group meetings and answering every email I ever sent begging for insight into oceanography. His PhD student Qing Li has also been incredibly helpful, offering me a copy of his thesis to shamelessly reference in the (many) areas of climate and fluid dynamics where my own knowledge falls woefully short. A very special thanks goes to Prof. Jeff Oishi of Bates College, one of the developers of Dedalus, who thought he was coming to Brown to give a talk but quite cheerfully agreed to sit for three hours in a windowless office and pour over code with me. Prof. Oishi is also the one who finally caught and fixed The Bug that plagued my simulations for six straight months from July through January, and for that he has my eternal gratitude.

More broadly, I want to thank the Physics Department at Brown for igniting and supporting my interest in physics; the UTRA program for providing the funding that made it possible for me to start this research over the summer when I had time as opposed to in the fall when, as it turns out, I most assuredly did not; and all of the friends, family, mentors, and teachers who have given me confidence and encouragement along the way.

Bibliography

[1] Greg Shirah. Perpetual ocean. Web, August 2011. Published on NASA's Scientific Visualization Studio website.
`https://svs.gsfc.nasa.gov/3827`.

[2] Leo H. Holthuijsen. *Waves in Oceanic and Coastal Waters*. Cambridge University Press, New York, 2007.

[3] S. A. Thorpe. Langmuir circulation. *Annual Review of Fluid Mechanics*, 36:55–79, 2004.

[4] NPR Staff. Gulf oil spill could eclipse exxon valdez disaster. Web, April 2010. News article published by npr.org.
`https://www.npr.org/templates/story/story.php?storyId=126373753`.

[5] Jeff A. Polton and Stephen E. Belcher. Langmuir turbulence and deeply penetrating jets in an unstratified mixed layer. *Journal of Geophysical Research*, 112(C9), 2007.

[6] Stephen E. Belcher, Alan L. M. Grant, Kirsty E. Hanley, Baylor FoxKemper, Luke Van Roekel, Peter P. Sullivan, William G. Large, Andy Brown, Adrian Hines, Daley Calvert, Anna Rutgersson, Heidi Pettersson, JeanRaymond Bidlot, Peter A. E. M. Janssen, and Jeff A. Polton. A global perspective on langmuir turbulence in the ocean surface boundary layer. *Geophysical Research Letters*, 39(18), 2012.

[7] E. A. D'Asaro, J. Thomson, A. Y. Shcherbina, R. R. Harcourt, M. F. Cronin, M. A. Hemer, and B. FoxKemper. Quantifying upper ocean turbulence driven by surface waves. *Geophysical Research Letters*, 41(1):102–107, 2014.

[8] Joseph Skitka, J. B. Marston, and Baylor Fox-Kemper. Reduced-order quasilinear ocean boundary-layer turbulence modeling. (unpublished), April 2018.

[9] J. B. Marston, E. Conover, and Tapio Schneider. Statistics of an unstable barotropic jet from a cumulant expansion. *Journal of Atmospheric Science*, 65(1955), 2008.

[10] J. B. Marston, G. P. Chini, and S. M. Tobias. The generalized quasilinear approximation: Application to zonal jets. *Physical Review Letters*, 116(21), 2016.

[11] Han Liang. Generalized quasi-linear approximation of three-dimensional convective boundary layer. Master's thesis, Brown University, December 2016. Received from author in PDF form.

[12] Qing Li. *Langmuir turbulence and its effects on global climate*. PhD thesis, Brown University, May 2018. Received from author in PDF form.

[13] S. A. Thorpe. *An Introduction to Ocean Turbulence*. Cambridge University Press, New York, 2007.

[14] M. A. C. Teixeira and S. E. Belcher. On the distortion of turbulence by a progressive surface wave. *Journal of Fluid Mechanics*, 458:229–267, 2002.

[15] James C. McWilliams, Peter Sullivan, and Chin-Hoh Moeng. Langmuir turbulence in the ocean. *Journal of Fluid Mechanics*, 334:1–30, 1997.

[16] L. Cavaleri, B. Fox-Kemper, and M. Hemer. Wind waves in the coupled climate system. *Bulletin of the American Meteorological Society*, 93(11), November 2012.

[17] A. D. D. Craik and S. Leibovich. A rational model for langmuir circulations. *Journal of Fluid Mechanics*, 73(3):401–426, February 1976.

[18] Nobuhiro Suzuki and Baylor Fox-Kemper. Understanding stokes forces in the waveaveraged equations. *Journal of Geophysical Research*, 121(5):3579–3596, 2016.

[19] J. M. McDonough. Lectures in elementary fluid dynamics: Physics, mathematics and applications. Online PDF, 2009. Originated from the Departments of Mechanical Engineering and Mathematics at the University of Kentucky. `http://web.engr.uky.edu/~acfd/me330-lctrs.pdf`.

[20] Altan Allawala, S. M. Tobias, and J. B. Marston. Dimensional reduction of direct statistical simulation. Published on arXiv: 1708.07805. `https://arxiv.org/abs/1708.07805`, February 2018.

[21] S. M. Tobias, K. Dagon, and J. B. Marston. Astrophysical fluid dynamics via direct statistical simulation. *The Astrophysical Journal*, 727(2), January 2011.

[22] Vaughan L. Thomas, Brian F. Farrell, Petros J. Ioannou, and Dennice F. Gayme. A minimal model of self-sustaining turbulence. *Physics of Fluids*, 27(10), September 2015.

[23] N. C. Constantinou, A. Lozano-Durán, M.-A. Nikolaidis, B. F. Farrell, P. J. Ioannou, and J. Jiménez. Turbulence in the highly restricted dynamics of a closure at second order: comparison with dns. *Journal of Physics: Conference Series*, 506(012004), 2014.

[24] Farid Ait-Chaalal, Tapio Schneider, Bettina Meyer, and J B Marston. Cumulant expansions for atmospheric flows. *New Journal of Physics*, 18, February 2016.

[25] J. Squire and A. Bhattacharjee. Statistical simulation of the magnetorotational dynamo. *Physical Review Letters*, 114(8), February 2015.

[26] S. M. Tobias and J. B. Marston. Direct statistical simulation of out-of-equilibrium jets. *Physical Review Letters*, 110(10), March 2013.

[27] B. F. Farrell and P. J. Ioannou. Statistical state dynamics-based analysis of the physical mechanisms sustaining and regulating turbulence in couette flow. *Physical Review Fluids*, 2(8), August 2017.

[28] Keaton J. Burns, Geoffrey M. Vasil, Jeffrey S. Oishi, Daniel Lecoanet, and Benjamin Brown. Dedalus: Flexible framework for spectrally solving differential equations. Astrophysical Source Code Library. At `http://ascl.net/1603.015` (code) and `http://dedalus-project.org` (homepage).

[29] Sir Edward Bullard. Physical properties of sea water. Web. Based on Kaye and Laby's book Tables of Physical and Chemical Constants and Some Mathematical Functions, 16th edition (1995). Website hosted by the National Physical Laboratory.
`http://www.kayelaby.npl.co.uk/general_physics/2_7/2_7_9.html`.

Appendix A

Nonlinear Langmuir Turbulence Code

```
# Load Python packages, Dedalus
import numpy as np
from mpi4py import MPI
import time

from dedalus import public as de
from dedalus.extras import flow_tools

# Import Dedalus logger
  # the logger keeps track of and periodically saves the progress
    of the simulation
  # all logger entries are given a date and time stamp as well as
    the ID of the CPU process which is printing the message
    # for CPU process example, 0/4 means the logger message is
   printed from the first (0) of four (4) processes
  # to print a message to the logger: >>logger.info('String to
   print to the logger goes here')
import logging
logger = logging.getLogger(__name__)

# Begin timing initialization
start_init_time = time.time()

# Domain parameters
Lx, Ly, Lz = (288., 288., 64.8)  # Physical dimensions of the
   domain in meters
x_modes, y_modes, z_modes = (96, 96, 72)  # Number of modes (
   degrees of freedom) along each direction in the domain
  # simulation will run slightly faster if the number of modes in
    the periodic directions is a power of 2
```

```
# Create bases
  # x and y directions have Fourier (periodic) basis
  # z direction has a Chebyshev (non-periodic) basis
  # interval=(-Lx/2, Lx/2) means x=0 (for example, in Analysis)
    corresponds to the center of the domain
  # dealias controls resolution when converting from spectral
    space (where real data is stored) and real space (where we
    usually want to look at the data)
     # when converting from spectral space to real space, you end
    up with a grid of 3/2 * (number of modes) in each direction
x_basis = de.Fourier('x', x_modes, interval=(-Lx/2, Lx/2),
    dealias=3/2)
y_basis = de.Fourier('y', y_modes, interval=(-Ly/2, Ly/2),
    dealias=3/2)
z_basis = de.Chebyshev('z', z_modes, interval=(-Lz, 0), dealias
    =3/2)

# Construct domain
  # mesh=[n,m] is for running in parallel, divides domain among n
     processors in one periodic (x or y, in this case) direction
    and m processors in the other
     # number of processors must equal n*m
domain = de.Domain([x_basis, y_basis, z_basis], grid_dtype=np.
    float64, mesh=[4,8])

# Set up an initial value problem (IVP)
  # p = pressure, T = temperature, (u,v,w) = velocity field
  # Tz, uz, etc. are the z-derivatives of corresponding variables
     (explained in "Equations" section, below)
problem = de.IVP(domain, variables=['p','T','u','v','w','Tz','uz'
    ,'vz','wz'], time='t')

# Metadata (optional)
  # Dirichlet preconditioning (for variables with Dirichlet
    boundary conditions; see "Boundary conditions" below) will
    speed up computation
problem.meta['p','w','uz','vz','Tz']['z']['dirichlet'] = True

# Define constant parameters for the equations of motion
  # adapted from McWilliams et al. (1997)
  # SI units given in brackets at end of each comment
  # alpha, T_0, and g are also given local variables for
    initializing temperature (in "initialize temperature", below)
problem.parameters['q'] = 5. # heat flux from upper surface [W/m
    ^2]
problem.parameters['c_p'] = 3994 # specific heat of sea water at
    constant pressure [J/kg*C]
```

```
problem.parameters['rho_0'] = 1035 # background density [kg/m^3]
problem.parameters['nu_z'] = 1e-3 # vertical harmonic kinematic viscosity [m^2/s]
problem.parameters['nu'] = 2e-3 # horizontal harmonic kinematic viscosity [m^2/s]
problem.parameters['kappa_z'] = 1e-3 # vertical coefficient of thermal diffusivity [m^2/s]
problem.parameters['kappa'] = 2e-3 # horizontal coefficient of thermal diffusivity [m^2/s]
problem.parameters['alpha'] = alpha = 2e-4 # coefficient of thermal expansion [1/C]
problem.parameters['T_0'] = T_0 = 20 # background temperature [C]
problem.parameters['g'] = g = 9.81 # gravitational acceleration [m/s^2]
problem.parameters['f'] = 1e-4 # Coriolis parameter [1/s]
problem.parameters['tau'] = 0.037 # surface wind stress [N/m^2]

# Make domain dimensions into problem parameters
  # for calculating integrals in "Mean Fields for Analysis" and "Analysis", below
problem.parameters['Lx'] = Lx
problem.parameters['Ly'] = Ly
problem.parameters['Lz'] = Lz

# Define Stokes field
Stokes_field = domain.new_field(name='u_s')
z = domain.grid(2)  # real space grid points for z-axis (2 is the index for the z-axis)
k_s = 0.105  # Stokes wavenumber [1/m]
a_s = 0.068  # Stokes velocity at surface (z=0) [m/s]
Stokes_field['g'] = a_s * np.exp(2. * k_s * z)  # define Stokes field in real grid ('g') space
Stokes_field.meta['x', 'y']['constant'] = True  # metadata tells Dedalus that Stokes field does not couple periodic (x, y) directions; Dedalus does not support fields that do not meet this criteria
problem.parameters['u_s'] = Stokes_field  # set Stokes field as a named parameter

# Mean Fields for Analysis
  # for use in "Analysis: mean fields and correlations" below
problem.substitutions['u_bar'] = "integ(u, 'x', 'y')/(Lx*Ly)"
problem.substitutions['v_bar'] = "integ(v, 'x', 'y')/(Lx*Ly)"
problem.substitutions['w_bar'] = "integ(w, 'x', 'y')/(Lx*Ly)"
problem.substitutions['T_bar'] = "integ(T, 'x', 'y')/(Lx*Ly)"

# Equations
```

```
  # Phase-averaged Craik-Leibovich equations from Craik (1977)
  # Tz, uz, etc are used because Dedalus will only allow first-order derivatives in the inhomogeneous (Chebyshev) direction
  # All non-linear terms and terms that vary in only one direction (like f*u_s) must be on right-hand side (explicitly time-stepped)

  # divergence condition:
problem.add_equation('dx(u) + dy(v) + wz = 0')
  # thermal energy:
problem.add_equation('dt(T) - kappa*(dx(dx(T)) + dy(dy(T))) - kappa_z*dz(Tz) = -u*dx(T) - u_s*dx(T) - v*dy(T) - w*Tz')
  # Navier-Stokes/momentum equations:
problem.add_equation('dt(u) + (1/rho_0)*dx(p) - nu*(dx(dx(u)) + dy(dy(u))) - nu_z*dz(uz) - f*v + dx(u*u_s)      = -u*dx(u) - v*dy(u) - w*uz')
problem.add_equation('dt(v) + (1/rho_0)*dy(p) - nu*(dx(dx(v)) + dy(dy(v))) - nu_z*dz(vz) + f*u + dy(u*u_s) - u_s*dy(u) + u_s*dx(v) = -u*dx(v) - v*dy(v) - w*vz - f*u_s')
problem.add_equation('dt(w) + (1/rho_0)*dz(p) - nu*(dx(dx(w)) + dy(dy(w))) - nu_z*dz(wz)        + dz(u*u_s) - u_s*uz    + u_s*dx(w) = -u*dx(w) - v*dy(w) - w*wz - 0.5*dz(u_s*u_s) + g*alpha*(T-T_0)')
  # match z-derivatives to Tz, uz, etc:
problem.add_equation("Tz - dz(T) = 0")
problem.add_equation("uz - dz(u) = 0")
problem.add_equation("vz - dz(v) = 0")
problem.add_equation("wz - dz(w) = 0")

# Boundary conditions
  # apply to the bounds of the non-periodic (Chebyshev) basis direction

  # left z (bottom of domain)
problem.add_bc("left(Tz) = 0")  # no heat exchange in stratified deep ocean (bottom of domain)
problem.add_bc("left(uz) = 0")  # free surface condition
problem.add_bc("left(vz) = 0")  # free surface condition
problem.add_bc("left(w) = 0", condition="(nx != 0) or (ny != 0)")    # "condition" prevents system from being over-constrained; this boundary condition is replaced with the pressure gauge choice, below

  # right z (ocean surface)
problem.add_bc("right(Tz) = -q/(kappa*c_p*rho_0)")  # heat flux
problem.add_bc("right(uz) = tau/(nu*rho_0)")  # surface wind stress
problem.add_bc("right(vz) = 0")  # free surface condition
problem.add_bc("right(w) = 0")  # no vertical transport out of
    domain

  # pressure gauge choice
problem.add_bc("integ_z(p) = 0", condition="(nx == 0) and (ny ==
    0)")

# Build solver
  # timestepper is 3rd-order 4-stage DI (diagonally implicit) + E
     (explicit) Runge-Kutta scheme
solver = problem.build_solver(de.timesteppers.RK443)
logger.info('Solver built')

# Initial conditions for temperature
T = solver.state['T']
Tz = solver.state['Tz']
T.set_scales(domain.dealias, keep_data=False) # initialize T on
    dealiased domain (higher resolution)
Tz.set_scales(domain.dealias, keep_data=False)  # initialize T on
     dealiased domain (higher resolution)

# Define filter field function
  # Dedalus doesn't like random Gaussian noise of the type we
    introduce for the initial temperature (not very compatible
    with spectral space representation)
  # this filter will throw out high-frequency noise
def filter_field(field,frac=0.5):
  field.require_coeff_space()
  dom = field.domain
  local_slice = dom.dist.coeff_layout.slices(scales=dom.dealias)
  coeff = []
  for n in dom.global_coeff_shape:
    coeff.append(np.linspace(0,1,n,endpoint=False))
  cc = np.meshgrid(*coeff,indexing='ij')
  field_filter = np.zeros(dom.local_coeff_shape,dtype='bool')
  for i in range(dom.dim):
    field_filter = field_filter | (cc[i][local_slice] > frac)
  field['c'][field_filter] = 0j

# Set up variables for initial temperature
N_2 = 1.936e-5  # Brunt-Vaisala frequency [1/s^2]
MLD = -32.1  # mixed-layer depth [m]
filter_frac = 0.5  # set fraction of frequencies to keep in
    filter_field

# Domain shape and slices
```

```
  # when run in parallel, Dedalus divides domain along one of the periodic directions and allocates a slice to each processor
  # for setting initial temperature, need "slices" to tell us what part of the full domain is being run on that processor
gshape = domain.dist.grid_layout.global_shape(scales=domain.dealias)  # [x, y, z] dimensions of full domain
slices = domain.dist.grid_layout.slices(scales=domain.dealias)  # takes domain slice of current process (for running in parallel)

# Initialize random noise globally
rand = np.random.RandomState(seed=60)
noise = rand.standard_normal(gshape)[slices]

# Initialize temperature
  # in top 4.5 m, background temperature of T_0 + random noise of magnitude 1e-4
  # from surface to mixed-layer depth (MLD), background temperature of T_0
  # below MLD (deep ocean), linear stratification from Brunt-Vaisala frequency
temp_array = np.zeros(gshape)[slices]
z_vals = domain.grid(2, scales=domain.dealias)[0,0,:]   # real space grid points along z-axis
  # loop over depths:
for ind,depth in enumerate(z_vals):
  # array index "ind" and z-value "depth" of each domain slice along z-axis
  current_shape = temp_array[:,:,ind].shape
  if depth > -4.5:  # surface with noise
    rand_noise = noise[:,:,ind]
    temp_array[:,:,ind] = T_0 * np.ones(current_shape) + (1e-4 * rand_noise)
  elif depth > MLD:  # within mixed layer
    temp_array[:,:,ind] = T_0 * np.ones(current_shape)
  elif depth <= MLD:  # stratified deep ocean
    dist_from_MLD = abs(MLD-depth)
    T_BV = - (N_2*dist_from_MLD)/(alpha*g)  # temperature contribution from Brunt-Vaisala frequency
    temp_array[:,:,ind] = (T_0 + T_BV) * np.ones(current_shape)
T['g'] = 1.0 * temp_array  # set initial temperature as defined in "temp_array" above
logger.info('Beginning filter')
filter_field(T,frac=filter_frac)  # apply filter to remove high-frequency components
logger.info('Finished filter')
T.differentiate('z', out=Tz)  # set initial dz(T)
```

```

# Integration parameters
solver.stop_sim_time = 78.*60.*60.  # stops simulation after running for 72 hours of simulated time
solver.stop_wall_time = np.inf  # no constraint on walltime
solver.stop_iteration = np.inf  # no constraint on number of iterations

# Analysis: full state variables
  # set up "state_variables" folder for HDF5 file to write to
  # sim_dt sets frequency of writing tasks to the file
    # sim_dt=21600 -> write out all snapshots1 tasks every 21600 seconds (6 hours) of simulated time
  # max_writes limits total number of writes per HDF5 file to keep file size reasonable
snapshots1 = solver.evaluator.add_file_handler('state_variables', sim_dt=21600., max_writes=10)
  # save full state variable in grid space ('g') for T, u, v, and w
snapshots1.add_task(solver.state['T'], layout='g', name='T')
snapshots1.add_task(solver.state['u'], layout='g', name='u')
snapshots1.add_task(solver.state['v'], layout='g', name='v')
snapshots1.add_task(solver.state['w'], layout='g', name='w')

# Analysis: mean fields and correlations
snapshots2 = solver.evaluator.add_file_handler('diagnostics', sim_dt=21600., max_writes=10)
  # save mean fields (defined in "Mean Fields for Analysis", above)
snapshots2.add_task('u_bar', layout='g', scales=1, name='u_bar')
snapshots2.add_task('v_bar', layout='g', scales=1, name='v_bar')
snapshots2.add_task('w_bar', layout='g', scales=1, name='w_bar')
snapshots2.add_task('T_bar', layout='g', scales=1, name='T_bar')
  # save correlations: both momentum flux (velocity*velocity) and turbulent transport (velocity*temperature)
snapshots2.add_task("integ(u*v, 'x', 'y')/(Lx*Ly)", layout='g', scales=1, name='uv_corr')
snapshots2.add_task("integ(u*w, 'x', 'y')/(Lx*Ly)", layout='g', scales=1, name='uw_corr')
snapshots2.add_task("integ(v*w, 'x', 'y')/(Lx*Ly)", layout='g', scales=1, name='vw_corr')
snapshots2.add_task("integ(u*T, 'x', 'y')/(Lx*Ly)", layout='g', scales=1, name='uT_corr')
snapshots2.add_task("integ(v*T, 'x', 'y')/(Lx*Ly)", layout='g', scales=1, name='vT_corr')
snapshots2.add_task("integ(w*T, 'x', 'y')/(Lx*Ly)", layout='g', scales=1, name='wT_corr')
```

```

# Courant-Friedrichs-Lewy condition
  # numerical convergence condition that sets time step based on velocities
  # initial_dt: starting point for CFL calculation for first iteration
  # max_dt: maximum allowed time step
  # safety: multiplier; time step = CFL condition * safety
  # max_change: maximum fractional change between computed time steps
  # cadence: iteration cadence at which to re-compute CFL condition
initial_dt = 1.
CFL = flow_tools.CFL(solver, initial_dt=initial_dt, max_dt=20., safety=1.2, max_change=10., cadence=1)
CFL.add_velocities(('u', 'v', 'w'))  # give state variable velocities
CFL.add_velocity('u_s',0)  # give Stokes drift velocity and axis along which it acts (0 is the index for the x-axis)

# Flow properties
  # allows printing of min/max of flow properties to the logger during the main loop
flow = flow_tools.GlobalFlowProperty(solver, cadence=1)
  # add state variables as flow properties
flow.add_property('T', name='Temperature')
flow.add_property('u', name='u-velocity')
flow.add_property('v', name='v-velocity')
flow.add_property('w', name='w-velocity')

# Elapsed initialization time
end_init_time = time.time()
logger.info('Initialization time: {:f}'.format(end_init_time-start_init_time))

# Main loop
try:
  logger.info('Starting loop')
  start_run_time = time.time()
  while solver.ok:
    dt = CFL.compute_dt()  # calculates time step based on CFL condition
    solver.step(dt)  # this step right here is the money-maker; advances the simulation forward in time by dt seconds
    if (solver.iteration-1) % 20 == 0:  # print status message to logger every 20 iterations
```

```python
      logger.info('Iteration: {}, Time: {:e}, dt: {:e}'.format(solver.iteration, solver.sim_time, dt))
      logger.info('  Max/Min T = {:f}, {:f}'.format(flow.max('Temperature'), flow.min('Temperature')))
      logger.info('  Max/Min U = {:f}, {:f}'.format(flow.max('u-velocity'), flow.min('u-velocity')))
      logger.info('  Max/Min V = {:f}, {:f}'.format(flow.max('v-velocity'), flow.min('v-velocity')))
      logger.info('  Max/Min W = {:f}, {:f}'.format(flow.max('w-velocity'), flow.min('w-velocity')))
except:
  logger.error('Exception raised, triggering end of main loop.')
  raise
finally:  # print summary statistics to logger on completion of simulation
  end_run_time = time.time()
  logger.info('Iterations: {}'.format(solver.iteration))
  logger.info('Sim end time: {:f}'.format(solver.sim_time))
  logger.info('Run time: {:.2f} sec'.format(end_run_time-start_run_time))
  logger.info('Run time: {:f} cpu-hr'.format((end_run_time-start_run_time)/(60.*60.)*domain.dist.comm_cart.size))
```

Appendix B

Quasilinear Langmuir Turbulence Code

```
# Load Python packages, Dedalus
import numpy as np
from mpi4py import MPI
import time

from dedalus import public as de
from dedalus.extras import flow_tools

# Import Dedalus logger
import logging
logger = logging.getLogger(__name__)

# Begin timing initialization
start_init_time = time.time()

# Domain parameters
Lx, Ly, Lz = (288., 288., 64.8)
x_modes, y_modes, z_modes = (96, 96, 72)

# Create bases
x_basis = de.Fourier('x', x_modes, interval=(-Lx/2, Lx/2),
    dealias=3/2)
y_basis = de.Fourier('y', y_modes, interval=(-Ly/2, Ly/2),
    dealias=3/2)
z_basis = de.Chebyshev('z', z_modes, interval=(-Lz, 0), dealias
    =3/2)

# Construct domain
domain = de.Domain([x_basis, y_basis, z_basis], grid_dtype=np.
    float64, mesh=[4,8])
```

```
# Set up an initial value problem (IVP)
  # for quasilinear (QL) approximation, each state variable q is split into two parts (Reynolds decomposition):
    # ql -> low-order or mean field; also written q_bar
    # qh -> high-order or fluctuations; also written q'
    # such that q = ql + qh
problem = de.IVP(domain, variables=['pl','Tl','ul','vl','wl','Tzl','uzl','vzl','wzl','ph','Th','uh','vh','wh','Tzh','uzh','vzh','wzh'], time='t')

# Metadata (optional)
problem.meta['pl','uzl','vzl','wl','Tzl','uzh','vzh','wh','Tzh']['z']['dirichlet'] = True

# Define constant parameters for the equations of motion
problem.parameters['q'] = 5. # heat flux from upper surface [W/m^2]
problem.parameters['c_p'] = 3994 # specific heat of sea water at constant pressure [J/kg*C]
problem.parameters['rho_0'] = 1035 # background density [kg/m^3]
problem.parameters['nu_z'] = 1e-3 # vertical harmonic kinematic viscosity [m^2/s]
problem.parameters['nu'] = 2e-3 # horizontal harmonic kinematic viscosity [m^2/s]
problem.parameters['kappa_z'] = 1e-3 # vertical coefficient of thermal diffusivity [m^2/s]
problem.parameters['kappa'] = 2e-3 # horizontal coefficient of thermal diffusivity [m^2/s]
problem.parameters['alpha'] = alpha = 2e-4 # coefficient of thermal expansion [1/C]
problem.parameters['T_0'] = T_0 = 20 # background temperature [C]
problem.parameters['g'] = g = 9.81 # gravitational acceleration [m/s^2]
problem.parameters['f'] = 1e-4 # Coriolis parameter [1/s]
problem.parameters['tau'] = 0.037 # surface wind stress [N/m^2]

# Make domain dimensions into problem parameters
problem.parameters['Lx'] = Lx
problem.parameters['Ly'] = Ly
problem.parameters['Lz'] = Lz

# Define Stokes field
Stokes_field = domain.new_field(name='u_s')
z = domain.grid(2)
k_s = 0.105 # Stokes wavenumber [1/m]
a_s = 0.068 # Stokes velocity at surface (z=0) [m/s]
Stokes_field['g'] = a_s * np.exp(2. * k_s * z)
```

```
Stokes_field.meta['x', 'y']['constant'] = True
problem.parameters['u_s'] = Stokes_field

# Mean Fields for Analysis
  # recall that, for a state variable q, q = ql + qh
problem.substitutions['u_mean'] = "integ(ul + uh, 'x', 'y')/(Lx*
    Ly)"
problem.substitutions['v_mean'] = "integ(vl + vh, 'x', 'y')/(Lx*
    Ly)"
problem.substitutions['w_mean'] = "integ(wl + wh, 'x', 'y')/(Lx*
    Ly)"
problem.substitutions['T_mean'] = "integ(Tl + Th, 'x', 'y')/(Lx*
    Ly)"

# Equations for nx=0, ny=0 mode
  # nx and ny are Fourier modes in x- and y-direction
  # these equations correspond to dt(ql), also written dt(q_bar)

  # divergence condition:
problem.add_equation('dx(ul) + dy(vl) + wzl = 0', condition='(nx
    == 0) and (ny == 0)')
  # thermal energy:
problem.add_equation('dt(Tl) - kappa*(dx(dx(Tl)) + dy(dy(Tl))) -
    kappa_z*dz(Tzl) = -ul*dx(Tl) - vl*dy(Tl) - wl*Tzl - uh*dx(Th)
    - vh*dy(Th) - wh*Tzh - u_s*dx(Tl)', condition='(nx == 0) and (
    ny == 0)')
  # Navier-Stokes/momentum equations:
problem.add_equation('dt(ul) + (1/rho_0)*dx(pl) - nu*(dx(dx(ul))
    + dy(dy(ul))) - nu_z*dz(uzl) - f*vl + dx(ul*u_s)
               = -ul*dx(ul) - vl*dy(ul) - wl*uzl - uh*dx(uh) - vh*
    dy(uh) - wh*uzh', condition='(nx == 0) and (ny == 0)')
problem.add_equation('dt(vl) + (1/rho_0)*dy(pl) - nu*(dx(dx(vl))
    + dy(dy(vl))) - nu_z*dz(vzl) + f*ul + dy(ul*u_s) - u_s*dy(ul)
    + u_s*dx(vl) = -ul*dx(vl) - vl*dy(vl) - wl*vzl - uh*dx(vh) -
    vh*dy(vh) - wh*vzh - f*u_s', condition='(nx == 0) and (ny ==
    0)')
problem.add_equation('dt(wl) + (1/rho_0)*dz(pl) - nu*(dx(dx(wl))
    + dy(dy(wl))) - nu_z*dz(wzl)       + dz(ul*u_s) - u_s*uzl    +
     u_s*dx(wl) = -ul*dx(wl) - vl*dy(wl) - wl*wzl - uh*dx(wh) - vh
    *dy(wh) - wh*wzh - 0.5*dz(u_s*u_s) + g*alpha*(Tl-T_0)',
    condition='(nx == 0) and (ny == 0)')
  # high-order components (fluctuations) are zero for nx=0, ny=0
    mode
problem.add_equation('ph = 0', condition='(nx == 0) and (ny == 0)
    ')
problem.add_equation('Th = 0', condition='(nx == 0) and (ny == 0)
    ')
```

```
problem.add_equation('uh = 0', condition='(nx == 0) and (ny == 0)')
problem.add_equation('vh = 0', condition='(nx == 0) and (ny == 0)')
problem.add_equation('wh = 0', condition='(nx == 0) and (ny == 0)')
problem.add_equation('Tzh = 0', condition='(nx == 0) and (ny == 0)')
problem.add_equation('uzh = 0', condition='(nx == 0) and (ny == 0)')
problem.add_equation('vzh = 0', condition='(nx == 0) and (ny == 0)')
problem.add_equation('wzh = 0', condition='(nx == 0) and (ny == 0)')
  # match z-derivatives to Tz, uz, etc:
problem.add_equation('Tzl - dz(Tl) = 0', condition='(nx == 0) and (ny == 0)')
problem.add_equation('uzl - dz(ul) = 0', condition='(nx == 0) and (ny == 0)')
problem.add_equation('vzl - dz(vl) = 0', condition='(nx == 0) and (ny == 0)')
problem.add_equation('wzl - dz(wl) = 0', condition='(nx == 0) and (ny == 0)')

# Equations for nx!=0, ny!=0 mode
  # these equations correspond to dt(qh), also written dt(q')

  # divergence condition:
problem.add_equation('dx(uh) + dy(vh) + wzh = 0', condition='(nx != 0) or (ny != 0)')
  # thermal energy:
problem.add_equation('dt(Th) - kappa*(dx(dx(Th)) + dy(dy(Th))) - kappa_z*dz(Tzh) = -ul*dx(Th) - vl*dy(Th) - wl*Tzh - uh*dx(Tl) - vh*dy(Tl) - wh*Tzl - u_s*dx(Th)', condition='(nx != 0) or (ny != 0)')
  # Navier-Stokes/momentum equations:
problem.add_equation('dt(uh) + (1/rho_0)*dx(ph) - nu*(dx(dx(uh)) + dy(dy(uh))) - nu_z*dz(uzh) - f*vh + dx(uh*u_s)             = -ul*dx(uh) - vl*dy(uh) - wl*uzh - uh*dx(ul) - vh*dy(ul) - wh*uzl', condition='(nx != 0) or (ny != 0)')
problem.add_equation('dt(vh) + (1/rho_0)*dy(ph) - nu*(dx(dx(vh)) + dy(dy(vh))) - nu_z*dz(vzh) + f*uh + dy(uh*u_s) - u_s*dy(uh) + u_s*dx(vh) = -ul*dx(vh) - vl*dy(vh) - wl*vzh - uh*dx(vl) - vh*dy(vl) - wh*vzl - f*u_s', condition='(nx != 0) or (ny != 0)')
problem.add_equation('dt(wh) + (1/rho_0)*dz(ph) - nu*(dx(dx(wh)) + dy(dy(wh))) - nu_z*dz(wzh)       + dz(uh*u_s) - u_s*uzh    +
    u_s*dx(wh) = -ul*dx(wh) - vl*dy(wh) - wl*wzh - uh*dx(wl) - vh
    *dy(wl) - wh*wzl - 0.5*dz(u_s*u_s) + g*alpha*(Th-T_0)',
    condition='(nx != 0) or (ny != 0)')
  # low-order components (mean fields) are zero for nx!=0, ny!=0
    mode
problem.add_equation('pl = 0', condition='(nx != 0) or (ny != 0)'
    )
problem.add_equation('Tl = 0', condition='(nx != 0) or (ny != 0)'
    )
problem.add_equation('ul = 0', condition='(nx != 0) or (ny != 0)'
    )
problem.add_equation('vl = 0', condition='(nx != 0) or (ny != 0)'
    )
problem.add_equation('wl = 0', condition='(nx != 0) or (ny != 0)'
    )
problem.add_equation('Tzl = 0', condition='(nx != 0) or (ny != 0)
    ')
problem.add_equation('uzl = 0', condition='(nx != 0) or (ny != 0)
    ')
problem.add_equation('vzl = 0', condition='(nx != 0) or (ny != 0)
    ')
problem.add_equation('wzl = 0', condition='(nx != 0) or (ny != 0)
    ')
  # match z-derivatives to Tz, uz, etc:
problem.add_equation('Tzh - dz(Th) = 0', condition='(nx != 0) or
    (ny != 0)')
problem.add_equation('uzh - dz(uh) = 0', condition='(nx != 0) or
    (ny != 0)')
problem.add_equation('vzh - dz(vh) = 0', condition='(nx != 0) or
    (ny != 0)')
problem.add_equation('wzh - dz(wh) = 0', condition='(nx != 0) or
    (ny != 0)')


# Boundary conditions: nx=0, ny=0 mode
  # left z (bottom of domain)
problem.add_bc("left(Tzl) = 0", condition='(nx == 0) and (ny ==
    0)')
problem.add_bc("left(uzl) = 0", condition='(nx == 0) and (ny ==
    0)')
problem.add_bc("left(vzl) = 0", condition='(nx == 0) and (ny ==
    0)')
problem.add_bc("left(wl) = 0", condition='(nx == 0) and (ny == 0)
    ')
  # right z (ocean surface)
problem.add_bc("right(Tzl) = -q/(kappa*c_p*rho_0)", condition='(
    nx == 0) and (ny == 0)')  # heat flux
```

```
problem.add_bc("right(uzl) = tau/(nu*rho_0)", condition='(nx == 0) and (ny == 0)')  # surface wind stress
problem.add_bc("right(vzl) = 0", condition='(nx == 0) and (ny == 0)')
#problem.add_bc("right(wl) = 0", condition='(nx == 0) and (ny == 0)')  # replaced with pl gauge choice
  # pressure gauge choice
problem.add_bc("integ_z(pl) = 0", condition="(nx == 0) and (ny == 0)")

# Boundary conditions: nx!=0, ny!=0 mode
  # left z
problem.add_bc("left(Tzh) = 0", condition='(nx != 0) or (ny != 0)')
problem.add_bc("left(uzh) = 0", condition='(nx != 0) or (ny != 0)')
problem.add_bc("left(vzh) = 0", condition='(nx != 0) or (ny != 0)')
problem.add_bc("left(wh) = 0", condition='(nx != 0) or (ny != 0)')
  # right z
problem.add_bc("right(Tzh) = 0", condition='(nx != 0) or (ny != 0)')  # heat flux is a mean field, 0 for nx!=0 or ny!=0
problem.add_bc("right(uzh) = 0", condition='(nx != 0) or (ny != 0)')  # surface wind stress is also a mean field
problem.add_bc("right(vzh) = 0", condition='(nx != 0) or (ny != 0)')
problem.add_bc("right(wh) = 0", condition='(nx != 0) or (ny != 0)')
  # pressure gauge choice
#problem.add_bc("integ_z(ph) = 0", condition='(nx != 0) or (ny != 0)')  # ph does not exist for nx,ny != 0

# Build solver
solver = problem.build_solver(de.timesteppers.RK443)
logger.info('Solver built')

# Initial conditions for temperature
Tl = solver.state['Tl']
Tzl = solver.state['Tzl']
Tl.set_scales(domain.dealias, keep_data=False)
Tzl.set_scales(domain.dealias, keep_data=False)
Th = solver.state['Th']
Tzh = solver.state['Tzh']
Th.set_scales(domain.dealias, keep_data=False)
Tzh.set_scales(domain.dealias, keep_data=False)

```

```
# Define filter field function
def filter_field(field,frac=0.5):
  field.require_coeff_space()
  dom = field.domain
  local_slice = dom.dist.coeff_layout.slices(scales=dom.dealias)
  coeff = []
  for n in dom.global_coeff_shape:
    coeff.append(np.linspace(0,1,n,endpoint=False))
  cc = np.meshgrid(*coeff,indexing='ij')
  field_filter = np.zeros(dom.local_coeff_shape,dtype='bool')
  for i in range(dom.dim):
    field_filter = field_filter | (cc[i][local_slice] > frac)
  field['c'][field_filter] = 0j

# Set up variables for initial temperature
N_2 = 1.936e-5 # Brunt-Vaisala frequency [1/s^2]
MLD = -32.1 # mixed-layer depth [m]
filter_frac = 0.5

# Domain shape and slices
gshape = domain.dist.grid_layout.global_shape(scales=domain.
    dealias)
slices = domain.dist.grid_layout.slices(scales=domain.dealias)

# Initialize random noise globally
rand = np.random.RandomState(seed=70)
noise = rand.standard_normal(gshape)[slices]

# Initialize temperature
  # Th (fluctuations): in top 4.5 m, random noise of magnitude 1e
    -4; otherwise zero
  # Tl (mean field): above MLD,  background temperature of T_0;
    below MLD, linear stratification from BV frequency
temp_array_l = np.zeros(gshape)[slices]
temp_array_h = np.zeros(gshape)[slices]
z_vals = domain.grid(2, scales=domain.dealias)[0,0,:]
for ind,depth in enumerate(z_vals):
  current_shape = temp_array_l[:,:,ind].shape
  if depth > -4.5:  # surface with noise
    rand_noise = noise[:,:,ind]
    temp_array_h[:,:,ind] = 1e-4 * rand_noise
    temp_array_l[:,:,ind] = T_0 * np.ones(current_shape)
  elif depth > MLD:  # within mixed layer
    temp_array_l[:,:,ind] = T_0 * np.ones(current_shape)
  elif depth <= MLD:  # stratified deep ocean
    dist_from_MLD = abs(MLD-depth)
    T_BV = - (N_2*dist_from_MLD)/(alpha*g)
    temp_array_l[:,:,ind] = (T_0 + T_BV) * np.ones(current_shape)
Tl['g'] = 1.0 * temp_array_l
Tl.differentiate('z', out=Tzl)
Th['g'] = 1.0 * temp_array_h
logger.info('Beginning filter')
filter_field(Th,frac=filter_frac)  # apply filter to remove high-
   frequency components
logger.info('Finished filter')
Th.differentiate('z', out=Tzh)

# Integration parameters
solver.stop_sim_time = 78.*60.*60.
solver.stop_wall_time = np.inf
solver.stop_iteration = np.inf

# Analysis: full state variables
snapshots1 = solver.evaluator.add_file_handler('state_variables',
    sim_dt=21600., max_writes=10)
snapshots1.add_task(solver.state['Tl'], layout='g', name='Tl')
snapshots1.add_task(solver.state['ul'], layout='g', name='ul')
snapshots1.add_task(solver.state['vl'], layout='g', name='vl')
snapshots1.add_task(solver.state['wl'], layout='g', name='wl')
snapshots1.add_task(solver.state['Th'], layout='g', name='Th')
snapshots1.add_task(solver.state['uh'], layout='g', name='uh')
snapshots1.add_task(solver.state['vh'], layout='g', name='vh')
snapshots1.add_task(solver.state['wh'], layout='g', name='wh')

# Analysis: mean fields and correlations
snapshots2 = solver.evaluator.add_file_handler('diagnostics',
   sim_dt=21600., max_writes=10)
  # mean fields
snapshots2.add_task('u_mean', layout='g', scales=1, name='u_bar')
snapshots2.add_task('v_mean', layout='g', scales=1, name='v_bar')
snapshots2.add_task('w_mean', layout='g', scales=1, name='w_bar')
snapshots2.add_task('T_mean', layout='g', scales=1, name='T_bar')
  # correlations
snapshots2.add_task("integ((ul + uh)*(vl + vh), 'x', 'y')/(Lx*Ly)
   ", layout='g', scales=1, name='uv_corr')
snapshots2.add_task("integ((ul + uh)*(wl + wh), 'x', 'y')/(Lx*Ly)
   ", layout='g', scales=1, name='uw_corr')
snapshots2.add_task("integ((vl + vh)*(wl + wh), 'x', 'y')/(Lx*Ly)
   ", layout='g', scales=1, name='vw_corr')
snapshots2.add_task("integ((ul + uh)*(Tl + Th), 'x', 'y')/(Lx*Ly)
   ", layout='g', scales=1, name='uT_corr')
snapshots2.add_task("integ((vl + vh)*(Tl + Th), 'x', 'y')/(Lx*Ly)
   ", layout='g', scales=1, name='vT_corr')
```

```
snapshots2.add_task("integ((wl + wh)*(Tl + Th), 'x', 'y')/(Lx*Ly)
    ", layout='g', scales=1, name='wT_corr')

# CFL condition
initial_dt = 1.
CFL = flow_tools.CFL(solver, initial_dt=initial_dt, max_dt=20.,
    safety=1.2, max_change=10., cadence=1)
CFL.add_velocities(('ul + uh', 'vl + vh', 'wl + wh'))
CFL.add_velocity('u_s',0)

# Flow properties
flow = flow_tools.GlobalFlowProperty(solver, cadence=1)
flow.add_property('Tl + Th', name='Temperature')
flow.add_property('ul + uh', name='u-velocity')
flow.add_property('vl + vh', name='v-velocity')
flow.add_property('wl + wh', name='w-velocity')

# Elapsed initialization time
end_init_time = time.time()
logger.info('Initialization time: {:f}'.format(end_init_time-
    start_init_time))

# Main loop
try:
  logger.info('Starting loop')
  start_run_time = time.time()
  while solver.ok:
    dt = CFL.compute_dt()
    solver.step(dt)
    if (solver.iteration-1) % 20 == 0:
      logger.info('Iteration: {}, Time: {:e}, dt: {:e}'.format(
    solver.iteration, solver.sim_time, dt))
      logger.info('  Max/Min T = {:f}, {:f}'.format(flow.max('
    Temperature'), flow.min('Temperature')))
      logger.info('  Max/Min U = {:f}, {:f}'.format(flow.max('u-
    velocity'), flow.min('u-velocity')))
      logger.info('  Max/Min V = {:f}, {:f}'.format(flow.max('v-
    velocity'), flow.min('v-velocity')))
      logger.info('  Max/Min W = {:f}, {:f}'.format(flow.max('w-
    velocity'), flow.min('w-velocity')))
except:
  logger.error('Exception raised, triggering end of main loop.')
  raise
finally:
  end_run_time = time.time()
  logger.info('Iterations: {}'.format(solver.iteration))
  logger.info('Sim end time: {:f}'.format(solver.sim_time))
logger.info('Run time: {:.2f} sec'.format(end_run_time-
 start_run_time))
logger.info('Run time: {:f} cpu-hr'.format((end_run_time-
 start_run_time)/(60.*60.)*domain.dist.comm_cart.size))
```

Appendix C

Generalized Quasilinear Langmuir Turbulence Code

```
# Load Python packages, Dedalus
import numpy as np
from mpi4py import MPI
import time

from dedalus import public as de
from dedalus.extras import flow_tools

# Import Dedalus logger
import logging
logger = logging.getLogger(__name__)

# Begin timing initialization
start_init_time = time.time()

# Domain parameters
Lx, Ly, Lz = (288., 288., 64.8)
x_modes, y_modes, z_modes = (96, 96, 72)

# Create bases
x_basis = de.Fourier('x', x_modes, interval=(-Lx/2, Lx/2),
    dealias=3/2)
y_basis = de.Fourier('y', y_modes, interval=(-Ly/2, Ly/2),
    dealias=3/2)
z_basis = de.Chebyshev('z', z_modes, interval=(-Lz, 0), dealias
    =3/2)

# Construct domain
domain = de.Domain([x_basis, y_basis, z_basis], grid_dtype=np.
    float64, mesh=[4,8])

```

```
# Set up an initial value problem (IVP)
  # like in QL, split each state variable into low-order and high-order components, but in GQL those are not just mean fields vs fluctuations
  # still true that q = ql + qh
problem = de.IVP(domain, variables=['pl','Tl','ul','vl','wl','Tzl','uzl','vzl','wzl','ph','Th','uh','vh','wh','Tzh','uzh','vzh','wzh'], time='t')

# Metadata (optional)
problem.meta['pl','uzl','vzl','wl','Tzl','uzh','vzh','wh','Tzh']['z']['dirichlet'] = True

# Define constant parameters for the equations of motion
problem.parameters['q'] = 5. # heat flux from upper surface [W/m^2]
problem.parameters['c_p'] = 3994 # specific heat of sea water at constant pressure [J/kg*C]
problem.parameters['rho_0'] = 1035 # background density [kg/m^3]
problem.parameters['nu_z'] = 1e-3 # vertical harmonic kinematic viscosity [m^2/s]
problem.parameters['nu'] = 2e-3 # horizontal harmonic kinematic viscosity [m^2/s]
problem.parameters['kappa_z'] = 1e-3 # vertical coefficient of thermal diffusivity [m^2/s]
problem.parameters['kappa'] = 2e-3 # horizontal coefficient of thermal diffusivity [m^2/s]
problem.parameters['alpha'] = alpha = 2e-4 # coefficient of thermal expansion [1/C]
problem.parameters['T_0'] = T_0 = 20 # background temperature [C]
problem.parameters['g'] = g = 9.81 # gravitational acceleration [m/s^2]
problem.parameters['f'] = 1e-4 # Coriolis parameter [1/s]
problem.parameters['tau'] = 0.037 # surface wind stress [N/m^2]

# Make domain dimensions into problem parameters
problem.parameters['Lx'] = Lx
problem.parameters['Ly'] = Ly
problem.parameters['Lz'] = Lz

# Define Stokes field
Stokes_field = domain.new_field(name='u_s')
z = domain.grid(2)
k_s = 0.105 # Stokes wavenumber [1/m]
a_s = 0.068 # Stokes velocity at surface (z=0) [m/s]
Stokes_field['g'] = a_s * np.exp(2. * k_s * z)
Stokes_field.meta['x', 'y']['constant'] = True
```

```
problem.parameters['u_s'] = Stokes_field

# Mean Fields for Analysis
problem.substitutions['u_mean'] = "integ(ul + uh, 'x', 'y')/(Lx*Ly)"
problem.substitutions['v_mean'] = "integ(vl + vh, 'x', 'y')/(Lx*Ly)"
problem.substitutions['w_mean'] = "integ(wl + wh, 'x', 'y')/(Lx*Ly)"
problem.substitutions['T_mean'] = "integ(Tl + Th, 'x', 'y')/(Lx*Ly)"

# Equations for nx, ny <= 10
  # we have set Lambda, the cut-off frequency for GQL that distinguishes between "low-order" and "high-order", to 10

  # divergence condition:
problem.add_equation('dx(ul) + dy(vl) + wzl = 0', condition='(abs(nx) <= 10) and (abs(ny) <= 10)')
  # thermal energy:
problem.add_equation('dt(Tl) - kappa*(dx(dx(Tl)) + dy(dy(Tl))) - kappa_z*dz(Tzl) = -ul*dx(Tl) - vl*dy(Tl) - wl*Tzl - uh*dx(Th) - vh*dy(Th) - wh*Tzh - u_s*dx(Tl)', condition='(abs(nx) <= 10) and (abs(ny) <= 10)')
  # Navier-Stokes/momentum equations:
problem.add_equation('dt(ul) + (1/rho_0)*dx(pl) - nu*(dx(dx(ul)) + dy(dy(ul))) - nu_z*dz(uzl) - f*vl + dx(ul*u_s)                = -ul*dx(ul) - vl*dy(ul) - wl*uzl - uh*dx(uh) - vh*dy(uh) - wh*uzh', condition='(abs(nx) <= 10) and (abs(ny) <= 10)')
problem.add_equation('dt(vl) + (1/rho_0)*dy(pl) - nu*(dx(dx(vl)) + dy(dy(vl))) - nu_z*dz(vzl) + f*ul + dy(ul*u_s) - u_s*dy(ul) + u_s*dx(vl) = -ul*dx(vl) - vl*dy(vl) - wl*vzl - uh*dx(vh) - vh*dy(vh) - wh*vzh - f*u_s', condition='(abs(nx) <= 10) and (abs(ny) <= 10)')
problem.add_equation('dt(wl) + (1/rho_0)*dz(pl) - nu*(dx(dx(wl)) + dy(dy(wl))) - nu_z*dz(wzl)       + dz(ul*u_s) - u_s*uzl    + u_s*dx(wl) = -ul*dx(wl) - vl*dy(wl) - wl*wzl - uh*dx(wh) - vh*dy(wh) - wh*wzh - 0.5*dz(u_s*u_s) + g*alpha*(Tl-T_0)', condition='(abs(nx) <= 10) and (abs(ny) <= 10)')
  # high-order components are zero for nx,ny <= Lambda (10)
problem.add_equation('ph = 0', condition='(abs(nx) <= 10) and (abs(ny) <= 10)')
problem.add_equation('Th = 0', condition='(abs(nx) <= 10) and (abs(ny) <= 10)')
problem.add_equation('uh = 0', condition='(abs(nx) <= 10) and (abs(ny) <= 10)')
problem.add_equation('vh = 0', condition='(abs(nx) <= 10) and (
    abs(ny) <= 10)')
problem.add_equation('wh = 0', condition='(abs(nx) <= 10) and (
    abs(ny) <= 10)')
problem.add_equation('Tzh = 0', condition='(abs(nx) <= 10) and (
    abs(ny) <= 10)')
problem.add_equation('uzh = 0', condition='(abs(nx) <= 10) and (
    abs(ny) <= 10)')
problem.add_equation('vzh = 0', condition='(abs(nx) <= 10) and (
    abs(ny) <= 10)')
problem.add_equation('wzh = 0', condition='(abs(nx) <= 10) and (
    abs(ny) <= 10)')
  # match z-derivatives to Tz, uz, etc:
problem.add_equation('Tzl - dz(Tl) = 0', condition='(abs(nx) <=
    10) and (abs(ny) <= 10)')
problem.add_equation('uzl - dz(ul) = 0', condition='(abs(nx) <=
    10) and (abs(ny) <= 10)')
problem.add_equation('vzl - dz(vl) = 0', condition='(abs(nx) <=
    10) and (abs(ny) <= 10)')
problem.add_equation('wzl - dz(wl) = 0', condition='(abs(nx) <=
    10) and (abs(ny) <= 10)')

# Equations for nx, ny > 10

  # divergence condition:
problem.add_equation('dx(uh) + dy(vh) + wzh = 0', condition='(abs
    (nx) > 10) or (abs(ny) > 10)')
  # thermal energy:
problem.add_equation('dt(Th) - kappa*(dx(dx(Th)) + dy(dy(Th))) -
    kappa_z*dz(Tzh) = -ul*dx(Th) - vl*dy(Th) - wl*Tzh - uh*dx(Tl)
    - vh*dy(Tl) - wh*Tzl - u_s*dx(Th)', condition='(abs(nx) > 10)
    or (abs(ny) > 10)')
  # Navier-Stokes/momentum equations:
problem.add_equation('dt(uh) + (1/rho_0)*dx(ph) - nu*(dx(dx(uh))
    + dy(dy(uh))) - nu_z*dz(uzh) - f*vh + dx(uh*u_s)
                = -ul*dx(uh) - vl*dy(uh) - wl*uzh - uh*dx(ul) - vh*
    dy(ul) - wh*uzl', condition='(abs(nx) > 10) or (abs(ny) > 10)'
    )
problem.add_equation('dt(vh) + (1/rho_0)*dy(ph) - nu*(dx(dx(vh))
    + dy(dy(vh))) - nu_z*dz(vzh) + f*uh + dy(uh*u_s) - u_s*dy(uh)
    + u_s*dx(vh) = -ul*dx(vh) - vl*dy(vh) - wl*vzh - uh*dx(vl) -
    vh*dy(vl) - wh*vzl - f*u_s', condition='(abs(nx) > 10) or (abs
    (ny) > 10)')
problem.add_equation('dt(wh) + (1/rho_0)*dz(ph) - nu*(dx(dx(wh))
    + dy(dy(wh))) - nu_z*dz(wzh)        + dz(uh*u_s) - u_s*uzh     +
     u_s*dx(wh) = -ul*dx(wh) - vl*dy(wh) - wl*wzh - uh*dx(wl) - vh
    *dy(wl) - wh*wzl - 0.5*dz(u_s*u_s) + g*alpha*(Th-T_0)',
    condition='(abs(nx) > 10) or (abs(ny) > 10)')
  # high-order components are zero for nx,ny > Lambda (10)
problem.add_equation('pl = 0', condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_equation('Tl = 0', condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_equation('ul = 0', condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_equation('vl = 0', condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_equation('wl = 0', condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_equation('Tzl = 0', condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_equation('uzl = 0', condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_equation('vzl = 0', condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_equation('wzl = 0', condition='(abs(nx) > 10) or (abs(ny) > 10)')
  # match z-derivatives to Tz, uz, etc:
problem.add_equation('Tzh - dz(Th) = 0', condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_equation('uzh - dz(uh) = 0', condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_equation('vzh - dz(vh) = 0', condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_equation('wzh - dz(wh) = 0', condition='(abs(nx) > 10) or (abs(ny) > 10)')


# Boundary conditions: nx, ny <= 10 modes
  # left z (bottom of domain)
problem.add_bc("left(Tzl) = 0", condition='(abs(nx) <= 10) and (abs(ny) <= 10)')
problem.add_bc("left(uzl) = 0", condition='(abs(nx) <= 10) and (abs(ny) <= 10)')
problem.add_bc("left(vzl) = 0", condition='(abs(nx) <= 10) and (abs(ny) <= 10)')
problem.add_bc("left(wl) = 0", condition='(abs(nx) <= 10) and (abs(ny) <= 10)')
  # right z (ocean surface)
problem.add_bc("right(Tzl) = -q/(kappa*c_p*rho_0)", condition='(abs(nx) <= 10) and (abs(ny) <= 10)')  # heat flux
problem.add_bc("right(uzl) = tau/(nu*rho_0)", condition='(abs(nx) <= 10) and (abs(ny) <= 10)')  # surface wind stress
problem.add_bc("right(vzl) = 0", condition='(abs(nx) <= 10) and (abs(ny) <= 10)')
problem.add_bc("right(wl) = 0", condition='((abs(nx) <= 10) and (abs(ny) <= 10)) and ((nx != 0) or (ny != 0))')  # replace nx,ny=0 with pl gauge choice
  # pressure gauge choice
problem.add_bc("integ_z(pl) = 0", condition="(nx == 0) and (ny == 0)")

# Boundary conditions: nx, ny > 10 modes
  # left z
problem.add_bc("left(Tzh) = 0", condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_bc("left(uzh) = 0", condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_bc("left(vzh) = 0", condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_bc("left(wh) = 0", condition='(abs(nx) > 10) or (abs(ny) > 10)')
  # right z
problem.add_bc("right(Tzh) = 0", condition='(abs(nx) > 10) or (abs(ny) > 10)')  # heat flux
problem.add_bc("right(uzh) = 0", condition='(abs(nx) > 10) or (abs(ny) > 10)')  # surface wind stress
problem.add_bc("right(vzh) = 0", condition='(abs(nx) > 10) or (abs(ny) > 10)')
problem.add_bc("right(wh) = 0", condition='(abs(nx) > 10) or (abs(ny) > 10)')
  # pressure gauge choice
#problem.add_bc("integ_z(ph) = 0", condition='(abs(nx) > 10) or (abs(ny) > 10)')  # ph does not exist for nx,ny != 0

# Build solver
solver = problem.build_solver(de.timesteppers.RK443)
logger.info('Solver built')

# Initial conditions
Tl = solver.state['Tl']
Tzl = solver.state['Tzl']
Tl.set_scales(domain.dealias, keep_data=False)
Tzl.set_scales(domain.dealias, keep_data=False)
Th = solver.state['Th']
Tzh = solver.state['Tzh']
Th.set_scales(domain.dealias, keep_data=False)
Tzh.set_scales(domain.dealias, keep_data=False)

# Define filter field function
```

```python
def filter_field(field,frac=0.5):
  field.require_coeff_space()
  dom = field.domain
  local_slice = dom.dist.coeff_layout.slices(scales=dom.dealias)
  coeff = []
  for n in dom.global_coeff_shape:
    coeff.append(np.linspace(0,1,n,endpoint=False))
  cc = np.meshgrid(*coeff,indexing='ij')
  field_filter = np.zeros(dom.local_coeff_shape,dtype='bool')
  for i in range(dom.dim):
    field_filter = field_filter | (cc[i][local_slice] > frac)
  field['c'][field_filter] = 0j

# Set up variables for initial temperature
N_2 = 1.936e-5 # Brunt-Vaisala frequency [1/s^2]
MLD = -32.1 # mixed-layer depth [m]
filter_frac = 0.5

# Domain shape and slices
gshape = domain.dist.grid_layout.global_shape(scales=domain.
    dealias)  # [x, y, z] dimensions of full domain
slices = domain.dist.grid_layout.slices(scales=domain.dealias)  #
     takes domain slice of current process (for running in
    parallel)

# Initialize random noise globally
rand = np.random.RandomState(seed=80)
noise = rand.standard_normal(gshape)[slices]

# Initialize temperature
  # Th: in top 4.5 m, random noise of magnitude 1e-4; otherwise
    zero
  # Tl: in top 4.5 m, background temperature of T_0 + random
    noise of magnitude 1e-4; above MLD but below 4.5 m,
    background temperature of T_0; below MLD, linear
    stratification from BV frequency
temp_array_l = np.zeros(gshape)[slices]
temp_array_h = np.zeros(gshape)[slices]
z_vals = domain.grid(2, scales=domain.dealias)[0,0,:]
for ind,depth in enumerate(z_vals):
  current_shape = temp_array_l[:,:,ind].shape
  if depth > -4.5:  # surface with noise
    rand_noise = noise[:,:,ind]
    temp_array_h[:,:,ind] = 1e-4 * rand_noise
    temp_array_l[:,:,ind] = T_0 * np.ones(current_shape) + (1e-4
   * rand_noise)
  elif depth > MLD:  # within mixed layer
```

```
      temp_array_l[:,:,ind] = T_0 * np.ones(current_shape)
    elif depth <= MLD:  # stratified deep ocean
      dist_from_MLD = abs(MLD-depth)
      T_BV = - (N_2*dist_from_MLD)/(alpha*g)
      temp_array_l[:,:,ind] = (T_0 + T_BV) * np.ones(current_shape)
Tl['g'] = 1.0 * temp_array_l
logger.info('Beginning Tl filter')
filter_field(Tl,frac=filter_frac)  # apply filter to remove high-frequency components (both Tl and Th have high-frequency random noise for GQL)
logger.info('Finished Tl filter')
Tl.differentiate('z', out=Tzl)

Th['g'] = 1.0 * temp_array_h
logger.info('Beginning Th filter')
filter_field(Th,frac=filter_frac)  # apply filter to remove high-frequency components
logger.info('Finished Th filter')
Th.differentiate('z', out=Tzh)

# Integration parameters
solver.stop_sim_time = 78.*60.*60.
solver.stop_wall_time = np.inf
solver.stop_iteration = np.inf

# Analysis: full state variables
snapshots1 = solver.evaluator.add_file_handler('state_variables', sim_dt=21600., max_writes=10)
snapshots1.add_task(solver.state['Tl'], layout='g', name='Tl')
snapshots1.add_task(solver.state['ul'], layout='g', name='ul')
snapshots1.add_task(solver.state['vl'], layout='g', name='vl')
snapshots1.add_task(solver.state['wl'], layout='g', name='wl')
snapshots1.add_task(solver.state['Th'], layout='g', name='Th')
snapshots1.add_task(solver.state['uh'], layout='g', name='uh')
snapshots1.add_task(solver.state['vh'], layout='g', name='vh')
snapshots1.add_task(solver.state['wh'], layout='g', name='wh')

# Analysis: mean fields and correlations
snapshots2 = solver.evaluator.add_file_handler('diagnostics', sim_dt=21600., max_writes=10)
  # mean fields
snapshots2.add_task('u_mean', layout='g', scales=1, name='u_bar')
snapshots2.add_task('v_mean', layout='g', scales=1, name='v_bar')
snapshots2.add_task('w_mean', layout='g', scales=1, name='w_bar')
snapshots2.add_task('T_mean', layout='g', scales=1, name='T_bar')
  # correlations
```

```
snapshots2.add_task("integ((ul + uh)*(vl + vh), 'x', 'y')/(Lx*Ly)
    ", layout='g', scales=1, name='uv_corr')
snapshots2.add_task("integ((ul + uh)*(wl + wh), 'x', 'y')/(Lx*Ly)
    ", layout='g', scales=1, name='uw_corr')
snapshots2.add_task("integ((vl + vh)*(wl + wh), 'x', 'y')/(Lx*Ly)
    ", layout='g', scales=1, name='vw_corr')
snapshots2.add_task("integ((ul + uh)*(Tl + Th), 'x', 'y')/(Lx*Ly)
    ", layout='g', scales=1, name='uT_corr')
snapshots2.add_task("integ((vl + vh)*(Tl + Th), 'x', 'y')/(Lx*Ly)
    ", layout='g', scales=1, name='vT_corr')
snapshots2.add_task("integ((wl + wh)*(Tl + Th), 'x', 'y')/(Lx*Ly)
    ", layout='g', scales=1, name='wT_corr')

# CFL condition
initial_dt = 1.
CFL = flow_tools.CFL(solver, initial_dt=initial_dt, max_dt=20.,
    safety=1.2, max_change=10., cadence=1)
CFL.add_velocities(('ul + uh', 'vl + vh', 'wl + wh'))
CFL.add_velocity('u_s',0)

# Flow properties
flow = flow_tools.GlobalFlowProperty(solver, cadence=1)
flow.add_property('Tl + Th', name='Temperature')
flow.add_property('ul + uh', name='u-velocity')
flow.add_property('vl + vh', name='v-velocity')
flow.add_property('wl + wh', name='w-velocity')

# Elapsed initialization time
end_init_time = time.time()
logger.info('Initialization time: {:f}'.format(end_init_time-
    start_init_time))

# Main loop
try:
  logger.info('Starting loop')
  start_run_time = time.time()
  while solver.ok:
    dt = CFL.compute_dt()
    solver.step(dt)
    if (solver.iteration-1) % 20 == 0:
      logger.info('Iteration: {}, Time: {:e}, dt: {:e}'.format(
    solver.iteration, solver.sim_time, dt))
      logger.info('  Max/Min T = {:f}, {:f}'.format(flow.max('
    Temperature'), flow.min('Temperature')))
      logger.info('  Max/Min U = {:f}, {:f}'.format(flow.max('u-
    velocity'), flow.min('u-velocity')))
```

```
      logger.info('  Max/Min V = {:f}, {:f}'.format(flow.max('v-
    velocity'), flow.min('v-velocity')))
      logger.info('  Max/Min W = {:f}, {:f}'.format(flow.max('w-
    velocity'), flow.min('w-velocity')))
except:
  logger.error('Exception raised, triggering end of main loop.')
  raise
finally:
  end_run_time = time.time()
  logger.info('Iterations: {}'.format(solver.iteration))
  logger.info('Sim end time: {:f}'.format(solver.sim_time))
  logger.info('Run time: {:.2f} sec'.format(end_run_time-
    start_run_time))
  logger.info('Run time: {:f} cpu-hr'.format((end_run_time-
    start_run_time)/(60.*60.)*domain.dist.comm_cart.size))
```