Uncertainty Quantification for models of chaotic mixing

Ian Towey
04128591

The thesis is submitted to University College Dublin
in part fulfilment of the requirements for the degree of
MSc Data and Computational Science

School of Mathematics and Statistics
University College Dublin

Supervisor: Dr. Lennon Ó Naráigh

August 20, 2018

Abstract

A stochastic model of the orientation dynamics of a tracer gradient mixed by a steady flow is presented. The stochastic model contains a mean and random part for the X from the orientation dynamics model. The random components of the model are assumed to be described as Ornstein-Uhlenbeck processes, from this, the associated Fokker-Planck partial differential equation is derived. This models the stationary probability density function of the stochastic system. The probability density function of the Lyapunov exponent, which measures the level of chaotic mixing, can then be computed. The parameters of the Fokker-Planck equation are validated against data obtained from a numerical simulation of turbulent flow, also the density functions of the angles and Lyapunov exponent can be compared against the density functions from the simulated data. Methods from Uncertainty Quantification are presented, these methods may help fit the parameters of the Fokker-Planck equation better than the moments of the simulated data presented in other works.

Acknowledgments

I would like to thank my supervisor, Dr. Lennon Ó Naŕaigh, for him guidance and and advice on this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This paper investigates properties of turbulent fluid flow. Turbulent fluid flow is a complex and requires advanced mathematics to describe this motion. The motion of a fluid is usually though as of a group of unit fluid elements with similar physical properties, the mathematics is modelled on how the fluid motion effects a single fluid element. In a turbulent flow there is mixing of fluid particles, which are stretched, distorted and mixed in knots by the flow. In this setting, the motion of the fluid element is highly irregular and statistical averages are used to describe the various properties of the fluid. Turbulent fluid flow has important application in industrial mixing applications, examples outlined cite:mixingwiki and cite:fluidmixing.

This paper investigates using a stochastic model to describe the *mixing/turbulent* motion of a passive tracer in a fluid flow. Rapid mixing or Turbulent flow is conventionally visualised as a cascade of large eddies breaking into successfully smaller eddies and the transfer of energy from these larger unstable eddies to smaller eddies. A passive tracer (such as a dye) is any fluid property we can measure to track fluid flow that does not influence the properties of the flow. Turbulent / mixing action stretches and distorts these fluid elements. The rate of stretching/distortion is described by a quantity known as the Lyapunov exponent. The solution to the stochastic model provides the probability density function of the Lyapunov exponent.

In cite:main, a model of the orientation dynamics is derived based on the alignment dynamics of the tracer gradient with the straining direction of the flow. The orientation dynamics model emerges from the standard advection-diffusion equation (ref eqn) for viscous fluids by introducting a vector field which describes the the gradient of the tracer. The eigenvalues and eigenvectors of the rate-of-strain tensor associated with the orientation dynamics model are the basis from which the stochastic model for the Lyapunov exponent PDF is found. The stochastic fluctuations are modeled as as Ornstein-Uhlenbeck (OU) processes, this is an appropriate assumption as the analysis in cite:main introduces a external random forcing in the simulation. The Fokker-Planck partial differential equation associated with the stochastic model is presented, which describes the stationary solution of the stochastic model. The parameters of the Fokker-Planck were choosen via the moment of the vorticity and tracer concentration from the DNS. One of the aims of this project is to try and use uncertainty quanification methods to fite these parameters more accurately.

The aim of this paper is to fully dervice the equations presented in cite:main, numerically solve the vorticity, advection diffusion and the Fokker-Planck equations. Reproducing the simulation results in cite:main and use uncertainty quatification methods to fit the FP parameters more actually accurately. Uncertainty quantification (UQ) methods are planned to be used to verify and validate the results of the stochastic model against the output of direct numerical simulation of the vorticity/advection-diffusion equations. This allows us to understand the expected uncertainty in the output of the model and quantify the error from the experiment. A number of open-source UQ libraries are investigated to see if they can be used to solve the Fokker-Planck equation and fit the parameters to lead to a more accurate fit between the stochastic model and the direct numerical simulation output.

First we review and detail the underlying theory, derive the stochastic model and present the numerical schemes.

Chapter 2

Theory and Methods

2.1 Vorticity

The Navier-Stokes equations are the complete equations of motion for a viscous Newtonian fluid.

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \cdot \mathbf{u} = -\nabla p + (-1)^p \nu_p \nabla^{2p} \mathbf{u} + \mathbf{F}(\mathbf{x}, t) - \mathbf{D}(\mathbf{x}, t) \tag{2.1}$$

In this setting, a forcing term $\mathbf{F}$ and dissapative term $\mathbf{D}$ are added, these are added to stop energy build up at small scales in the direct numercal simulation. The viscoity term is taken to be hyperviscous of order p. This hyperviscosity also helps to stablise the direct numercal simulation of 2.1 and avoid singular solutions. The flow is assumed to be incompressible, $\nabla \cdot \mathbf{u} = 0$.

The numerical simulation is not performed directly against the Navier-stokes equation. Instead the simulated data is generated from solving the 2-D Vorticity equation as we are intersted in the orientation dynamics of the fluid elements, and the vorticity provides up with the spinning of a fluid element, from which simulated probability density function of the lyapunov exponent can be extracted.

The following outlines the derivation of the vorticity transport eqaution from equation 2.1. The vorticity equation can be derived by

- takes the curl of equation 2.1
- defining $\omega = \nabla \times \mathbf{u}$

$$\nabla \times \left[\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \cdot \mathbf{u} = -\nabla p + (-1)^p \nu_p \nabla^{2p} \mathbf{u} + \mathbf{F}(\mathbf{x}, t) - \mathbf{D}(\mathbf{x}, t)\right] \tag{2.2}$$

$$\frac{\partial \omega}{\partial t} + \underbrace{\nabla \times \Big[(\mathbf{u} \cdot \nabla) \cdot \mathbf{u}\Big]}_{\text{Convective term}} = \underbrace{\nabla \times \Big[-\nabla p\Big]}_{\text{Pressure gradient}} + \underbrace{\nabla \times \Big[(-1)^p \nu_p \nabla^{2p} \mathbf{u}\Big]}_{\text{Hyperviscosity}} + \nabla \times \Big[\mathbf{F}(\mathbf{x}, t)\Big] - \nabla \times \Big[\mathbf{D}(\mathbf{x}, t)\Big] \tag{2.3}$$

Using the following vector identities and results for any vectors ω and $\mathbf{u}$

1. incompressible fluid, $\nabla \cdot \mathbf{u} = 0$
2. for a scalar, the curl of the gradient of a scalar is zero, ϕ, $\nabla \times \nabla\phi = 0$
3. the div of the curl of a vector is zero, $\nabla \cdot \omega = \nabla \cdot (\nabla \times \mathbf{u}) = 0$
4. $\frac{1}{2}\nabla(\mathbf{u} \cdot \mathbf{u}) = (\mathbf{u} \cdot \nabla)\mathbf{u} + \mathbf{u} \times (\nabla \times \mathbf{u})$
5. $\nabla \times (\mathbf{u} \times \omega) = (\omega \cdot \nabla)\mathbf{u} - (\mathbf{u} \cdot \nabla)\omega + \mathbf{u} \underbrace{\nabla \cdot \omega}_{= 0 \text{ (from 3)}} - \omega \underbrace{\nabla \cdot \mathbf{u}}_{= 0 \text{ (from 1)}}$

Applying these to the Convective term in 2.3 yields

$$\begin{aligned}
\nabla \times \Big[(\mathbf{u} \cdot \nabla) \cdot \mathbf{u}\Big] &= \nabla \times \Big[\frac{1}{2}\nabla(\mathbf{u} \cdot \mathbf{u}) - \mathbf{u} \times (\underbrace{\nabla \times \mathbf{u}}_{\omega})\Big] \\
&= \frac{1}{2}\underbrace{\nabla \times \nabla(\mathbf{u} \cdot \mathbf{u})}_{= 0 \text{ (from 2)}} - \nabla \times (\mathbf{u} \times \omega) \\
&= (\omega \cdot \nabla)\mathbf{u} - (\mathbf{u} \cdot \nabla)\omega
\end{aligned}$$

The pressure term in 2.3 goes away as the curl of the gradient of a scalar is zero from (2).

$$\frac{\partial \omega}{\partial t} + (\mathbf{u} \cdot \nabla)\omega - (\omega \cdot \nabla)\mathbf{u} = (-1)^p \nu_p \nabla^{2p}\omega + \nabla \times \Big[\mathbf{F}(\mathbf{x}, t)\Big] - \nabla \times \Big[\mathbf{D}(\mathbf{x}, t)\Big] \tag{2.4}$$

For 2-d flows, vortex stretching is absent since $\mathbf{u} = u(x, y)\mathbf{e}_x + v(x, y)\mathbf{e}_y$ and $\omega = \omega(x, y)\mathbf{e}_z$, therfore the vortex stretching term is zero $(\omega \cdot \nabla)\mathbf{u} = 0$ and letting $\nabla \times \mathbf{F}(\mathbf{x}, t) = \mathbf{Q}$ and $\nabla \times \mathbf{D}(\mathbf{x}, t) = \mathbf{N}$, 2.4 is resolved to

$$\frac{\partial \omega}{\partial t} + (\mathbf{u} \cdot \nabla)\omega = (-1)^p \nu_p \nabla^{2p}\omega + \mathbf{Q} - \mathbf{N} \tag{2.5}$$

For the purposes of the direct numerical simulation we will define the $\mathbf{D}$ and $\mathbf{N}$ terms here as they will be required when outlining the numerical algorithm next. Defining the damping term as a constant, $\mathbf{N} = \nu_0\omega$ and from [7] the forcing term is defined by the difference equation $\mathbf{Q}_{n+1} = R_n\mathbf{Q}_n + (1 - R_n^2)^{1/2}x_*$ where R_n is a dimenionless correlation coefficient and $x_* \sim \mathcal{N}(0, 1)$, this forcing term is chosen so that a steady turbulent flow is produced in a long running simulation.

Before solving the vorticity equation, we outline some assumptions that aid and simplify the numerical simulation. ω is defined on the 2-d grid domain $[0, L] \times [0, L]$ and ω

is periodic over a wavelength L. This yields the initial conditions.

$$\omega(x,y,t) = \omega(x+L,y,t)$$
$$\omega(x,y,t) = \omega(x,y+L,t)$$

Discretizing 2.5 and adding in the defined terms for damping and forcing yeilds the n-th time-step difference equation

$$\frac{\omega_{n+1}-\omega_n}{\Delta t} + (\mathbf{u}\cdot\nabla\omega)_n = (-1)^p\nu_p\nabla^{2p}\left(\frac{\omega_{n+1}-\omega_n}{2}\right) + \left[R_nQ_n + (1-R_n^2)^{1/2}x_*\right] - \nu_0\omega_n \quad (2.6)$$

Since we have assumed periodic boundary conditions , 2.6 can be transformed to an equation in fourier space and solved using a simpler DFT (which used the FFT for optimal performance) approach rather than a finite-dufference methods.

In Fourier space, the forcing term is also subject to a further constraint, we want to inject energy into the system only within a certain range to produce a steady flow, see [9] and [7]. We defined the binary scaling matrix M^*, with entries

$$M^*_{ij} = \begin{cases} 1 & \text{if } k_{min}\frac{2\pi}{L} < kx_i^2 + ky_j^2 < k_{max}\frac{2\pi}{L} \\ 0 & \text{otherwise} \end{cases}$$

This makes the forcing term non-zero only in the specified range in fourier space.

Another issue with computation in fourier space is that multiplication of two wave numbers can produce a number that is smaller than its factors or a wave numbers with infinite value. A method known as dealiasing is used to resolved this, this is discussed in depth in [4],. The dealiasing method used here is known as the 2/3-rule, this truncates the computed waves numbers which blowup. The binary matrix D^* implements the truncating 2/3-rule.

$$D^*_{ij} = \begin{cases} 1 & \text{if } |kx_i| < \frac{2}{3}max_{1\le i\le N}kx_i \cap |ky_j| < \frac{2}{3}max_{1\le j\le N}ky_j \\ 0 & \text{otherwise} \end{cases}$$

Adding these yields the final representation of the numerical algorithm in wavespace. The wavenumber are then transformed back to real numbers using a reverse fourier transform.

$$\hat{\omega}_{n+1} = D^*\left[\frac{\hat{\omega}_n - \Delta t\left(u\hat{\cdot}\omega + \frac{|k^{2p}|\hat{\omega}_n}{2} + M^*\left[R_nQ_n + (1-R_n^2)^{1/2}x_*\right] - \nu_0\hat{\omega_n}\right)}{1 + \frac{\nu_p|k^{2p}\Delta t|}{2}}\right] \quad (2.7)$$

2.2 Advection-Diffusion equation & relation to vorticity

Now we focus on the velocity of the concentration of the tracer gradient. This is defined using the advection-diffusion equation with hyperdiffusivity to stablise the numerical simulation.

$$\frac{\partial\theta}{\partial t} + \mathbf{u}\cdot\nabla\theta = -(-1)^p\nu_p\nabla^{2p}\theta \tag{2.8}$$

We need to relate the velocity $\mathbf{u} = (u, v)$ to the vorticity ω. In two-dimensional flow of an incompressible viscous fluid, a stream function may be defined. The stream function ψ is a scalar function defined as

$$u = -\frac{\partial\psi}{\partial y}; \qquad v = \frac{\partial\psi}{\partial x}$$

As we are restricting outselves to 2-d. ω can be reduced to 1-d by computing the vorticity vector

$$\bar{\omega} = \begin{bmatrix} \hat{x} & \hat{y} & \hat{z} \\ \partial_x & \partial_y & \partial_z \\ u(x,y,0,t) & u(x,y,0,t) & 0 \end{bmatrix}$$

$$= \hat{x}\cdot 0 - \hat{y}\cdot 0 + \hat{z}\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right)$$

Plugging the streamfunctions for u and v above gives;

$$\begin{aligned}\omega &= \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \\ &= \frac{\partial^2\psi}{\partial^2 x} - \frac{\partial^2\psi}{\partial^2 y} \\ &= \nabla^2\psi\end{aligned}$$

This relation will allow us to solve the advection diffusion equation in the same numerical simulation as the vorticity.

Using the same assumptions (periodic boundary conditions, scaling issue and dealiasing issue etc) as for the vorticity numerical scheme, a spectral method is also used to solve for θ, the concentration of the tracer being mixed by the flow. Again discretizing and transforming to wave space yield the difference equation

$$\hat{\theta}_{n+1} = D^* \left[\frac{\hat{\theta}_n - \Delta t \hat{\theta}_n^{conv}}{1 + \Delta t(-1)^p * ((-1)^p)\nu_p |\mathbf{k}|^p} \right] \tag{2.9}$$

where

$$\begin{aligned} \theta^{conv} &= \mathbf{u} \cdot \nabla \theta \\ &= u \cdot \frac{\partial \theta}{\partial x} - v \cdot \frac{\partial \theta}{\partial y} \end{aligned}$$

Therefore

$$\begin{aligned} \hat{\theta}^{conv} &= u \cdot \frac{\partial \hat{\theta}}{\partial x} - v \cdot \frac{\partial \hat{\theta}}{\partial y} \\ &= u \cdot ik_x \hat{\theta} - vik_y \hat{\theta} \end{aligned}$$

2.3 Orientation Dynamics

The context in which the orientation dynamics model is set, starts with defining the 2-D vector field $\mathcal{B} = (-\theta_y, \theta_x)$ Where θ is the concentration of the passively advected tracer in an incompressible flow and diffusion , as outlined in [9].

In this setting, the governing equation is the advection-diffusion

$$\frac{\partial \theta}{\partial t} + \mathbf{u} \cdot \nabla \theta = 0 \tag{2.10}$$

Where $\mathbf{u} = (u, v)$ is the velocity field of the fluid in 2-dimensions.

Investigating the θ under the action of the advection-diffusion equation 2.10 separately in the x and y direction

$$\begin{aligned} \frac{\partial}{\partial t}\frac{\partial \theta}{\partial x} + \mathbf{u} \cdot \nabla \frac{\partial \theta}{\partial x} + \frac{\partial \mathbf{u}}{\partial x} \cdot (\nabla \theta) = 0 \\ \frac{\partial}{\partial t}\frac{\partial \theta}{\partial y} + \mathbf{u} \cdot \nabla \frac{\partial \theta}{\partial y} + \frac{\partial \mathbf{u}}{\partial y} \cdot (\nabla \theta) = 0 \end{aligned}$$

Writing this in matrix notation

$$\frac{\partial}{\partial t} \begin{pmatrix} -\theta_y \\ \theta_x \end{pmatrix} + \mathbf{u} \cdot \nabla \begin{pmatrix} -\theta_y \\ \theta_x \end{pmatrix} = \begin{pmatrix} \mathbf{u}_y \cdot (\nabla \theta) \\ -\mathbf{u}_x \cdot (\nabla \theta) \end{pmatrix}$$

Subsituting in $\mathcal{B}$ the gradient of the vector field and multiplying out the RHS

$$\frac{\partial \mathcal{B}}{\partial t} + \mathbf{u} \cdot \nabla \mathcal{B} = \begin{bmatrix} u_y \cdot \theta_x & v_y \cdot \theta_y \\ -u_x \cdot \theta_x & -v_x \cdot \theta_y \end{bmatrix}$$

$$\frac{\partial \mathcal{B}}{\partial t} + \mathbf{u} \cdot \nabla \mathcal{B} = \begin{bmatrix} -v_y & u_y \\ v_x & -u_x \end{bmatrix} \begin{pmatrix} -\theta_y \\ \theta_x \end{pmatrix}$$

Making use of the incompressiblity condition

$$\nabla \cdot \mathbf{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \Rightarrow \frac{\partial u}{\partial x} = -\frac{\partial v}{\partial y}$$

and replacing the negative signed values in the matrix on the RHS above yields

$$\frac{\partial \mathcal{B}}{\partial t} + \mathbf{u} \cdot \nabla \mathcal{B} = \begin{bmatrix} u_x & u_y \\ v_x & v_y \end{bmatrix} \begin{pmatrix} -\theta_y \\ \theta_x \end{pmatrix}$$

$$\frac{\partial \mathcal{B}}{\partial t} + \mathbf{u} \cdot \nabla \mathcal{B} = \mathcal{B} \cdot \nabla \mathbf{u} \tag{2.11}$$

Writing 2.11 using the material derivative operator, $\frac{d}{dt} = \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla$ and computing the dot product of 2.11 with $\mathcal{B}$

$$\mathcal{B} \left(\frac{d\mathcal{B}}{dt} \right) = \mathcal{B} \left(\mathcal{B} \cdot \nabla \mathbf{u} \right)$$

$$\frac{1}{2} \frac{d}{dt} \mathcal{B}^2 = \langle \mathcal{B} \, , (\nabla \mathbf{u}) \, \mathcal{B} \rangle$$

From matrix theory , any $N \times N$ square matrix $\mathbf{M}$ can be written as a sum of its symmetric and anti-symmetric parts

$\mathbf{M} = \mathbf{S} + \mathbf{S}^*$

where

$\mathbf{S} = \frac{\mathbf{M} + \mathbf{M}^T}{2}$

$\mathbf{S}^* = \frac{\mathbf{M} - \mathbf{M}^T}{2}$

Therefore the 2x2 martix $\nabla \cdot \mathbf{u}$ can be decomposed into the sum of its symmetric and anti-symmetric parts

$$\frac{1}{2} \frac{d}{dt} \mathcal{B}^2 = \langle \mathcal{B} \, , [(\nabla \mathbf{u})_s + (\nabla \mathbf{u})_{s^*}] \, \mathcal{B} \rangle$$

Since we only care about the symmetric part of the $\nabla \cdot \mathbf{u}$

$$\frac{1}{2}\frac{d}{dt}|\mathcal{B}|^2 = \langle \mathcal{B}\,, (\nabla \mathbf{u})_s\, \mathcal{B}\rangle$$

$$\boxed{\begin{aligned}(\nabla \mathbf{u})_s &= \frac{1}{2}\left(\nabla \mathbf{u} + (\nabla \mathbf{u})^T\right)\\ &= \frac{1}{2}\left(\begin{bmatrix} u_x & u_y \\ v_x & v_y\end{bmatrix} + \begin{bmatrix} u_x & v_x \\ u_y & v_y\end{bmatrix}\right)\\ &= \begin{bmatrix} u_x & \frac{v_x+u_y}{2} \\ \frac{v_x+u_y}{2} & v_y\end{bmatrix}\\ &= \begin{bmatrix} s & d \\ d & -s\end{bmatrix}\\ &= \mathcal{S}\ \text{(Rate-of-strain matrix)}\end{aligned}}$$

$$\frac{1}{2}\frac{d}{dt}|\mathcal{B}|^2 = \langle \mathcal{B}\,, \mathbf{S}\mathcal{B}\rangle$$

Regarding $\mathcal{B}$ as a complex-valued function of the complex variable $z = x + iy$ and $\bar{z} = x - iy$, $\mathcal{B} = \mathcal{B}_1 + i\mathcal{B}_2$

$$.\frac{1}{2}\frac{d}{dt}|\mathcal{B}|^2 = (\mathcal{B}_1, \mathcal{B}_2)\begin{bmatrix} s & d \\ d & -s\end{bmatrix}\begin{pmatrix}\mathcal{B}_1 \\ \mathcal{B}_2\end{pmatrix} \tag{2.12}$$

This produces an equation for the magnitude of the tracer gradient, we want to understand the associated angle β of this complex-valued function.

The advection equation for β is found by defining $\tan\beta = \frac{\theta_y}{\theta_x}$, and plugging this into the 2.10

$$\partial_t \tan\beta + \mathbf{u}\cdot\nabla\tan\beta = 0$$

Expanding all terms

$$\begin{aligned}(\partial_t + \mathbf{u}\cdot\nabla)\tan\beta &= \frac{1}{\theta_x^2}\left[-\theta_x(\mathcal{B}\cdot\nabla)u - \theta_y(\mathcal{B}\cdot\nabla)v\right]\\ (\partial_t + \mathbf{u}\cdot\nabla)\beta &= \frac{1}{|\mathcal{B}|^2}\left[-\theta_x(\mathcal{B}\cdot\nabla)u - \theta_y(\mathcal{B}\cdot\nabla)v\right]\\ (\partial_t + \mathbf{u}\cdot\nabla)\beta &= \frac{1}{|\mathcal{B}|^2}(-\theta_x, -\theta_y)\left[(\nabla\mathbf{u})\mathcal{B}\right]\end{aligned}$$

Expanding $\nabla\mathbf{u}$ into the su of its symmetric and antisymmetric parts

$$\boxed{\begin{aligned}
\nabla\mathbf{u} &= \frac{1}{2}\left(\nabla\mathbf{u} + (\nabla\mathbf{u})^T\right) - \frac{1}{2}\left(\nabla\mathbf{u} - (\nabla\mathbf{u})^T\right) \\
&= \frac{1}{2}\left(\begin{bmatrix} u_x & u_y \\ v_x & v_y \end{bmatrix} + \begin{bmatrix} u_x & v_x \\ u_y & v_y \end{bmatrix}\right) + \frac{1}{2}\left(\begin{bmatrix} u_x & u_y \\ v_x & v_y \end{bmatrix} + \begin{bmatrix} u_x & v_x \\ u_y & v_y \end{bmatrix}\right) \\
&= \begin{bmatrix} u_x & \frac{v_x+u_y}{2} \\ \frac{v_x+u_y}{2} & v_y \end{bmatrix} + \begin{bmatrix} 0 & \frac{-v_x+u_y}{2} \\ \frac{v_x-u_y}{2} & 0 \end{bmatrix} \\
&= \begin{bmatrix} u_x & \frac{v_x+u_y}{2} \\ \frac{v_x+u_y}{2} & v_y \end{bmatrix} - \begin{bmatrix} 0 & \frac{v_x-u_y}{2} \\ \frac{-(v_x-u_y)}{2} & 0 \end{bmatrix} \\
&= \mathcal{S} - \frac{\omega}{2}\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}
\end{aligned}}$$

$$\begin{aligned}
(\partial_t + \mathbf{u}\cdot\nabla)\,\beta &= \frac{1}{|\mathcal{B}|^2}(-\theta_x, -\theta_y)\left[\left(\mathcal{S} - \frac{\omega}{2}\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}\right)\mathcal{B}\right] \\
(\partial_t + \mathbf{u}\cdot\nabla)\,\beta &= \frac{1}{|\mathcal{B}|^2}\left[(-\theta_x, -\theta_y)\,\mathcal{S}\mathcal{B} - \frac{\omega}{2}(-\theta_x, -\theta_y)\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}\mathcal{B}\right] \\
(\partial_t + \mathbf{u}\cdot\nabla)\,\beta &= \frac{\omega}{2} - \frac{1}{|\mathcal{B}|^2}(-\theta_x, -\theta_y)\,\mathcal{S}\mathcal{B}
\end{aligned}$$

$$\frac{d\beta}{dt} = \frac{\omega}{2} - \frac{1}{|\mathcal{B}|^2}(\mathcal{B}_2, -\mathcal{B}_1)\begin{bmatrix} s & d \\ d & -s \end{bmatrix}\begin{pmatrix} \mathcal{B}_1 \\ \mathcal{B}_2 \end{pmatrix} \tag{2.13}$$

Finally we want to re-write 2.12 in terms of the the rate-of-strain matrix $\mathcal{S}$, which is the symmetric part of matrix $\nabla\mathbf{u}$.

The eigenvalues $\mathcal{S}$ are real

$$\begin{aligned}
\lambda_{(+)} &= \sqrt{s^2 + d^2} \\
\lambda_{(-)} &= -\sqrt{s^2 + d^2}
\end{aligned}$$

And orthonormal eigenvectors

$$\mathbf{X}_{(+)} = -\frac{1}{\mathcal{N}}\begin{pmatrix} 1 \\ -\alpha + \sqrt{\alpha^2+1} \end{pmatrix}$$
$$\mathbf{X}_{(-)} = -\frac{1}{\mathcal{N}}\begin{pmatrix} \alpha - \sqrt{\alpha^2+1} \\ 1 \end{pmatrix}$$

where $\alpha = s/d$ and $\mathcal{N} = \sqrt{2\sqrt{\alpha^2+1}\left(-\alpha + \sqrt{(\alpha^2+1)}\right)}$

Since $\mathbf{X}_{(+)}$ and $\mathbf{X}_{(-)}$ are orthonormal, they can be re-expressed in terms of an angle φ, the angle φ is the angle between the x-axis and the expanding direction of the straining flow.

$$\mathbf{X}_{(+)} = \begin{pmatrix} \cos\varphi \\ \sin\varphi \end{pmatrix}$$
$$\mathbf{X}_{(-)} = \begin{pmatrix} -\sin\varphi \\ \cos\varphi \end{pmatrix}$$

φ is the angle we want to model as part of the SDE model introduced later.

Computing inner product $\mathcal{B}$ and $\mathcal{SB}$ as outlined in [8] yields the ODE model for the orientation dynamics

$$\frac{d}{dt}|\mathbf{B}^2| = -2\lambda\sin\zeta|\mathbf{B}^2| \tag{2.14}$$

where the term $\Lambda = -2\lambda\sin\zeta$ is the growth rate of the gradient (Lypunov exponent), and $\zeta = \beta - \varphi - \frac{1}{4}\pi$

Using the result of 2.13, the rate of change of the angle ζ is model by the ODE ,

$$\frac{d\zeta}{dt} = -2\lambda\cos\zeta + \omega - 2\frac{d\varphi}{dt} \tag{2.15}$$

The stochastic differential equation system outlined next aims to model Λ and ζ.

2.4 Stochastic Differential Equations Model

In this setting we denote the angle $\zeta =$ X and the $\lambda = \mu$, so rewritting 2.15 as the ODE

$$\frac{dX}{dt} = -2\mu\cos X + \omega - 2\frac{d\varphi}{dt} \tag{2.16}$$

The terms ω and μ are modelled as two stochastic differential eqautions, decomposed into mean and random components.

$$\begin{aligned}\mu &= \mu_0 + Y(t)\\ \frac{\omega}{2} - \frac{d\varphi}{dt} &= Z(t)\end{aligned}$$

Substituting these values into 2.16 yields the SDE for the angle X

$$\frac{1}{2}\frac{dX}{dt} = -\mu_0 \cos X + \omega - Y(t)\cos X + Z(t) \tag{2.17}$$

As outlined in [9], the random parts of $Y(t)$ and $Z(t)$ are modelled as Ornstein-Uhlenbeck processes, with mean zero , time decay τ_Y and τ_Z, and strengths DY and DZ This yields the system of stochastic differential equations to model the angle X. An Ornstein-Uhlenbeck process is a stochastic process that is a Gaussian process, a Markov process, and is temporally homogeneous.

$$\begin{aligned}\frac{1}{2}\frac{dX}{dt} &= -\mu_0 \cos X + \omega - Y(t)\cos X + Z(t)\\ \frac{dY}{dt} &= -\frac{Y}{\tau_Y} + \sqrt{\frac{2\sigma_Y^2}{\tau_Y}}dW_Y\\ \frac{dZ}{dt} &= -\frac{Z}{\tau_Z} + \sqrt{\frac{2\sigma_Z^2}{\tau_Z}}dW_Z + \sqrt{\frac{2k^2}{\tau_Z}}dW_Y\end{aligned} \tag{2.18}$$

Where dW_Y and dW_Z are uncorrelated weiner processes with $\langle dW_{Y,Z}\rangle = 0$ and $\langle dW_{Y,Z}^2\rangle = dt$, and cross-corrlation -$k \neq 0$.

Since the solution of an SDE is a markov process, obtaining the probability distribution of the underlying equilbrium solution requires many simulations of some numerical method and averaging over these. This approach is outlined in the results section, but this direct numerical solution of the SDE's model converges very slowly.

The SDE model can be solved for a single sample path using the below scheme.

$$\begin{aligned}y_t &= y_{t-1} - \frac{y_{t-1}}{\tau_Y}\Delta t + \frac{D_Y}{\tau_y}W_{Y_t}\\ z_t &= z_{t-1} - \frac{z_{t-1}}{\tau_Z}\Delta t + \frac{\sqrt{D_Z(1-k^2)}}{\tau_Z}W_{Z_t}\\ x_t &= x_{t-1} + \frac{\Delta t}{\gamma}\left[\omega + \left(-\cos x_{t-1} + k\left(\frac{D_Z}{D_Y}\right)^{1/2}\right)y_{t-1} + z_{t-1}\right]\end{aligned} \tag{2.19}$$

2.4.1 Fokker-Planck

Theory tells us that every stochastic differential equation has a corresponding partial differential equation that has a solution which is the probability density function of the stochastic differential equation.

The Fokker-Planck equation describes the evolution of conditional probability density for given initial states for a Markov process, since the random part of the SDE model is based on an Ornstein-Uhlenbeck process we should be able to define a Fokker-Planck for the SDE model here

There are two ways of dealing with the random terms in SDE's is the terms, the Itô and Stratonovich interpretations. For multiplicative non-constant random terms each interpretations can yield different results.

The Ito interpretation requires the use of the Itô calculus. Stratonovichs interpretation is based on the limit of the random terms as the correlation time limits to zero, and it allows the use of the ordinary rules of calculus

Following the derivation outlined in [10] and [5], the Stratonovich method is used to define the n-dimensional the Fokker-Planck equation as

$$\frac{\partial P(\mathbf{X},t)}{\partial t} = \sum_{i=1}^{N} -\frac{\partial}{\partial X_i}(b(\mathbf{X})P(\mathbf{X},t)) - \sum_{i=1}^{N}\sum_{j=1}^{N} \frac{\partial}{\partial X_i \partial X_j}(\sigma(\mathbf{X})P(\mathbf{X},t)) \tag{2.20}$$

where $\mathbf{X} = (X_1, X_2, ..., X_N)$, $D^1(\mathbf{X}) = b(\mathbf{X})$ and $D^2(\mathbf{X}) = \sigma(\mathbf{X})$ and coefficients defined as

$$D^n(\mathbf{Z}) = \frac{1}{n!}\frac{1}{\Delta t}\int_{-\infty}^{\infty} (\mathbf{Y} - \mathbf{Z})^n P(\mathbf{Y}, \delta t|\mathbf{Z}) d\mathbf{Y} \tag{2.21}$$

Using this result and equating the coefficients of the 2.19, yields the required Fokker-Planck equation

$$\frac{\partial P}{\partial t} = \mathcal{L}_{OU} P - \frac{\partial}{\partial x}(VP) \tag{2.22}$$

where P is the probability density function of the triple (X, Y, Z) and

$$V(x, y, z) = 2(\omega - y\cos x + z)$$
$$\mathcal{L}_{OU} = \frac{1}{\tau_Y}\frac{\partial}{\partial y}(y\circ) + \frac{1}{\tau_Y^2}\frac{\partial^2}{\partial y^2} + \frac{1}{\tau_Z}\frac{\partial}{\partial z}(z\circ) + \frac{\rho}{\tau_Z^2}\frac{\partial^2}{\partial^2 z} + \frac{2c\rho^{1/2}}{\tau_Y \tau_Z}\frac{\partial^2}{\partial y \partial z}$$

where $c = \sqrt{\frac{k^2 \tau_Z}{D_Z}}$, $\rho = \frac{D_Z}{D_Y}$

By solving 2.22 for the equalibrium distribution of $P(x, y, z)$, its follows that the PDF of the X-angle can be computed by finding the marginal distribution of X

$$P_X(x) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} P(x,y,z)dYdZ \tag{2.23}$$

Similarly the PDF of the lypunov exponent is computed via a coordinate transformation on the joint marginal of P_{XY}

$$P_\Lambda(\lambda) = \int_{-\pi}^{\pi} P_{XY}\left(x, \frac{\lambda}{-2\sin x}\right)\frac{1}{2|\sin x|}dX \tag{2.24}$$

Fokker-Planck numerical scheme

The Fokker-Planck is solved using the same pseudo-spectral approach as was done for the vorticity and tracer concentration. One other aspect that must be addressed while solving the Fokker-Planck PDF is the *CFL* condition. The general CFL condition for the n-dimensional case is defined as

$$C = \Delta t \sum_{i=1}^{n} \frac{u_{x_i}}{\Delta x_i} \leq C_{\max}.$$

In this setting

$$C = \Delta t \min\{\Delta x, \Delta y, \Delta z\} = 0.1V_{max}$$

where $V_{max} = \left[\omega + (L_y/2)\left(1 + k\delta^{1/2}\right) + (L_z/2)\right]/\gamma$.

This is a necessary condition for convergence when numerically solving certain classes of PDE's which includes the Fokker-Planck equation.

2.5 Using Uncertainty Quantification methods to fit Fokker-Planck parameters

In [9], the values of the parameters in 2.22 are estimated using the moments of the ω and θ, thus the model of the stochastic orientation dynamics are shown as $D_Y = 0.05/\tau$, $D_Z = 0.9/\tau$, $k = 0$, $w = 0$ and $\tau = 0.1\sqrt{\langle ||\omega||_2^2\rangle}$

Uncertainty Quantification (UQ) methods try to encapsulate all the error and uncertainty in a models parameters so that the output can be evaluated and interpreted constrained by the inherent uncertainty.

Due to time constraints applying UQ methods to 2.22 was not attempted. A few Uncertainty Quantification methods as outlined in [3] are breifly summaried, that could be applied to to quantify the parameters.

2.5.1 Monte Carlo

In this method the SDE is solved directly by randomly sampling from some distribution that the unknown parameters are assumed to come from, this makes the SDE deterministic and a solution can be found. This proceess is repeated N times and average statistics can be computed. But this has very poor convergence. The Euler-Maruyama described in the result section is an example of this, but the parameters D_Y, D_Z, k and w are not treated as unknown parameters drawn from some distribution.

2.5.2 Stochastic Collocation Method

This similar to the vanilla Monte Carlo method except that random space is represneted by fewer points , each with a corresponding weight, which are used to calculate the mean and avergae from N runs of the method. This has exponential convergence.

2.5.3 Stochastic Galerkin Method

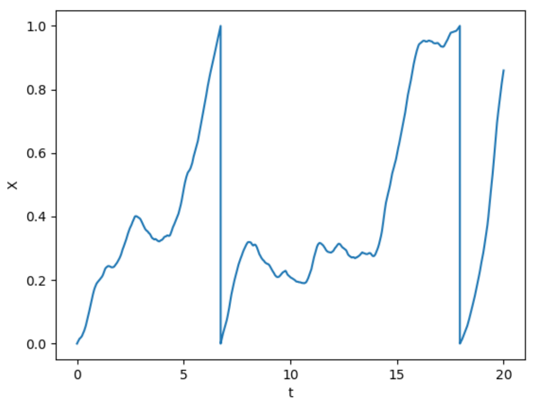
In this setting the solution is represented by a *Polynomial Chaos Expansion.* The type of polynomial choosen depends on the assumed distribution of the unknown parameter,e.g for Gaussian random variables , Hermite polynomials are used. This method also has exponential convergence.

Chapter 3

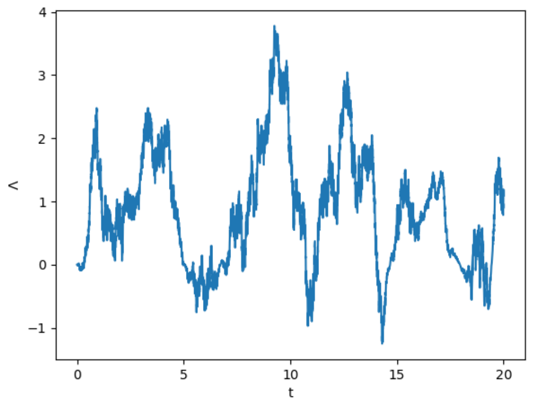
Results

3.1 Euler Maruyama

Most SDEs do not have closed form solutions, but solutions may be computed numerically. The Euler-Maruyama method is a way to create approximate sample paths. This method is outlined in 2.19. A *Python* implementation of this numerical scheme is listed in B.1



(a)



(b)

Figure 3.1: (a) Time series of the angle X mod 2π, (b) Time series of Lyapunov exponent

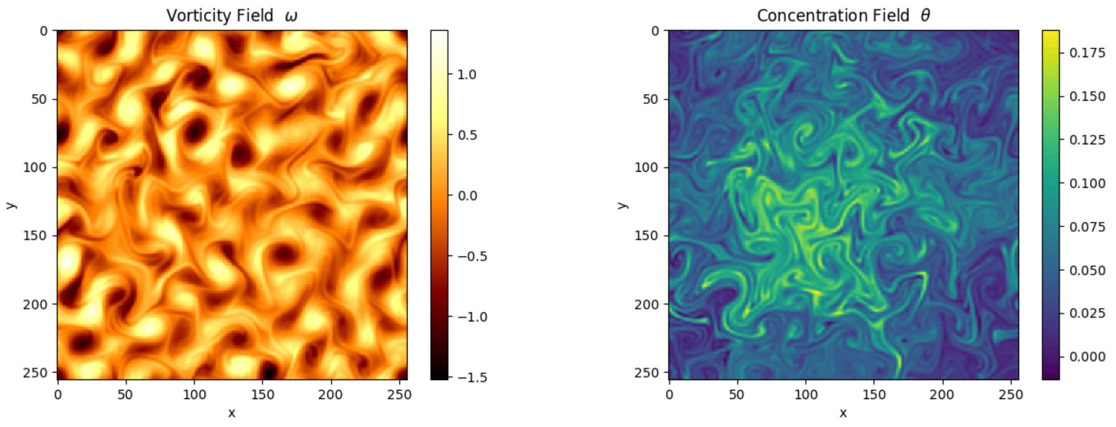
The implemented Euler-Maruyama method with parameters $\omega = 0.5$, $k = 0.5$, $\tau = 0.5$, $D_Y = D_Z = 2$, $\gamma = 0.5$, $\Delta t = 10^{-6}$, the results of which are shown in Figure 3.1 The plot of the normalized X-angle, 3.1 (a), shows that the X-angle movement to $\pm\pi/2$ as the simulation progresses. The plot of the lyapunov exponent, 3.1 (b) shows it is positive for the majority of the simulation, a positive lyapunov exponent indicates chaotic flow.

This method could be used to discover the stationary probability density functions for Λ and the angle X, by averaging sample paths over many simulations, but this approach converges very slowly. Instead the corresponding Fokker-planck equation is more computationally efficient.

3.2 Direct Numerical Simulation

3.2.1 DNS vorticity and advection

The direct numerical simulation instantaneous vorticity and the advection of the tracer at time $T = 250$ are shown in 3.2, this was based on *matlab* code developed for [9], and listed in B.3



(a) (b)

Figure 3.2: (a) The vorticity field ω, (b) The concentration field θ

3.2.2 DNS statistically steady

To extract an accurate distribution for X-angle and the lyapunov exponent λ from the DNS, the simulation is run for a long time until it has reached a statistically steady state. At each $T_i (i \in [1, 250])$, ω and θ are saved to a file, these are used to create a graphic to visually inspect for the system to have reached a steady state. Figure 3.3 shows the L^2 norm of the vorticity at each T_i,

From 3.3 we can infer that the flow has reached a statistically homogenous state after T_{20}. The data in the files saved from T_{20} to T_{250} are used to estimate the PDF of the X-angle and theLyaounov exponent from the DNS.

3.2.3 DNS probability density functions, X and Λ

The code listed in B.4 estimates the PDF of the X-angle and the Lyaounov exponent from the DNS. Figure 3.4 shows the estimated PDFs of the X-angle and Lyapunov exponent extracted frin the DNS solution to vorticity and advection equation.

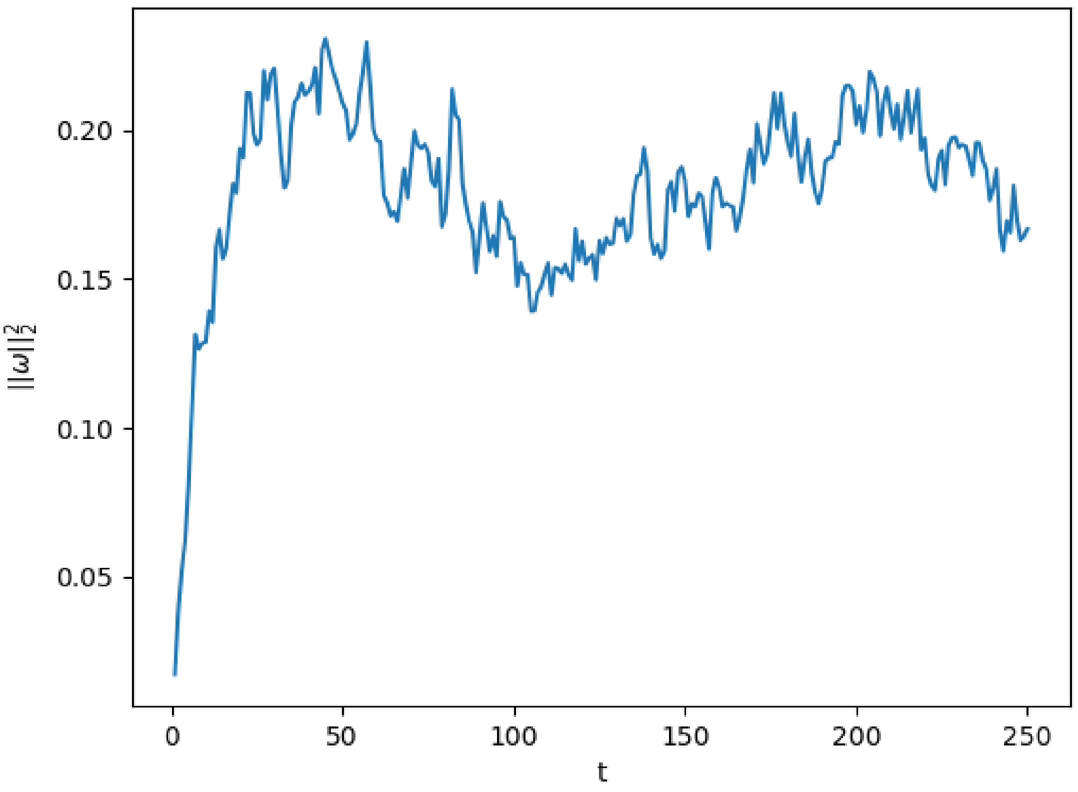
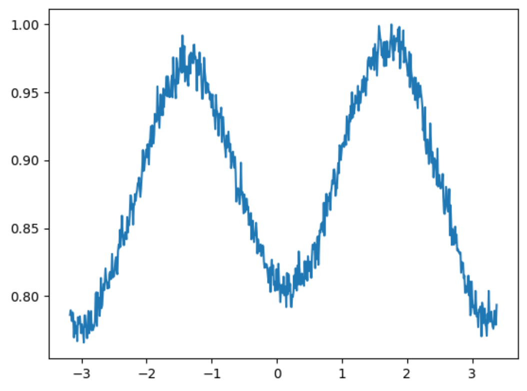
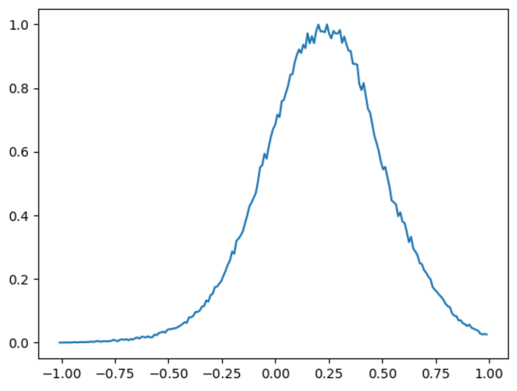


Figure 3.3: L^2 norm of ω vs. simulation time



(a)

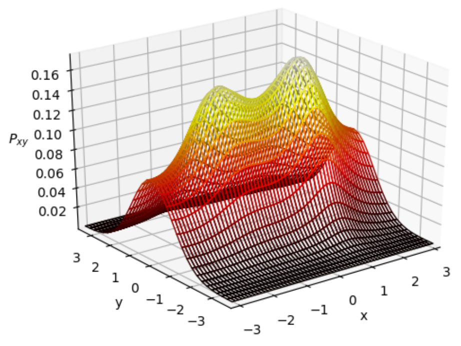


(b)

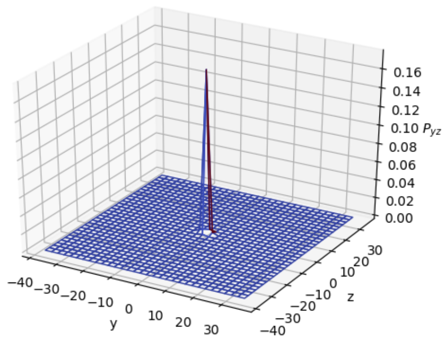
Figure 3.4: (a) The probability density function of the angle X, (b)The probability density function of the Lyaounov exponent extracted from the 2-d turbulence simulation

3.3 Fokker-Planck numerical solution

Equation (2.22) is solved numerically for the stationary distribution. The marginal distributions of P_{XY} and P_{YZ} are shown in Figure 3.5. The distribution of P_{YZ} is Gaussian, which matches up with the numerical implemetation.



(a)



(b)

Figure 3.5: The marginal probability distribution got from the stationary solution to the Fokker-Planck equation, (a) joint marginal $x - y$, (b) joint marginal $Y - Z$

The more import result is the marginal distribution of X , define in Equation (2.23) and the marginal distribution of Λ defined in Equation (2.24), shown in Figure (3.6)

The distribution of the X angles are whien in Figure. 3.6(a). This has two maxima close to $X = \pm\pi/2$. They do not line up exactly at $\pm\pi/2$ due to the correlcation $k = 0.5$ between the random terms and the drift $\omega = 0.5$. Setting $k = \omega = 0$ the distribution of the X angle aligns with $\pm\pi/2$. From this result we can conclude most probable alignment

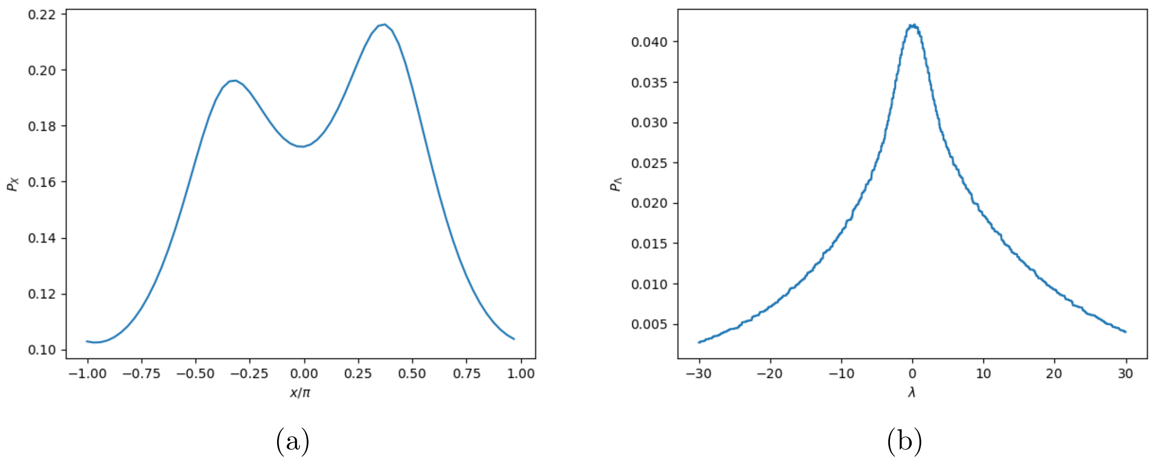


Figure 3.6: The distribution of the X-angles from Fokker-Planck (a) and The distribution of the Lyaounov exponents from Fokker-Planck (b)

of the X-angle is close to $\pi/2$. This reveals the most likely direction the vector $\mathbf{X}_{(+)}$ (the positive eigenvector of the rate-of-strain matrix $\mathcal{S}$) will point.

Figure. 3.6(b) shows the distribution of the Lyapunov exponent. THis distribution is slightly asymmetric to the right with the first moment of the distribution positive.

From these results we can conclude the most likely direction of the stretching direction is close to $\pi/2$ with a positive Lyaounov exponent.

Chapter 4

Discussion

A model was derived for the orientation dynamics of a fluid under idealised conditions with a specfically chosen forcing term. The model described is a coupled system of stochastic differential equations for the angle X, which is the angle between the rate of strain tensor's positive eigenvector and the x-axis. The associated Fokker-Planck equation is dervied and the probability density function of the angle X is found via numerical simulation , from this the probability density function of the Lyapunov exponent Λ is computed.

The distributions (for X and Λ) from the Fokker-Planck are checked against numerically simulation data from the vorticity and advection-diffusion equations. The vorticity and advection-diffusion equations are solved via a pseudo-spectral method. From extracting snapshots of vorticity (ω) and concentration (θ) at fixed time intervals, the distributions of the angle X and the Lyapunov exponent From this, a comparison of the distributuons from the Fokker-Planck solution versus the numerical simulation of forced turbulence , shows that the stochastic model is a good fit for the observed experimental data. A number of methods from Uncertainty Quantification are listed that could be employeed to fit the parameters of the Fokker-Planck more accurately, but time constrainst limited implementation of these methods.

The libraries documentation and examples were simple algebraic equations and time did not permit figuring how how to encode the FP equation within the constraints of these frameworks, A number of open source libraries were investigated to see if suitable for to fit the numerically simulationed data aginst the Fokker-Planck partial differential equation. UQ-PyL (Uncertainty Quantification Python Laboratory) http://www.uq-pyl.com/ OpenTURNS (Treatment of Uncertainties, Risks'N Statistics) http://www.openturns.org/ chaospy https://github.com/jonathf/chaospy unfortunately it was not possible to encode the complexity of the Fokker-Planck equation within these framework, given more time this could be investigated further.

The distribution of the Lyapunov exponent is found to be modelled accurately by the stochastic model.

References

[1] Mixing (process engineering).

[2] Fluid mixing equipment. *Industrial & Engineering Chemistry*, 49(3):42A–52A, 1957.

[3] Study in modern uncertainty quantification methods. 2012.

[4] John C. Bowman. How important is dealiasing for turbulence simulations?

[5] Scott Hottovy. The fokker-planck equation, 2011.

[6] Guillaume Lapeyre, P Klein, and B L. Hua. Does the tracer gradient vector align with the strain eigenvectors in 2d turbulence? 11:3729–3737, 12 1999.

[7] D.K. Lilly. Numerical simulation of two-dimensional turbulence, (1969). *The Physics of Fluids Supplement II*, pages 240–249, 1969.

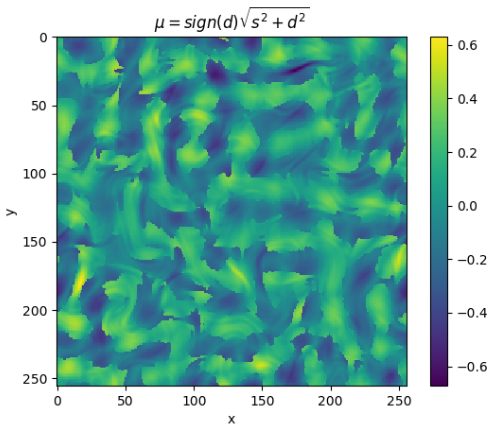
[8] Lennon Ó Náraigh. Stretching revisited.

[9] Lennon Ó Náraigh. Modelling the growth rate of a tracer gradient using stochastic differential equations. *European Journal of Mechanics B/Fluids*, 30(1):89–98, 2011.

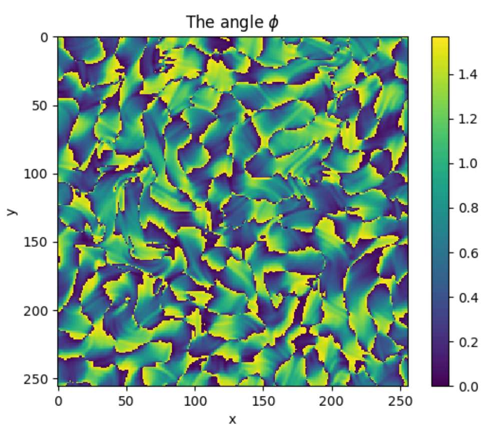
[10] H Risken. *The Fokker-Planck Equation: Methods of Solutions and Applications, 2nd ed.* Springer, 1989.

Appendix A

First Appendix

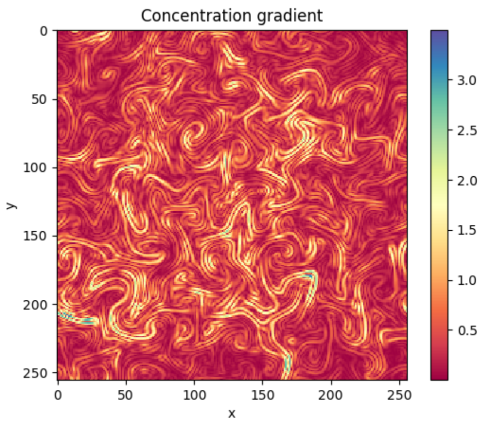


(a)

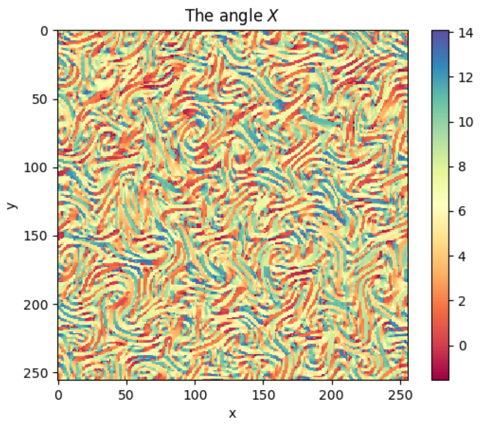


(b)

Figure A.1



(a)



(b)

Figure A.2

Appendix B

Code

B.1 Class SDEModelSolve

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

"""

Solve the SDE model directly using Euler-Maruyama algorithm.

"""
class SDEModelSolve:
    """
    Constructor.

    Args:
        t_periods   : mandatory integer
        rng_seed_1 : optional Y rng seed
        rng_seed_2 : optional Z rng seed

    Returns:
    """
    def __init__(self,t_periods, rng_seed_1=None, rng_seed_2=None):
        assert(t_periods != None or t_periods > 0)
        self.rng_seed_1 = rng_seed_1
        self.rng_seed_2 = rng_seed_2
        self.w = 0.5          #from page 11
        self.k = 0.5          #from page 11
        self.tau = 2          #from page 11
        self.Dy = 2           #from page 11
        self.Dz = 2           #from page 11
        self.dt=1e-6          #from page 11
        self.gamma=0.5        #from page 9
        self.T=self.tau*t_periods          #"integrate over T_periods
            time constants"
        self.strength=2       #"Weiner process of strength 2"
        self.ts = np.arange(0, self.T, self.dt)
        self.dX = np.zeros(self.ts.size)
        self.dY = np.zeros(self.ts.size)
        self.dZ = np.zeros(self.ts.size)

    """
    X SDE function definition.

    Args:
        x: value of x at time t ; x(t)
        y: value of y at time t ; y(t)
```

```
            z: value of z at time t ; z(t)

        Returns:
            float: value of x at time t+1 ; x(t+1)
        """
        def dX_sde(self,x,y,z):
            return ((self.w +(-np.cos(x) + self.k*np.sqrt(self.Dz/self.Dy
                ))*y + z)/self.gamma)*self.dt

        """
        Y SDE function definition.

        Args:
            y:      value of y at time t ; y(t)
            Wy:     wiener process value at time t; Wy(t)

        Returns:
            float: value of y at time t+1 ; y(t+1)
        """
        def dY_sde(self,y,Wy):
            return (-y/self.tau)*self.dt + (np.sqrt(self.Dy)/self.tau)*Wy

        """
        Z SDE function definition.

        Args:
            z:      value of z at time t ; z(t)
            Wz:     wiener process value at time t; Wz(t)

        Returns:
            float: value of z at time t+1 ; z(t+1)
        """
        def dZ_sde(self,z,Wz):
            return (-z/self.tau)*self.dt + (np.sqrt(self.Dz*(1-self.k**2)
                )/self.tau)*Wz

        """
        Solve SDE model and plot lambda and angle X.

        Args:

        Returns:
        """
        def solve_n_plot(self, save_plot_dir=None):
            np.random.seed(self.rng_seed_1)
            dWy=np.random.normal(loc = 0, scale = np.sqrt(self.dt),size=
                self.ts.size)
```

```python
        np.random.seed(self.rng_seed_2)
        dWz=np.random.normal(loc = 0, scale = np.sqrt(self.dt),size=self.ts.size)

        for i in range(1, self.ts.size):

            x_t_minus_1 = self.dX[i-1]
            y_t_minus_1 = self.dY[i-1]
            z_t_minus_1 = self.dZ[i-1]

            y_t = y_t_minus_1 + self.dY_sde(y_t_minus_1,dWy[i])
            z_t = z_t_minus_1 + self.dZ_sde(z_t_minus_1,dWz[i])
            x_t = x_t_minus_1 + self.dX_sde(x_t_minus_1,y_t_minus_1,z_t_minus_1)

            self.dX[i] = x_t
            self.dY[i] = y_t
            self.dZ[i] = z_t

        dX_scaled_mod_2pi = np.array([ x % ((-1 if x < 0 else 1)*2*np.pi) for x in self.dX])/(2*np.pi)
        #plot angle X
        plt.figure(1)
        plt.plot(self.ts, dX_scaled_mod_2pi)
        plt.ylabel('X')
        plt.xlabel('t')
        plt.xticks(np.arange(0, self.T+1, step=5))
        plt.show()
        if save_plot_dir != None:
            plt.savefig(save_plot_dir + '/x-angle.png')


        lambda_est = -2*np.array(self.dY)*np.sin(self.dX)
        #plot lamdba
        plt.figure(2)
        plt.plot(self.ts, lambda_est)
        plt.ylabel(r'$\Lambda$')
        plt.xlabel('t')
        plt.xticks(np.arange(0, self.T+1, step=5))
        plt.show()
        if save_plot_dir != None:
            plt.savefig(save_plot_dir + '/lambda.png')
```

B.2 Class FokkerPlankSolve

```
import numpy as np

class FokkerPlank:
    def __init__(self,Tf,opt_parm_dict):
        self.solved=False
        self.Tf = Tf
        self.gamma_val = float(opt_parm_dict.get('gamma',0.5))
        self.k_corr = float(opt_parm_dict.get('k_corr',0.5))
        self.tau=float(opt_parm_dict.get('tau',0.01))
        self.tauy=float(opt_parm_dict.get('tauy',self.tau))
        self.tauz=float(opt_parm_dict.get('tauz',self.tau))
        self.w = float(opt_parm_dict.get('w',0.5))
        #self.k = float(opt_parm_dict.get('k',0.5))
        self.Dy =float(opt_parm_dict.get('Dy',1))
        self.Dz =float(opt_parm_dict.get('Dz',1))
        self.delta=float(opt_parm_dict.get('delta',self.Dz/self.Dy))
        self.Lx=2*np.pi
        self.Lz=75
        self.Ly=self.Lz
        self.Nx=64
        self.Ny=32
        self.Nz=32
        self.dx=self.Lx/float(self.Nx)
        self.dy=self.Ly/float(self.Ny)
        self.dz=self.Lz/float(self.Nz)
        self.Vmax=2*(self.Lz/2)/self.gamma_val
        self.dx_min=min([self.dx,self.dy,self.dz])
        self.dt=0.1*self.dx_min/self.Vmax
        self.im=1j
        self.t_vec=np.arange(0,self.Tf,self.dt)
        self.n_timesteps=len(self.t_vec)
        self.ck=0.95
        self.x=np.arange(-self.Lx/float(2),(self.Lx/float(2)),self.dx
            )
        self.y=np.arange(-self.Ly/float(2),(self.Ly/float(2)),self.dy
            )
        self.z=np.arange(-self.Lz/float(2),(self.Lz/float(2)),self.dz
            )
        self.w_y=2*self.Dy/self.tauy
        self.w_z=2*self.Dz*(1-self.k_corr*self.k_corr)/self.tauz
        self.p0=np.zeros((self.Nx,self.Ny,self.Nz))
        self.lambda_range=np.arange(-30,30,0.01)
        for i in range(0,self.Nx):
            for j in range(0,self.Ny):
                for k in range(0,self.Nz):
                    self.p0[i,j,k] = (1/float(self.Lx)) * (1-0.5*np.
                        cos(2*self.x[i])) * np.exp(-(self.y[j]**2)/
                        self.w_y) * np.exp(-(self.z[k]**2)/self.w_z)
        self.nrm=np.sum(self.p0)*self.dx*self.dy*self.dz
        self.p0=self.p0/self.nrm

        self.kx=(([i+1 for i in range(0,self.Nx)] -np.ceil(self.Nx
            /2+1)) % self.Nx)-np.floor(self.Nx/2)
        self.ky=(([i+1 for i in range(0,self.Ny)] -np.ceil(self.Ny
            /2+1)) % self.Ny)-np.floor(self.Ny/2)
        self.kz=(([i+1 for i in range(0,self.Nz)] -np.ceil(self.Nz
            /2+1)) % self.Nz)-np.floor(self.Nz/2)

        self.Kx=np.zeros(shape=(self.Nx,self.Ny,self.Nz))
        self.Ky=np.zeros(shape=(self.Nx,self.Ny,self.Nz))
        self.Kz=np.zeros(shape=(self.Nx,self.Ny,self.Nz))

        for i in range(0,self.Nx):
            for j in range(0,self.Ny):
                for k in range(0,self.Nz):
                    self.Kx[i,j,k]=self.kx[i]
                    self.Ky[i,j,k]=self.ky[j]
                    self.Kz[i,j,k]=self.kz[k]

        self.Y=np.zeros(shape=(self.Nx,self.Ny,self.Nz))
        self.Z=np.zeros(shape=(self.Nx,self.Ny,self.Nz))

        for i in range(0,self.Nx):
            for j in range(0,self.Ny):
                for k in range(0,self.Nz):
                    self.Y[i,j,k]=self.y[j]
                    self.Z[i,j,k]=self.z[k]

        self.Kx=self.im*((2*np.pi/float(self.Lx))*self.Kx)
        self.Ky=self.im*((2*np.pi/float(self.Ly))*self.Ky)
        self.Kz=self.im*((2*np.pi/float(self.Lz))*self.Kz)

        small=0
        self.ksquare_lap=small*(self.Kx**2)+(self.Dy/(self.tauy**2))
            *(self.Ky**2)+(self.Dz*(1-self.k_corr*self.k_corr)/(self.
            tauz**2))*(self.Kz**2)

        self.Vx=np.zeros(shape=(self.Nx,self.Ny,self.Nz))

        for i in range(0,self.Nx):
            for j in range(0,self.Ny):
                for k in range(0,self.Nz):
                    self.x_val=self.x[i]
                    self.y_val=self.y[j]
                    self.z_val=self.z[k]
                    prefac=1
                    self.Vx[i,j,k]=prefac*((self.w/self.gamma_val)+(1/self.gamma_val)*(-np.cos(self.x_val)+self.k_corr*np.sqrt(self.delta))*self.y_val+(self.z_val/self.gamma_val))

        self.n_subdiv = 50

        self.residual_vec=np.zeros(self.n_timesteps)
        self.av_vec=self.residual_vec

        self.p_hat_old=np.fft.fftn(self.p0)
        self.Vp_hat=np.fft.fftn(self.Vx*self.p0)
        self.py_hat=np.fft.fftn(self.p0*self.Y)
        self.pz_hat=np.fft.fftn(self.p0*self.Z)
        self.dy_py_hat= self.Ky*self.py_hat
        self.dz_pz_hat= self.Kz*self.pz_hat

        self.src0_hat=-self.Kx*self.Vp_hat
        self.src1_hat=(1/self.tauy)*self.dy_py_hat+(1/self.tauz)*self.dz_pz_hat
        self.src_hat=self.src0_hat+self.src1_hat

        self.p_hat=((1+(1-self.ck)*self.dt*self.ksquare_lap)/(1-self.ck*self.dt*self.ksquare_lap))*self.p_hat_old+self.dt*(self.src_hat/(1-self.ck*self.dt*self.ksquare_lap))
        self.p=np.real(np.fft.ifftn(self.p_hat))

        self.residual_val=(abs(self.p-self.p0)).max()
        self.residual_vec[0]=self.residual_val

        self.p_x=0*self.x
        for i in range(0,len(self.x)):
            self.p_x[i]=np.sum(self.p[i,:,:])

        self.x1=np.concatenate([self.x, [np.pi]])
        self.p1=np.concatenate([self.p_x,[self.p_x[0]]])
        self.av_vec[0]=np.sum(self.p1*self.x1)/np.sum(self.p1)

    def __compute_marginal_distributions(self):
        self.p_x=np.zeros(len(self.x))
        self.p_y=np.zeros(len(self.y))
        self.p_z=np.zeros(len(self.z))

```

```python
        for i in range(0,len(self.x)):
            self.p_x[i]=np.sum(self.p_joint[i,:,:])*self.dy*self.dz

        for j in range(0,len(self.y)):
            self.p_y[j]=np.sum(self.p_joint[:,j,:])*self.dx*self.dz

        for k in range(0,len(self.z)):
            self.p_z[k]=np.sum(self.p_joint[:,:,k])*self.dy*self.dz

        self.p_xy=np.zeros((len(self.x),len(self.y)))
        for i in range(0,len(self.x)):
            for j in range(0,len(self.y)):
                self.p_xy[i,j]=np.sum(self.p_joint[i,j,:])*self.dz

        self.p_yz=np.zeros((len(self.y),len(self.z)))
        for j in range(0,len(self.y)):
            for k in range(0,len(self.z)):
                self.p_yz[j,k]=np.sum(self.p_joint[:,j,k])*self.dx

        nrm=np.sum(self.p_yz)*self.dy*self.dz
        self.p_yz=self.p_yz/nrm

        self.p_yz_anal=np.zeros((len(self.y),len(self.z)))

        w_y=2*self.Dy/self.tauy
        w_z=2*self.Dz*(1-self.k_corr*self.k_corr)/self.tauz

        for j in range(0,len(self.y)):
            for k in range(0,len(self.z)):
                self.p_yz_anal[j,k]=np.exp(-self.y[j]**2/w_y)*np.exp
                    (-self.z[k]**2/self.w_z)

        nrm=np.sum(self.p_yz_anal)*self.dy*self.dz
        self.p_yz_anal=self.p_yz_anal/nrm

    def __do_solve_iter(self, idx):
        Vp_hat=np.fft.fftn(self.Vx*self.p)
        py_hat=np.fft.fftn(self.p*self.Y)
        pz_hat=np.fft.fftn(self.p*self.Z)
        dy_py_hat= self.Ky*py_hat
        dz_pz_hat= self.Kz*pz_hat

        self.src0_hat=-self.Kx*Vp_hat
        self.src1_hat=(1/self.tauy)*dy_py_hat+(1/self.tauz)*dz_pz_hat

        self.src_hat=self.src0_hat+self.src1_hat

        self.p_hat_new=(1/(1-2*self.dt*self.ksquare_lap))*self.p_hat_old+2*self.dt*(self.src_hat/(1-2*self.dt*self.ksquare_lap))

        self.p_hat_old=self.p_hat
        self.p_old=np.real(np.fft.ifftn(self.p_hat_old))

        self.p_hat=self.p_hat_new
        self.p=np.real(np.fft.ifftn(self.p_hat))

        self.residual_val=(abs(self.p_old-self.p)).max()
        self.residual_vec[idx]=self.residual_val
        for i in range(1,len(self.x)):
            self.p_x[i]=np.sum(self.p[i,:,:])

        self.x1=np.concatenate([self.x, [np.pi]])
        self.p1=np.concatenate([self.p_x,[self.p_x[0]]])
        self.av_vec[idx]=np.sum(self.p1*self.x1)/np.sum(self.p1)
        return None

    def solve(self):
        arr_len = len(self.t_vec)
        if not self.solved:
            self.solved = True
            for t_ctr in range(1,arr_len):
                if t_ctr % 100 == 0:
                    print "% complete " + str(100*t_ctr/float(arr_len))
                self.__do_solve_iter(t_ctr)
            nrm=np.sum(self.p)*self.dx*self.dy*self.dz
            self.p_joint=self.p/nrm
            self.__compute_marginal_distributions()
            self.__p_lambda()
        else:
            print 'Fokker-Plank already solved for given parameters, use methods '

    def p(self):
        return self.p_joint

    def p_x(self):
        return self.p_x

    def p_y(self):
        return self.p_y

    def p_z(self):
        return self.p_z

    def p_xy(self):
        return self.p_xy

    def p_yz(self):
        return self.p_yz

    def __p_lambda(self):

        self.p_lambda=0*self.lambda_range
        for i in range(0,len(self.lambda_range)):
            sum_val=0
            lambda_val=self.lambda_range[i]
            if lambda_val==0:
                sum_val=0
            else:
                for ii in range(0,len(self.x)):
                    g_val=-2*np.sin(self.x[ii])

                    if((abs(lambda_val)/(abs(g_val)+1e-8))>=(self.Ly/2)):
                        summand=0
                    else:
                        ll_map=1+(1/self.dy)*((lambda_val/g_val)+(self.Ly/2))
                        ll_map=int(np.floor(ll_map)-1)

                        if(abs(ll_map)>self.Ny):
                            summand=0
                        else:
                            summand=self.p_xy[ii,ll_map]*(self.dx/abs(g_val))
                    sum_val=sum_val+summand
            self.p_lambda[i]=sum_val
        ix=np.argmin(abs(self.lambda_range))
        self.p_lambda[ix]=(self.p_lambda[ix-1] + self.p_lambda[ix+1])/2
        return None
```

B.3 Class VorticitySolve

```

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

class VorticitySolve:
    def __init__(self,T_final, T_init=None, omega_init_file=None,
       theta_init_file=None):
        self.T_final = T_final
        self.T_init = T_init if T_init != None else 0;
        self.Lx = 2*np.pi
        self.Ly = 2*np.pi
        self.nu_p = 5.9e-30
        self.nu_0 = 0.05
        self.p = 8
        self.dt = 1e-3
        self.im = 1j
        self.Nx = 256
        self.Ny = 256
        self.dx = self.Lx/float(self.Nx)
        self.dy = self.Ly/float(self.Ny)
        self.x = np.array([v*self.dx for v in range(0,self.Nx)])
        self.y = np.array([v*self.dx for v in range(0,self.Ny)])
        (self.X,self.Y) = np.meshgrid(self.x,self.y)
        self.N_timesteps = np.floor(self.T_final/self.dt)
        self.N_timesteps_init = np.floor(self.T_init/self.dt)
        self.kx_vec = [v*2*np.pi/self.Lx for v in range(-self.Nx/2,
           self.Nx/2)]
        self.ky_vec = [v*2*np.pi/self.Ly for v in range(-self.Ny/2,
           self.Ny/2)]
        self.Ksq = np.zeros((self.Nx,self.Ny))
        self.Kx = np.zeros((self.Nx,self.Ny))
        self.Ky = np.zeros((self.Nx,self.Ny))
        for i in range(0,self.Nx):
            self.kx_val = self.kx_vec[i]
            for j in range(0,self.Ny):
                self.ky_val = self.ky_vec[j]
                self.Ksq[i,j] = (self.kx_val**2)+(self.ky_val**2)
                self.Kx[i,j] = self.kx_val
                self.Ky[i,j] = self.ky_val
        self.Ksq_laplace = self.Ksq
        self.Ksq_laplace[(self.Nx/2)+1,(self.Ny/2)+1] = 1
        self.A0 = 1
        self.R = 0.9
        self.kmax = 9*(2*np.pi/self.Lx)
        self.kmin = 7*(2*np.pi/self.Lx)
        self.Mx_scale = np.zeros((self.Nx,self.Ny))
        for i in range(0,self.Nx):
            for j in range(0,self.Ny):
                k_radius = np.sqrt(self.kx_vec[i]**2+self.ky_vec[j
                   ]**2)
                if k_radius>self.kmin  and k_radius<self.kmax :
                    self.Mx_scale[i,j] = 1
        self.A0_num = self.A0*self.Nx*self.Ny/np.sqrt(np.pi*(self.
           kmax*self.kmax-self.kmin*self.kmin))*(np.sqrt(2)/1)
        self.omega0 = np.zeros((self.Nx,self.Ny))
        self.sigma = 0.1*self.Lx
        self.exponenttheta = ((self.X -self.Lx/2)**2 + (self.Y- self.
           Ly/2)**2)/(2*self.sigma**2)
        self.theta0 = np.exp(-self.exponenttheta)
        self.dealias = np.ones((self.Nx,self.Ny))
        self.abs_kx_max = self.kx_vec[self.Nx-1]
        self.abs_ky_max = self.ky_vec[self.Ny-1]
        for i in range(0,self.Nx):
            for j in range(0,self.Ny):
                abs_kx = abs(self.kx_vec[i])
                abs_ky = abs(self.ky_vec[j])
                if ((abs_kx>((2/float(3))*self.abs_kx_max)) and (
                   abs_ky>((2/float(3))*self.abs_ky_max)) ):
                    self.dealias[i,j] = 0
        self.omega = self.omega0
        self.theta = self.theta0

        if omega_init_file != None and theta_init_file != None:
            self.theta=np.loadtxt(theta_init_file,dtype='float',
               delimiter=',')
            self.omega=np.loadtxt(omega_init_file,dtype='float',
               delimiter=',')

        self.forcing_hat = np.zeros((self.Nx,self.Ny))
        self.omega_arr = [self.omega]

    def solve(self):
        print '
           ***************************************************************
           '
        print '**    Number of iterations to complete ' + str((self.
           N_timesteps+1)-self.N_timesteps_init)
        print '
           ***************************************************************
           '
        for t_ctr in range(int(self.N_timesteps_init)+1,int(self.
            N_timesteps)+1):

            omega_hat=np.fft.fftshift(np.fft.fft2(self.omega))
            theta_hat=np.fft.fftshift(np.fft.fft2(self.theta))

            psi_hat=np.zeros((self.Nx,self.Ny),'complex')
            for i in range(0,self.Nx):
                for j in range(0,self.Ny):
                    if abs(self.Ksq_laplace[i,j]) > 0:
                        psi_hat[i,j]=omega_hat[i,j]/complex(self.Ksq_laplace[i,j]) # make sure not to divide by zero here.

            dx_psi_hat=self.im*self.Kx*psi_hat
            dy_psi_hat=self.im*self.Ky*psi_hat
            dx_omega_hat=self.im*self.Kx*omega_hat
            dy_omega_hat=self.im*self.Ky*omega_hat
            dx_theta_hat=self.im*self.Kx*theta_hat
            dy_theta_hat=self.im*self.Ky*theta_hat

            dx_psi=np.real(np.fft.ifft2(np.fft.ifftshift(dx_psi_hat)))
            dy_psi=np.real(np.fft.ifft2(np.fft.ifftshift(dy_psi_hat)))
            dx_omega=np.real(np.fft.ifft2(np.fft.ifftshift(dx_omega_hat)))
            dy_omega=np.real(np.fft.ifft2(np.fft.ifftshift(dy_omega_hat)))
            dx_theta=np.real(np.fft.ifft2(np.fft.ifftshift(dx_theta_hat)))
            dy_theta=np.real(np.fft.ifft2(np.fft.ifftshift(dy_theta_hat)))

            u=dy_psi
            v=-dx_psi

            conv_theta     = u*dx_theta + v*dy_theta
            conv_theta_hat = np.fft.fftshift(np.fft.fft2(conv_theta))

            u_dot_grad_omega = u*dx_omega + v*dy_omega
            u_dot_grad_omega_hat=np.fft.fftshift(np.fft.fft2(u_dot_grad_omega))

            Mx_random=np.random.random((self.Nx,self.Ny))
            random_part_hat=self.Mx_scale*np.exp(2*np.pi*self.im*Mx_random)
            self.forcing_hat=np.sqrt(1-self.R*self.R)*self.A0_num*
```

```
                random_part_hat+self.R*self.forcing_hat

            #vorticity
            omega_hat_numerator=omega_hat*(1-0.5*self.dt*self.nu_p*(self.Ksq**self.p))-self.dt*u_dot_grad_omega_hat+self.dt*(self.forcing_hat-self.nu_0*omega_hat)
            omega_hat_denominator=1+0.5*self.dt*self.nu_p*(self.Ksq**self.p)
            omega_hat_new=self.dealias*(omega_hat_numerator/omega_hat_denominator)

            # invection / diffusion eqn
            numerator_theta = theta_hat - self.dt*conv_theta_hat;
            denominator_theta = (1 + self.dt*((-1)**self.p)*((-1)**self.p)*self.nu_p*((self.Ksq**self.p)));
            theta_hat_new = self.dealias*(numerator_theta/denominator_theta)

            self.theta=np.real(np.fft.ifft2(np.fft.ifftshift(theta_hat_new)))
            self.omega=np.real(np.fft.ifft2(np.fft.ifftshift(omega_hat_new)))
            print 'iter  ' + str(t_ctr) #+ ' ' + str(sum(self.omega)) + ' ' + str(self.theta[0,0])
            if t_ctr % 1000 == 0 :
                np.savetxt("/tmp/omega" + str(t_ctr) + ".csv", self.omega, delimiter=",",fmt='%1.64e')
                np.savetxt("/tmp/theta" + str(t_ctr) + ".csv", self.theta, delimiter=",",fmt='%1.64e')
        plt.imshow(self.omega)
        plt.imshow(self.theta)

    def save_data_frames(self):
        for i in range(0, len(self.omega_arr)):
            np.savetxt("/tmp/" + str(i) + "_omega.csv", self.omega_arr[i], delimiter=",")

    def view_animation(self,save=False):
        fig = plt.figure()

        # ims is a list of lists, each row is a list of artists to draw in the
        # current frame; here we are just animating one artist, the image, in
        # each frame
        ims = []
        for i in range(0,len(self.omega_arr)):
                im = plt.imshow(self.omega_arr[i], animated=True)
                ims.append([im])

        #create anaimation
        ani = animation.ArtistAnimation(fig, ims, interval=100, blit=
            True, repeat_delay=1000)

        #save animation to disk
        if save:
            ani.save('/tmp/Vorticity2D.mp4',bitrate=1000, dpi=100)

        plt.show()


#######################
#init
#######################

#vs = VorticitySolve(100)

#######################
#Solve
#######################
#vs.solve()

#######################
#Save data to filesystem
#######################
#vs.save_data_frames()

#######################
#View animation
#######################
#vs.view_animation(True)
```

B.4 Class VorticitySolvePostProcessing

```
import numpy as np

class VorticitySolvePostProcessing:

    @staticmethod
    def extract_model_parameters(w,theta):

        (nNx,nNy)=w.shape
        Nx=nNx
        Ny=nNy

        L=2*np.pi
        im=1j

        kx=[v % Nx for v in (range(1,Nx+1)-np.ceil(Nx/2+1) ) ] - np.
            floor(Nx/2)
        ky=[v % Ny for v in (range(1,Ny+1)-np.ceil(Ny/2+1) ) ] - np.
            floor(Ny/2)

        Kx=np.zeros((Nx,Ny))
        Ky=np.zeros((Nx,Ny))

        for i in range(0,Nx):
            Kx[:,i]=kx
            Ky[i,:]=ky

        Kx=(2*np.pi/L)*im*Kx
        Ky=(2*np.pi/L)*im*Ky

        ksquare_viscous=Kx**2+Ky**2          # Laplacian in Fourier
            space
        ksquare_poisson=ksquare_viscous
        ksquare_poisson[1,1]=1                  # fixed Laplacian in
            Fourier space for Poisson's equation

        w_hat=np.fft.fft2(w)
        theta_hat=np.fft.fft2(theta)

        psi_hat = 0*w_hat
        for i in range(0,Nx):
            for j in range(0,Ny):
                if abs(ksquare_poisson[i,j]) > 0:
                    psi_hat[i,j] = -w_hat[i,j]/complex(
                        ksquare_poisson[i,j]) # make sure not to
                        divide by zero here.


        s_hat=Kx*Ky*psi_hat
        d_hat=Ky*Ky*psi_hat-Kx*Kx*psi_hat
        d_hat=d_hat/2

        thetax_hat=Kx*theta_hat
        thetay_hat=Ky*theta_hat

        d=np.real(np.fft.ifft2(d_hat))
        s=np.real(np.fft.ifft2(s_hat))

        u = np.real(np.fft.ifft2( Ky*psi_hat))
        v = np.real(np.fft.ifft2(-Kx*psi_hat))

        theta_x=np.real(np.fft.ifft2(thetax_hat))
        theta_y=np.real(np.fft.ifft2(thetay_hat))

        grad=np.sqrt(theta_x**2+theta_y**2)

        phi=np.zeros((Nx,Ny))

        for i in range(0,Nx):
            for j in range(0,Ny):
                d_val=d[i,j]
                s_val=s[i,j]

                if d_val==0:
                   phi[i,j]=0
                else:
                    alpha_val=s_val/d_val
                    nrm=2*np.sqrt(alpha_val*alpha_val+1)*(-alpha_val+
                       np.sqrt(alpha_val*alpha_val+1))
                    if nrm < 1:
                        nrm = 1
                    nrm = np.sqrt(nrm)
                    phi[i,j] = np.arccos(1/nrm)

        beta_vec=np.zeros((Ny,Nx))

        for i in range(0,Nx):
            for j in range(0,Ny):

                nrm_theta=np.sqrt(theta_x[i,j]**2+theta_y[i,j]**2)

                if nrm_theta==0:
                    beta_vec[i,j]=0
                else:
                    cosbeta=theta_x[i,j]/nrm_theta
                    sinbeta=theta_y[i,j]/nrm_theta

                    if cosbeta >= 0:
                        if sinbeta>=0:
                            beta_val=np.arccos(cosbeta)
                        else:
                            beta_val=2*np.pi-np.arccos(cosbeta)
                    elif sinbeta >= 0:
                        beta_val=np.pi-np.arccos(abs(cosbeta))
                    else:
                        beta_val=np.pi+np.arccos(abs(cosbeta))

                    beta_vec[i,j]=beta_val

        psi_angle=(np.pi/4)-phi

        psi_angle_hat=np.fft.fft2(psi_angle)
        psi_angle_x=np.real(np.fft.ifft2(Kx*psi_angle_hat))
        psi_angle_y=np.real(np.fft.ifft2(Ky*psi_angle_hat))
        conv=u*psi_angle_x+v*psi_angle_y
        w_tot=(w/2)+conv

        mu=np.sign(d)*np.sqrt(d**2+s**2)

        sig1_w=np.sum(w_tot)/(Nx*Ny)
        sig2_w=np.sum(w_tot**2)/(Nx*Ny)

        sig1_l=np.sum(mu)/(Nx*Ny)
        sig2_l=np.sum(mu**2)/(Nx*Ny)

        sig_lw=np.sum((w_tot-sig1_w)*(mu-sig1_l))/(Nx*Ny)

        Xangle=2*(psi_angle+beta_vec)

        grad_av=np.sqrt(np.sum(grad**2)/(Nx*Ny))

        #initialise LAMBDA array
        LAMBDA=np.zeros(np.sum(grad > 3*grad_av))
        k=0
        for i in range(0,Nx):
            for j in range(0,Ny):
                if grad[i,j] > 3*grad_av:
                    LAMBDA[k] =-2*mu[i,j]*np.sin(Xangle[i,j])
                    k = k + 1

        LAMBDA=np.array(LAMBDA)
        prod=s*(theta_y**2-theta_x**2)-2*d*theta_x*theta_y
        prod=prod/(grad**2)

        return (u,v,d,s, phi,Xangle,w_tot,mu,sig1_w,sig2_w,sig1_l,
           sig2_l,sig_lw,grad,LAMBDA,prod,theta_x,theta_y)


    @staticmethod
    def histogram_avg(rootdir,nlow,nhigh,nres):

        hist_bin_width=0.01
        num_str=str(nlow)
        filename_theta ='theta' + num_str + '.csv'
        filename_omega ='omega' + num_str + '.csv'

        t=np.loadtxt(rootdir + filename_theta,dtype='float',
           delimiter=',')
        w=np.loadtxt(rootdir + filename_omega,dtype='float',
           delimiter=',')

        (u,v, d,s, phi,Xangle,w_tot,mu,sig1_w,sig2_w,sig1_l,sig2_l,
           sig_lw,grad,LAMBDA,prod,theta_x,theta_y)=
           VorticitySolvePostProcessing.extract_model_parameters(w,t)

        (yw, xw) = np.histogram(LAMBDA,np.arange(-1,1,hist_bin_width)
           )
        (Xyw, Xxw) = np.histogram(np.reshape(Xangle,Xangle.shape[0]*
           Xangle.shape[1],1),np.arange(-3,15,hist_bin_width))
        (Zyw, Zxw) = np.histogram(np.reshape(mu,mu.shape[0]*mu.shape
           [1],1),np.arange(-1,1,hist_bin_width))

        sum_vec_x=xw
        sum_vec_y=yw

        Xsum_vec_x=Xxw
        Xsum_vec_y=Xyw

        Zsum_vec_x=Zxw
        Zsum_vec_y=Zyw

        for k in np.arange(nlow,nhigh+1000,1000):
            print k
            filectr=str(k)
            num_str=str(filectr)
            filename_theta ='theta' + num_str + '.csv'
            filename_omega ='omega' + num_str + '.csv'
```

```python

            t=np.loadtxt(rootdir + filename_theta,dtype='float',delimiter=',')
            w=np.loadtxt(rootdir + filename_omega,dtype='float',delimiter=',')

            (u, v, d,s, phi,Xangle,w_tot,mu,sig1_w,sig2_w,sig1_l,sig2_l,sig_lw,grad,LAMBDA,prod,theta_x,theta_y)=VorticitySolvePostProcessing.extract_model_parameters(w,t)

            (yw, xw) = np.histogram(LAMBDA,np.arange(-1,1,hist_bin_width))
            (Xyw, Xxw) = np.histogram(np.reshape(Xangle,Xangle.shape[0]*Xangle.shape[1],1),np.arange(-3,15,hist_bin_width))
            (Zyw, Zxw) = np.histogram(np.reshape(mu,mu.shape[0]*mu.shape[1],1),np.arange(-1,1,hist_bin_width))

            sum_vec_x=sum_vec_x+xw
            sum_vec_y=sum_vec_y+yw

            Xsum_vec_x=Xsum_vec_x+Xxw
            Xsum_vec_y=Xsum_vec_y+Xyw

            Zsum_vec_x=Zsum_vec_x+Zxw
            Zsum_vec_y=Zsum_vec_y+Zyw

        xa=sum_vec_x/float((nhigh-nlow)/1000)
        ya=sum_vec_y/float((nhigh-nlow)/1000)

        Xxa=Xsum_vec_x/float((nhigh-nlow)/1000)
        Xya=Xsum_vec_y/float((nhigh-nlow)/1000)

        Zxa=Zsum_vec_x/float((nhigh-nlow)/1000)
        Zya=Zsum_vec_y/float((nhigh-nlow)/1000)

        return {'LAMBDA':(ya,xa),'Xangle':(Xya,Xxa),'mu':(Zya,Zxa)}


    @staticmethod
    def G(x, w):
        return np.exp(-x**2/(2*w*w))


    @staticmethod
    def gaussian_smoothing_loc(x,y,w):

        # Gaussian filter for a function on a periodic domain.

        x=x-2*np.pi
        L=x(len(x))-x[0]

        n=len(x)
        x_a=0*(range(0,3*n))
        y_a=x_a

        dx=abs(x[1]-x[0])

        for i in range(0,n):
            x_a[i]=x[i]-L
            y_a[i]=y[i]

        for i in range(0,n):
            x_a[i+n]=x[i]
            y_a[i+n]=y[i]

        for i in range(0,n):
            x_a[i+2*n]=x[i]+L
            y_a[i+2*n]=y[i]

        y_s=y_a

        for i in range(0,3*n):
            y_s[j]=sum(G(x_a-x_a[j],w)*y_a)/sum(G(x_a-x_a[j], w))

        x_ss=0*(range(0,n))
        y_ss=x_ss

        for i in range(0,n):
            x_ss[i] =x_a[i+n]
            y_ss[i] =y_s[i+n]

        for i in range(0,np.floor(n/2)):
            x_ss[i+np.floor(n/2)] = -x_ss[i]
            y_ss[i+np.floor(n/2)] = y_ss[i]

        (x_ss1,ix)=np.sort(x_ss)

        for i in range(0,len(x_ss1)):
            y_ss1[i]=y_ss[ix[i]]

        x_ss=x_ss1
        y_ss=y_ss1

        nrm=sum(y_ss)*dx
        y_ss=y_ss/nrm

        return (x_ss,y_ss,x_a,y_a)
```

B.5 Run simulations and plot generations commands

```python
import imp
import numpy as np
import matplotlib.pyplot as plt

code_dir ='/home/ian/Desktop/MSc-Proj/code/thesis_code/'
data_dir ='/home/ian/Desktop/MSc-Proj/saved-datasets/'
img_save_dir = '/home/ian/Desktop/MSc-Proj/thesistemplate/imgs/'

"""

Simulations and plot executions of direct solve of SDE model

"""
sde_solver = imp.load_source('SDEModelSolve', code_dir +'
    SDEModelSolve.py')
sde_solver.SDEModelSolve(10, 0, 2**32-1).solve_n_plot(img_save_dir)


"""

Vorticity Solve post processing

"""
theta=np.loadtxt(data_dir + 'theta250000.csv',dtype='float',
    delimiter=',')
w=np.loadtxt(data_dir + 'omega250000.csv',dtype='float', delimiter=',
    ')
dns_solver = imp.load_source('VorticitySolvePostProcessing', code_dir
     +'VorticitySolvePostProcessing.py')
(u,v, d,s, phi,Xangle,w_tot,lambda_unsigned,sig1_w,sig2_w,sig1_l,
    sig2_l,sig_lw,grad,ll,prod,theta_x,theta_y)=dns_solver.
    VorticitySolvePostProcessing.extract_model_parameters(w,theta)


"""

Plots

"""

plt.figure(1)
plt.imshow(w, cmap="afmhot")
plt.title("Vorticity Field  " + r'$\omega$')
plt.ylabel('y')
plt.xlabel('x')
plt.colorbar()
plt.savefig(img_save_dir+'vorticity-field-dns.png')

plt.figure(2)
plt.imshow(theta)
plt.title("Concentration Field  " + r'$\theta$')
plt.ylabel('y')
plt.xlabel('x')
plt.colorbar()
plt.savefig(img_save_dir+'concentration-field-dns.png')

plt.figure(3)
plt.imshow(lambda_unsigned)
plt.title(r'$\mu = sign(d)\sqrt{s^{2} + d^{2}}$')
plt.ylabel('y')
plt.xlabel('x')
plt.colorbar()
plt.savefig(img_save_dir+'mu-field-dns.png')

plt.figure(4)
plt.imshow(phi)
plt.title("The angle " + r'$\phi$')
plt.ylabel('y')
plt.xlabel('x')
plt.colorbar()
plt.savefig(img_save_dir+'phi-angle-dns.png')

plt.figure(5)
plt.imshow(Xangle, cmap="Spectral")
plt.title("The angle " + r'$X$')
plt.ylabel('y')
plt.xlabel('x')
plt.colorbar()
plt.savefig(img_save_dir+'X-angle-dns.png')

plt.figure(6)
plt.imshow(grad, cmap="Spectral")
plt.title("Concentration gradient")
plt.ylabel('y')
plt.xlabel('x')
plt.colorbar()
plt.savefig(img_save_dir+'theta-gradient-dns.png')

"""

Construct emperical PDF of the variables one the

```

```python
"""
hist_dict = dns_solver.VorticitySolvePostProcessing.histogram_avg(
    data_dir,20000,250000,-1)

(ya,xa) = hist_dict['LAMBDA']
(Xya,Xxa) = hist_dict['Xangle']
(Zya,Zxa) = hist_dict['mu']

plt.figure(1)
plt.title(r'$\Lambda$')
plt.plot(xa[:-1],ya/np.max(ya))
plt.savefig(img_save_dir+'lambda-hist-pdf.png')
plt.figure(2)
plt.title("The angle " + r'$X$')
Xxaa = Xxa[:-1]
plt.plot(Xxaa[600:1250]-6.2,Xya[600:1250]/np.max(Xya[600:1250]))
plt.savefig(img_save_dir+'angle-x-hist-pdf.png')
plt.figure(3)
plt.title('$\mu$')
plt.plot(Zxa[:-1],Zya)
plt.savefig(img_save_dir+'mu-hist-pdf.png')


"""

Run FokkerPlanck simulation

"""

import imp
import numpy as np
code_dir ='/home/ian/Desktop/MSc-Proj/code/thesis_code/'
data_dir ='/home/ian/Desktop/MSc-Proj/saved-datasets/'
img_save_dir = '/home/ian/Desktop/MSc-Proj/thesistemplate/imgs/'
chartDataDir='/home/ian/Desktop/MSc-Proj/chart-data/'

opt_params={'tau': 2}
opt_params_str=str(opt_params).replace("'","").replace(":","_").
    replace(" ","").replace("{","").replace("}","").replace(",","-")

chartDataDir=chartDataDir + str(opt_params_str) + '/'
import os
os.mkdir(chartDataDir)

fp_solver = imp.load_source('FokkerPlank', code_dir +'
    FokkerPlankSolve.py')
fp = fp_solver.FokkerPlank(1,opt_params)
```

```
fp.solve()

#save to file

fp_quantities={'x':None,'y':None,'z':None,'p_x':None,'p_y':None,'p_z':None,'p_xy':None,'p_yz':None, 'p_yz_anal': None,'p_lambda':None,'lambda_range':None}
for q in fp_quantities.keys():
    file_name = 'fp_<QUANT>.csv'.replace('<QUANT>',q)
    np.savetxt(chartDataDir + file_name , getattr(fp,q), delimiter=",",fmt='%1.10e')
#load from file
for q in fp_quantities.keys():
    file_name = 'fp_<QUANT>.csv'.replace('<QUANT>',q)
    fp_quantities[q] = np.loadtxt(chartDataDir + file_name, dtype='float', delimiter=',')


def plot_p_xy_surface(x_param,y_param,p_xy_param,data_dir):
    from mpl_toolkits.mplot3d import Axes3D
    import matplotlib.pyplot as plt
    from matplotlib import cm
    from matplotlib.ticker import LinearLocator, FormatStrFormatter
    import numpy as np
    XX,YY = np.meshgrid(x_param,y_param)
    XX = XX
    YY = YY
    ZZ = p_xy_param.T
    colors = cm.hot((ZZ-ZZ.min())/(ZZ.max() - ZZ.min()))
    rcount, ccount, _ = colors.shape
    fig = plt.figure(1)
    ax = fig.gca(projection='3d')
    surf = ax.plot_surface(XX, YY, ZZ, rcount=rcount, ccount=ccount, facecolors=colors, shade=True)
    surf.set_facecolor((0,0,0,0))
    ax.set_xlim(min(x_param), max(x_param))
    ax.set_ylim(min(y_param), max(y_param))
    ax.set_xlabel('x',fontsize=10)
    ax.set_ylabel('y', fontsize=10)
    ax.zaxis.set_rotate_label(False)
    ax.set_zlabel(r'$P_{xy}$', fontsize=10)
    plt.show()
    plt.savefig(data_dir + 'fp_p_xy.png')

plot_p_xy_surface(fp_quantities['x'],fp_quantities['y'],fp_quantities['p_xy'],chartDataDir)

```

```python
def plot_p_yz_surface(y_param, z_param, p_yz_param, data_dir):
    from mpl_toolkits.mplot3d import axes3d
    from matplotlib import cm
    import matplotlib.pyplot as plt
    YY,ZZ = np.meshgrid(y_param,z_param)
    YY = YY
    ZZ = ZZ
    P_YZ = p_yz_param.T
    colors = cm.coolwarm((P_YZ-P_YZ.min())/(P_YZ.max() - P_YZ.min()))
    rcount, ccount, _ = colors.shape
    fig = plt.figure(2)
    ax = fig.add_subplot(111, projection='3d')
    surf = ax.plot_surface(YY, ZZ, P_YZ, rcount=rcount, ccount=ccount,facecolors=colors, shade=True)
    surf.set_facecolor((0,0,0,0))
    ax.set_xlabel('y',fontsize=10)
    ax.set_ylabel('z', fontsize=10)
    ax.zaxis.set_rotate_label(False)
    ax.set_zlabel(r'$P_{yz}$', fontsize=10)
    plt.show()
    plt.savefig(data_dir + 'fp_p_yz.png')

plot_p_yz_surface(fp_quantities['y'],fp_quantities['z'],fp_quantities['p_yz'],chartDataDir)

def plot_lambda(lambda_range,p_lambda, data_dir):
    fig = plt.figure(3)
    plt.plot(lambda_range,p_lambda)
    plt.xlabel(r'$\lambda$')
    plt.ylabel(r'$P_{\Lambda}$')
    plt.show()
    plt.savefig(data_dir+'fp_p_lambda.png')

plot_lambda(fp_quantities['lambda_range'], fp_quantities['p_lambda'], chartDataDir)

def plot_p_x(x,p_x, data_dir):
    fig = plt.figure(4)
    plt.plot(x / np.pi,p_x)
    plt.xlabel(r'$x/\pi$')
    plt.ylabel(r'$P_{X}$')
    plt.show()
    plt.savefig(data_dir+'fp_p_x.png')

plot_p_x(fp_quantities['x'], fp_quantities['p_x'], chartDataDir)


```

```
fig = plt.figure(5)
plt.plot(fp.lambda_range,fp.p_lambda)
plt.plot(xa[:-1]/xa[:-1].max(),ya/ya.max())

fig = plt.figure(6)
plt.plot(fp.x / np.pi,fp.p_x)
plt.plot(Zxa[:-1],Zya/Zya.max())
```