DNS of Stratified Turbuluence with Rotation and Stochastic Forcing

Dante Buhl

October 2023

1 Previous Work

2 Current Work

3 Gaussian Processes

My current job is to design a stochastic forcing structure using the Gaussian random process. Gaussian Processes are a way of generating a regression from current data, fitting a line almost if you will. We are using gaussian processes to use the current data to inform a new point going forward in the code.

The concept of the Gaussian Process is not a novel idea. Its purpose is to generate new points which fit onto an informed window of uncertainty around a given set of initial data. Ultimately, the process samples a gaussian distribution whose mean and covariance matrices are created through the use of precise linear algebra and a kernel chosen to optimize on the desired properties of the gaussian regression.

The purpose of the Gaussian Process in the context of this work is to create a statistically stationary stochastic forcing in which to perturb and drive eddies in a stable manner as done in (Waite 2004) **SOURCE**. In our Spectral Code, the Gaussian Forcing was enforced on low horizontal wavenumbers as to affect the mean background flow, without directly interacting with the turbulence structures.

$$\begin{aligned} \boldsymbol{G}(k,t) &= \langle G_x(k,t), G_y(k,t)\rangle \\ \boldsymbol{k_h}\cdot\boldsymbol{G}(k,t) &= 0 \end{aligned}$$

3.1 Dealing with Finite Precision

The procedure in which a Gaussian Process is generated is usually not a very complex Linear Algebra structure. Given a training set, $\boldsymbol{x}$ and $\boldsymbol{y}$, both of dimension $(1\times n)$, and a test set, $\boldsymbol{x}_*$, of dimension $(1\times n_*)$, the gaussian process regression, $\boldsymbol{y}_*$ is given below.

$$\begin{aligned} \boldsymbol{y}_* &\sim \mathcal{N}(\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*) & \mathbf{K} &:= \boldsymbol{\mathcal{K}}(\boldsymbol{x},\boldsymbol{x}) \\ \boldsymbol{\mu}_* &= \boldsymbol{K}_*^T \times \mathbf{K}^{-1} \times \boldsymbol{y} & \mathbf{K}_* &:= \boldsymbol{\mathcal{K}}(\boldsymbol{x},\boldsymbol{x}_*) \\ \boldsymbol{\Sigma}_* &= \mathbf{K}_{**} - \mathbf{K}_*^T \times \mathbf{K}^{-1} \times \mathbf{K}_* & \mathbf{K}_{**} &:= \boldsymbol{\mathcal{K}}(\boldsymbol{x}_*,\boldsymbol{x}_*) \end{aligned}$$

It should be noted that the covariance matrices, $\mathbf{K}$, are positive definite and generated through the use of the Exponential Squared Kernel, $\boldsymbol{\mathcal{K}}$. The Kernel is defined below where, $\boldsymbol{a}$ and $\boldsymbol{b}$, are vectors with all real values (i.e. $\alpha_i, \beta_j \in \mathbb{R}$), and the kernel function, f, depends on the Gaussian Scale Parameter, σ.

$$\begin{aligned} f(x_1,x_2) &= \exp\left(\frac{-(x_1-x_2)^2}{2\sigma^2}\right) \\ \boldsymbol{a} &= [\alpha_1, \alpha_2, \cdots, \alpha_n] \\ \boldsymbol{b} &= [\beta_1, \beta_2, \cdots, \beta_m] \end{aligned} \qquad \boldsymbol{\mathcal{K}}(\boldsymbol{a},\boldsymbol{b}) = \begin{bmatrix} f(\boldsymbol{a}[1],\boldsymbol{b}[1]) & f(\boldsymbol{a}[1],\boldsymbol{b}[2]) & \cdots & f(\boldsymbol{a}[1],\boldsymbol{b}[m]) \\ f(\boldsymbol{a}[2],\boldsymbol{b}[1]) & f(\boldsymbol{a}[2],\boldsymbol{b}[2]) & \cdots & f(\boldsymbol{a}[2],\boldsymbol{b}[m]) \\ \vdots & \vdots & \ddots & \vdots \\ f(\boldsymbol{a}[n],\boldsymbol{b}[1]) & f(\boldsymbol{a}[n],\boldsymbol{b}[2]) & \cdots & f(\boldsymbol{a}[n],\boldsymbol{b}[m]) \end{bmatrix}$$

This model for the Gaussian Process works well when the matrices are well conditioned and generally, our training set of data is not very large ($n < 10^2$). However, since we will be running this code with very small timesteps ($\sim 5 \cdot 10^{-4}$) up to 1000 or 10,000 time units, if we don't restrict the training set, we will soon have a very ill-conditioned matrix, $\mathbf{K}$, and our inverse matrix, $\mathbf{K}^{-1}$, will be inaccurate due to finite precision errors. The remedy to this is to modify our equations and algorithm slightly. We introduce the concept of the psuedo inverse, so that we can discard eigenvalues and eigenvectors of our matrix, $\mathbf{K}$, which would be vulnerable to finite precision errors. Thus an eigendecomposition is needed in order to eliminate eigenvalues/vectors. The matrix, $\mathbf{K}$, is specifically an $(n \times n)$ square, symmetric matrix by definition (this will affect the eigendecomposition routine used in the code). The psuedo inverse, $\mathbf{K}_p^{-1}$, is defined using a matrix, $\boldsymbol{Q}$, whose columns are the eigenvectors of $\mathbf{K}$, $\boldsymbol{\Lambda_i}$, and a diagonal matrix, $\boldsymbol{D}_p$, whose diagonal entries are the corresponding eigenvalues of $\mathbf{K}$, λ_i, in ascending order. We also use a relative threshold, τ, such that any eigenvector less than, $\tau \cdot \lambda_n$, is discarded. The first eigenvalue above this relative tolerance is defined to be, λ_{n_c}.

$$\mathbf{K}_p := \boldsymbol{Q}\boldsymbol{D}_p^{-1}\boldsymbol{Q}^T$$
$$\boldsymbol{D}_p := \mathrm{diag}(\lambda_{n_c}, \cdots, \lambda_n)$$
$$\boldsymbol{Q} := [\boldsymbol{\Lambda_{n_c}}, \cdots, \boldsymbol{\Lambda_n}]$$

The aforementioned prodecure to generate $\boldsymbol{y}_*$ is modified to use the pseudo inverse and its components.

$$\boldsymbol{\mu}_* = \boldsymbol{K}_*^T\mathbf{K}^{-1}\boldsymbol{y} = (\boldsymbol{Q}^T\mathbf{K}_*)^T\boldsymbol{D}_p^{-1}(\boldsymbol{Q}^T\boldsymbol{x})$$
$$\boldsymbol{\Sigma}_* = \mathbf{K}_{**} - \mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{K}_* = \mathbf{K}_{**} - (\boldsymbol{Q}^T\mathbf{K}_*)^T\boldsymbol{D}_p^{-1}(\boldsymbol{Q}^T\mathbf{K}_*)$$

The specific value of the tolerance used is open to interpretation and experiementation. While the ratio of the largest and smallest eigenvalue of a matrix is related to the condition number of a matrix. This is the assignment for $\boldsymbol{\mu}_*$ and $\boldsymbol{\Sigma}_*$ as used in the code.

Our last consideration is to simplify sampling from a multivariate normal (AKA gaussian) distribution. Since our code will be written in fortran, which lacks a built in function to sample a standard normal distribution, we must build our own sampler. It is easier to sample a multivariate standard normal distribution and use the $\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*$ matrices to transform it to the distribution we want. Different methods of accomplishing this vary on the factorization method of $\boldsymbol{\Sigma}_*$. Remember for the univariate case, we have that if X is distributed normally with a mean, μ, and standard deviation, σ, we have,

$$X = \mu + \sigma Z,\ Z \sim \mathcal{N}(0, 1).$$

Similarly, we can obtain a "standard deviation" matrix with the

4 Code Design and Algorithm Structure