Finite Element Solution to Fokker-Planck Formulation of Chaotic Ordinary Differential equations

By Vignesh Ramakrishnan

RIN: 662028006

Abstract

This project is intended to arrive at a Finite Element based numerical solution to the Fokker-Planck formulation of Chaotic Ordinary Differential Equations (ODE). A little background about chaotic ODEs is necessary to realize the significance of the Fokker-Planck formulation for the system of ODEs. Random excitation of a dynamical system can trigger extremely varied responses in the trajectory of its states. Even for the slightest of changes in the excitation condition or the parameters that define the dynamics of the system, the response of state trajectories over time are extremely varied. This sort of a behavior is observed in a nonlinear system of ordinary differential equations [1]. Fokker-Planck equation describes the evolution of the Probability Density Function (PDF) of the state variables over time.

Physical Dynamics and its Fokker-Planck formulation

Three ~~cases of~~ dynamical systems are considered, and their individual Fokker-Planck formulation is explained in this project. But the general structure of all three cases is the same and they follow the system of equations described in Equation 1 and Equation 2.

$$\frac{dx}{dt} = f(x,t,s) + \eta(t) \qquad \textit{Equation 1}$$

$$\frac{\partial \rho}{\partial t} = -\nabla(f(x,t,s)\rho) + \frac{h^2}{2}\nabla^2\rho \qquad \textit{Equation 2}$$

$x - state\ variables$
$f(x,t,s) - system\ dynamics$
$s - parameters\ defining\ dynamics$
$\eta(t) - stochastic\ forcing\ of\ dynamic\ system$
$\rho - pdf\ of\ the\ system\ of\ equations$

Equation 2 is the Fokker-Planck formulation for the system of equations described in Equation 1. This project focusses on the addition ~~on~~ multiplicative white noise as its behavior has been extensively studied and understood. [2] states that FPE describes the flow of probability density in phase space considering the conservation of total probability given by $\int \rho \, d\Omega = 1$ over the entire domain under consideration. This linear Partial Differential Equation resembles the form of an advection, diffusion equation where the probability flow advects and diffuses in the phase-space domain.

There are analytical solutions available to some formulations of these dynamical systems represented as the FPE equivalent through stochastic forcing, ex. Ornstein Uhlenbeck equation shown in Equation 3. Finding the analytical solution becomes difficult when the number of dimensions ~~increase~~ along with the nonlinearity and the coupled phase space dynamics. So, numerous papers have been published trying to get numerical solutions to the FPE of different nonlinear chaotic dynamic systems. This project will concentrate on using Finite Element Methods to arrive at a numerical solution to multiple FPE equations. Finite Difference approach has also been adopted to perform a comparative study.

Another important aspect to keep note of is that most applications involve the infinite time stationery solution to the FPE [1]. This presents significant information regarding the long-term behavior of the system. Even though the parameters that are part of the system of ODEs are time dependent, it is not a bad assumption to consider them to remain constant over long periods of time.

The three specific cases considered in this project are described below:

$1D\ FPE$
Ornstein Uhlenbeck
$a = \{1,3\}$
Multiplicative noise $h^2/2$
$h = \{0.3,0.5,0.8\}$
Equation 3

$$\begin{cases} \dfrac{dx}{dt} = -ax + \eta(t) \\ \dfrac{\partial \rho}{\partial t} = -\dfrac{\partial(-ax\rho)}{\partial x} + \dfrac{h^2}{2}\left(\dfrac{\partial^2 \rho}{\partial x^2}\right) \end{cases}$$

$2D\ FPE - VanDerPol$
$\varepsilon = \{0.2,0.5,1\}$
Multiplicative noise $h^2/2$
$h = \{0.1, 0.3\}$
Equation 4

$$\begin{cases} \dfrac{dx_1}{dt} = x_2 + \eta_1(t) \\ \dfrac{dx_2}{dt} = x_1 + \varepsilon(1 - x_1^2)x_2 + \eta_2(t) \\ \dfrac{\partial \rho}{\partial t} = -\dfrac{\partial(x_2\rho)}{\partial x_1} - \dfrac{\partial\left((-x_1 + \varepsilon(1 - x_1^2)x_2)\rho\right)}{\partial x_2} + \dfrac{h^2}{2}\left\{\dfrac{\partial^2 \rho}{\partial x_1^2} + \dfrac{\partial^2 \rho}{\partial x_2^2}\right\} \end{cases}$$

$3D\ FPE - Lorenz\ System$
$\sigma = 3, \gamma = 26.5, \beta = 1$
Multiplicative noise $h^2/2$
Equation 5

$$\begin{cases} \dfrac{dx}{dt} = \sigma(y - x) + \eta_1(t) \\ \dfrac{dy}{dt} = x(\gamma - z) + \eta_2(t) \\ \dfrac{dz}{dt} = xy - \beta z + \eta_3(t) \\ \dfrac{\partial \rho}{\partial t} = -\left\{\dfrac{\partial(f_1\rho)}{\partial x} + \dfrac{\partial(f_2\rho)}{\partial y} + \dfrac{\partial(f_3\rho)}{\partial z}\right\} + \dfrac{h^2}{2}\left\{\dfrac{\partial^2 \rho}{\partial x^2} + \dfrac{\partial^2 \rho}{\partial y^2} + \dfrac{\partial^2 \rho}{\partial z^2}\right\} \\ f_1 = \sigma(y - x); f_2 = x(\gamma - z); f_3 = (xy - \beta z) \end{cases}$$

Finite Difference Discretization of the FPE equation

The left-hand side of the FPE is omitted in all 3 cases because most practical applications are interested in the infinite time response which vary with parameters that define the dynamics of the system. But if the time evolution of the Probability Density of the phase-space is of interest, Finite Difference discretization methodology is the best way to approach this problem in hand. Various explicit and implicit time marching techniques have been described in [3].

Linear Fokker-Planck for a general n-dimensional system of Ordinary Differential Equations is described by the following equation. As mentioned before, stochastic forcing enforced amounts to only Multiplicative

noise and not multiplicative noise and hence there is no cross-relation of noise covariance terms between any two states of the phase-space.

Linear Fokker-Planck formulation for any n-dimensional system of Ordinary Differential equation

Equation 6

$$\begin{cases} \dfrac{dx_i}{dt} = f_i(x,t) + \eta_i(t), i \in [1,n] \\ \dfrac{\partial \rho}{\partial t} = -\sum_{i=1}^{i=n} \dfrac{\partial f_i \rho}{\partial x_i} + \dfrac{h^2}{2} \sum_{i=1}^{i=n} \dfrac{\partial^2 \rho}{\partial x_i^2} \end{cases}$$

This discretization is coupled with normalization and a zero-flux condition for the probability density

$$\int_{\Omega} \rho \, d\Omega = 1$$

Equation 7. Normalization condition

$$\rho(x_i, t) \to 0, as \; x_i \to \pm\infty \;\; i = 1,2,3 \dots \dots, n$$

Equation 8. Zero-Flux condition

2-D FPE of the VanDerPol Oscillator shown in Equation 4, is considered to explain the Finite Difference discretization process.

$$f_1(x,t) = x_2; f_2(x,t) = -x_1 + \epsilon(1 - x_1^2)x_2$$

$$\rho_{i,j}^{m+1} = \rho_{i,j}^{m} + \Delta t \left\{ -\frac{(f_1\rho)_{i+1,j}^{m} - (f_1\rho)_{i-1,j}^{m}}{2\Delta x_1} - \frac{(f_2\rho)_{i,j+1}^{m} - (f_2\rho)_{i,j-1}^{m}}{2\Delta x_2} + \frac{h^2}{2}\left[\frac{\rho_{i+1,j}^{m} - 2\rho_{i,j}^{m} + \rho_{i-1,j}^{m}}{\Delta x_1^2} + \frac{\rho_{i,j+1}^{m} - 2\rho_{i,j}^{m} + \rho_{i,j-1}^{m}}{\Delta x_2^2} \right] \right\}$$

Equation 9

f_1, f_2 are flux quantities computed at the corresponding grid location (x_1, x_2). A central difference discretization is preferred to maintain stability of the system that is formed. This is explicit time marching and one of the simplest available techniques available to find the time evolution of the Probability density of state variables.

If the interest is to find the infinite time response, then the following discretization technique is used.

$\rho_{i,j} = \rho_k, k = 1,2, \dots \dots \dots (mn)$ where $m = \max(i)\,; n = \max(j)$

$$-\frac{(f_1\rho)_{i+1,j}^{m} - (f_1\rho)_{i-1,j}^{m}}{2\Delta x_1} - \frac{(f_2\rho)_{i,j+1}^{m} - (f_2\rho)_{i,j-1}^{m}}{2\Delta x_2} + \frac{h^2}{2}\left[\frac{\rho_{i+1,j}^{m} - 2\rho_{i,j}^{m} + \rho_{i-1,j}^{m}}{\Delta x_1^2} + \frac{\rho_{i,j+1}^{m} - 2\rho_{i,j}^{m} + \rho_{i,j-1}^{m}}{\Delta x_2^2} \right] = 0$$

Equation 10

$\sum_{k=1}^{k=mn} w_k \rho_k - 1 = 0$, where w_k denotes to the trapezoidal weights associated to ρ_k

This system of equations become,

$$\begin{bmatrix} A & W^T \\ W & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\rho} \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \textit{Equation 11}$$

$$A - system\ matrix\ = mn\ \times\ mn$$
$$W - Trapezoidal\ weight\ array = \ 1\ \times\ mn$$
$$\boldsymbol{\rho} - probability\ density = mn\ \times\ 1$$
$$\lambda - constraint\ multiplier = 1\ \times\ 1$$

Finite Element Discretization of the FPE

Once again, the general n- dimensional FPE described in Equation 6., is used for coming up with the weak form of the FPE. The derivation of weak form for the general case is stated below. The final weak form derived is given in Equation 13.

$$-\sum_{i=1}^{i=n} \frac{\partial (f_i \rho)}{\partial x_i} + \frac{h^2}{2} \frac{\partial^2 \rho}{\partial x_i^2} \ = 0$$

$$\Rightarrow \sum_{i=1}^{i=n} - \int w \frac{\partial (f_i \rho)}{\partial x_i}\, d\Omega + \frac{h^2}{2} \int w \frac{\partial^2 \rho}{\partial x_i^2}\, d\Omega = 0$$

$$\frac{\partial (w\, f_i \rho)}{\partial x_i} = \frac{\partial w}{\partial x_i} f_i \rho + \ w \frac{\partial (f_i \rho)}{\partial x_i}$$

$$\frac{\partial \left(w \frac{\partial \rho}{\partial x_i} \right)}{\partial x_i} = \frac{\partial \rho}{\partial x_i} \frac{\partial w}{\partial x_i} + \ w \frac{\partial^2 \rho}{\partial x_i^2}$$

$$\Rightarrow \sum_{i=1}^{i=n} - \int \frac{\partial (w\, f_i \rho)}{\partial x_i} - \frac{\partial w}{\partial x_i} f_i \rho \, d\Omega + \frac{h^2}{2} \int \frac{\partial \left(w \frac{\partial \rho}{\partial x_i} \right)}{\partial x_i} - \frac{\partial \rho}{\partial x_i} \frac{\partial w}{\partial x_i}\, d\Omega \ = 0$$

$$\Rightarrow \sum_{i=0}^{n} - \int_{\Gamma} w\, f_i \rho \, d\Gamma + \int \frac{\partial w}{\partial x_i} f_i \rho \, d\Omega \ + \frac{h^2}{2} \int_{\Gamma} \ w \frac{\partial \rho}{\partial x_i}\, d\Gamma - \frac{h^2}{2} \int \frac{\partial \rho}{\partial x_i} \frac{\partial w}{\partial x_i}\, d\Omega = 0 \qquad \textit{Equation 12}$$

$$Zero - Flux\ at\ domain\ edges \Rightarrow \rho = 0\ on\ \Gamma$$

$$\Rightarrow \sum_{i=0}^{n} \int \frac{\partial w}{\partial x_i} f_i \rho \, d\Omega - \frac{h^2}{2} \int \frac{\partial \rho}{\partial x_i} \frac{\partial w}{\partial x_i}\, d\Omega = 0 \qquad \textit{Equation 13}$$

$$\int \rho \, d\Omega - 1 = \int \rho \, d\Omega - \frac{1}{V} \int 1. d\Omega\ ;\ (V - Area\ of\ \Omega\ in\ 2D, Volume\ of\ \Omega \text{ in 3D})$$

$$\int w\,\rho\,d\Omega = \frac{1}{V}\int w\,.\,1.\,d\Omega$$

From this derivation three integrators have been generated, namely, Mixed Scalar Divergence integrator, Diffusion integrator and the Mass integrator.

$$V^h = \{w^h \in H^1 : w^h = 0\ on\ \Gamma\}$$

$$\delta^h = \{q^h \in H^1 :\ q^h = 0\ on\ \Gamma\}$$

$$\rho^h = u^h + q^h;\ \ u^h \in V^h, q^h \in \delta^h$$

$$w^h = \sum_{A=1}^{\eta-\eta_q} c_A\,N_A(x)$$

$$u^h = \sum_{B=1}^{\eta-\eta_q} d_B\,N_B(x)$$

Equation 14

$$\eta - \{global\ nodes\}; \eta_q - \{Boundary\ nodes\}$$

Finally, we get the form this form for each element as shown in and that can be assembled to get the final Linear system described in

$$K^e_{ab} = \sum_{i=0}^{n}\int_{\Omega^e}\frac{\partial N^e_a}{\partial x_i}\,f_i N^e_b\,d\Omega - \frac{h^2}{2}\int_{\Omega^e}\frac{\partial N^e_a}{\partial x_i}\frac{\partial N^e_b}{\partial x_i}\,d\Omega$$

$$M^e_{ab} = \int N^e_a\,N^e_b\,d\Omega$$

Equation 15

$$\begin{bmatrix} K & M^T \\ M & 0 \end{bmatrix}\begin{bmatrix} \boldsymbol{d} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} 0 \\ \mathbf{1} \end{bmatrix}$$

$$\rho = \boldsymbol{d}$$

H^1 class Finite Element shape functions can't be used if time evolution of the probability density needs to be studied. For an infinite-time response, the Petrov-Galerkin method of weighted residuals work well for dimensions up to 3. Formulation of weak form and the Bubnov-Galerkin method of weighted residuals to get time evolution have been studied in [4]. [3], [4] have discussions that explain why this formulation suffers the curse of dimensionality and for any number $n > 4$ dimensions, Finite Element methods have a hard time dealing with the exponentially growing nodes. Some methods to possibly avoid this is to have coarse mesh in areas of little to no action and refine the mesh to get finer elements in regions of action. This requires some prior knowledge of where the solution trajectory would lie on the phase-space.

Finite Element Method Description

The 1D Finite Element code developed in MATLAB for this project used both 1st and 2nd order Finite Elements shown in Figure 1 and Figure 2.

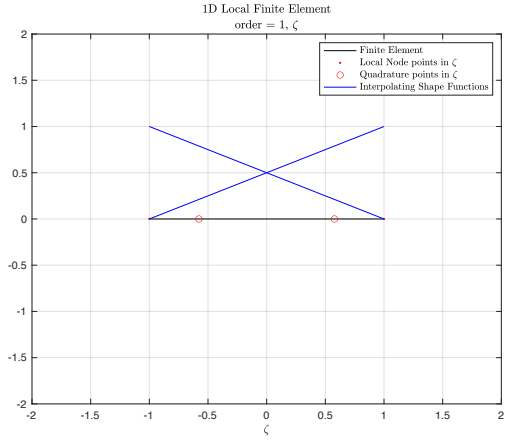


Figure 1. Linear Finite Element Shape Functions

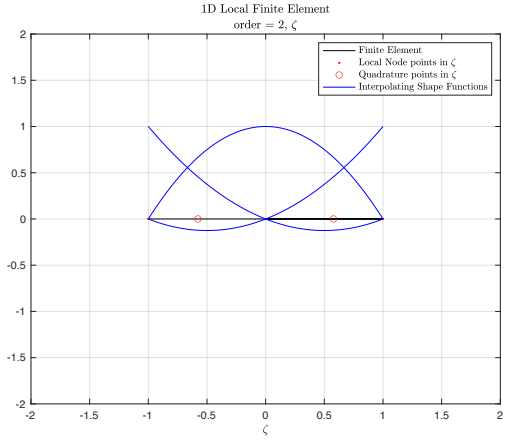


Figure 2. Quadratic Finite Element Shape Functions

The Shape Functions are given in Table 1.

1st Order		**2nd Order**	
$\zeta = -1$	$N_1(\zeta) = \frac{1}{2}(1-\zeta)$	$\zeta = -1$	$N_1(\zeta) = \frac{1}{2}\zeta(\zeta-1)$
$\zeta = 1$	$N_2(\zeta) = \frac{1}{2}(1+\zeta)$	$\zeta = 0$	$N_2(\zeta) = 1-\zeta^2$
		$\zeta = 1$	$N_3(\zeta) = \frac{1}{2}\zeta(\zeta+1)$
Element Transformation	$x = \sum_{i=1}^{2} N_i^e(\zeta)x_i^e$	Element Transformation	$x = \sum_{i=1}^{3} N_i^e(\zeta)x_i^e$

Table 1. Shape Functions of Finite Elements

$$\frac{dx}{d\zeta} = \sum_{i=1}^{nodes} \frac{dN_i^e(\zeta)}{d\zeta} x_i^e = Jacobian \qquad \text{Equation 16}$$

When the order is different, it is important to allocate multiple nodes to the same element with respect to the order of interpolating polynomials being used to approximate the probability density at the nodal points.

The 2D Finite Element code developed in MATLAB for this project uses 1st Order Finite Elements shown in Figure 3.

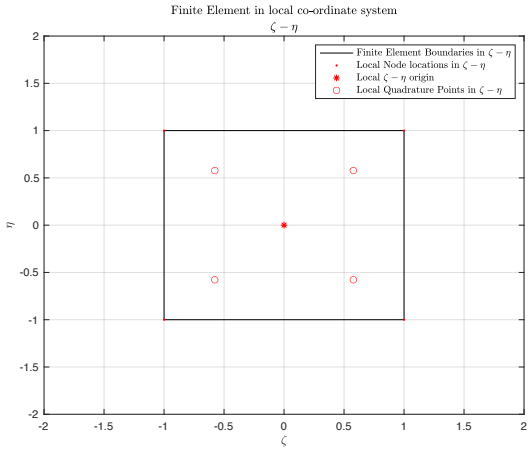


Figure 3. Finite Element in 2D local coordinate system

The shape functions used are described in the following Table 2.

Point	Lagrange Polynomial
$[-1,-1]$	$N_1(\zeta,\eta) = \frac{1}{4}(1-\zeta)(1-\eta)$
$[1,-1]$	$N_2(\zeta,\eta) = \frac{1}{4}(1+\zeta)(1-\eta)$
$[-1,1]$	$N_3(\zeta,\eta) = \frac{1}{4}(1-\zeta)(1+\eta)$
$[1,1]$	$N_4(\zeta,\eta) = \frac{1}{4}(1+\zeta)(1+\eta)$

Table 2. Shape Functions of the 2D Finite Element

These are iso-parametric elements and its corresponding transformation from the local $\zeta - \eta$ coordinate system to the global $x_1 - x_2$ coordinate system is given by Equation 17.

$$x = \sum_{i=1}^{4} N_i^e(\zeta,\eta)x_i^e \qquad y = \sum_{i=1}^{4} N_i^e(\zeta,\eta)y_i^e$$

$$J = \begin{bmatrix} \frac{\partial x}{\partial \zeta} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \zeta} & \frac{\partial y}{\partial \eta} \end{bmatrix}; \qquad j = |J|; \qquad \begin{bmatrix} \frac{\partial \zeta}{\partial x} & \frac{\partial \zeta}{\partial y} \\ \frac{\partial \eta}{\partial x} & \frac{\partial \eta}{\partial y} \end{bmatrix} = \frac{1}{j}\begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial x}{\partial \eta} \\ -\frac{\partial y}{\partial \zeta} & \frac{\partial x}{\partial \zeta} \end{bmatrix}$$

Equation 17

(x_i^e, y_i^e) are the global nodal coordinate values of the corresponding local element.
From the above derivation of the weak form, there are 3 main integrators that must be developed.

i. Diffusion Integrator
ii. Mixed Scalar Divergence Integrator
iii. Mass Integrator

For all three integrators, a numerical integration scheme is also developed using Gauss-Legendre polynomial rule. The quadrature points are plotted on Figure 3. This integration scheme is exact for the 1st order element and it's a good approximate integral for the 2nd order element. Another important aspect of numerical integration is to perform element transformation of each element from the global coordinate

system to the local coordinate system. This is where the iso-parametric representation comes in extremely handy. Hence, all integrals can be performed in ζ coordinate system (1D) or the $\zeta - \eta$ coordinate system in 2D. Then all local element stiffness matrices are assembled by combining the right contributions of the nodes of each element to form the global stiffness matrix. They assembly method described in [5], was adopted in this project. This leads to a symmetric and sparse Element Stiffness Matrix of huge sizes. Hence, sparse linear algebra is used to solve the system of Linear equations defined by Eq., example ~~*gmres*~~ solver. An example of ~~how the Left-Hand-Side of the equation looks~~ is shown in Figure 4.

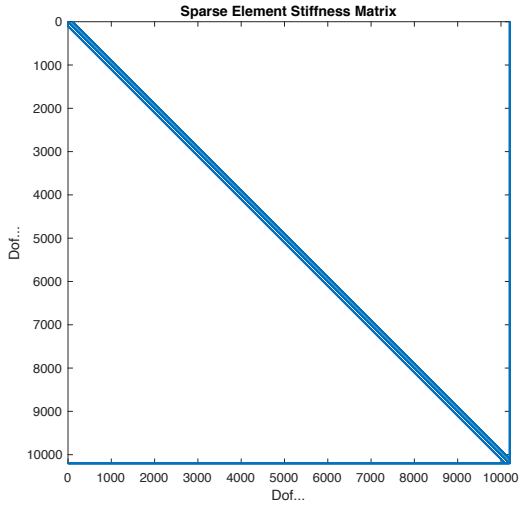


Figure 4. Checking for sparsity of the matrix

This is almost a strictly symmetric ~~diagonal~~ system of equations, but it is weighted in the last row and the last column with trapezoidal weights on each node of the mesh to force the probability density to add up to 1 over the domain. The boundary conditions are not applied directly on to each node as it will convert the sparse matrix into a dense matrix, which is extremely computationally costly. Hence it has been weakly applied as a form of constraint equation rather, to keep this problem still solvable. Sizes of the required storage rises exponentially as the dimensions of the problem increases. It is therefore important to keep this system of equations sparse in order to not run into storage troubles. If each dimension is split into 100 elements, the size requirement of the element stiffness matrix can be seen in Table 3.

Dimension	Degrees of Freedom	Size of Element Stiffness Matrix
1D	101	0.04MB
2D	10201	416.24 MB
3D	1030301	4246.1 GB

Table 3. Dense Matrix sizes requirements

1-D Fokker-Planck Equation

The infinite time response to the Ornstein-Uhlenbeck equation (Equation 3) has an analytic solution:

$$-\frac{\partial(-ax\rho)}{\partial x} + \frac{h^2}{2}\left(\frac{\partial^2\rho}{\partial x^2}\right) = 0 \qquad \Rightarrow \qquad \rho(x) = \sqrt{\frac{a}{2\pi\Gamma}}e^{-ax^2/2\Gamma}, \qquad \Gamma = \frac{h^2}{2}$$

I developed a MATLAB script for this 1D problem to arrive at a numerical solution using Finite Element methods. Second order Lagrange polynomial shape functions were used. Domain of the state variable $x \in [-3\,, 3]$. This domain was discretized to have 100 elements with 3 nodes on each element. The source code has been attached to the appendix of this report.

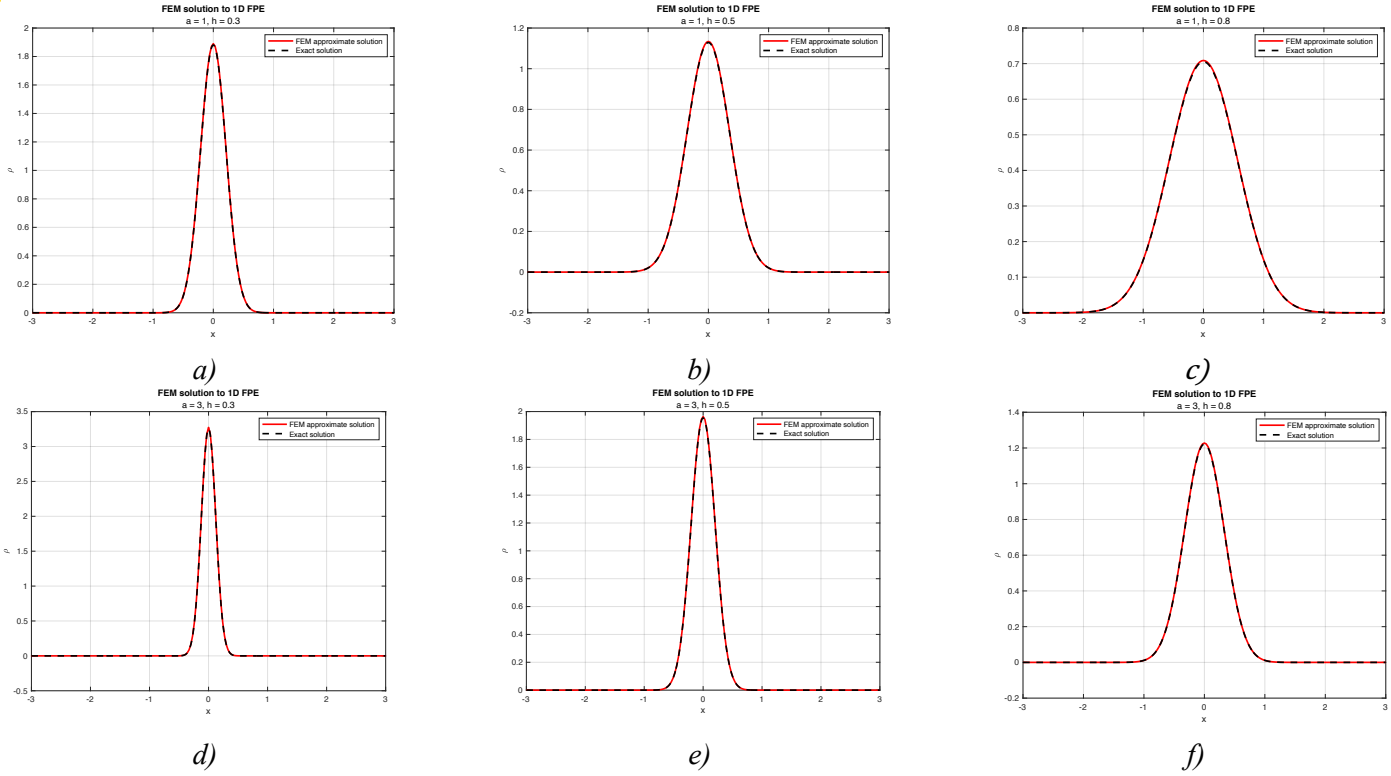


Figure 5. 1D FPE null space solutions.

The understanding behind these results is that the maximum possibility or probability to find the state variable $x \in \Omega$ that obeys the dynamic equation $\dot{x} = -ax$ is given by the probability density $\rho(x)$. In other words, the probability of the trajectory of the state-variable $x \in \Omega$, is given by the probability density function $\rho(x)$. This solution is the null-mode solution to the Fokker-Planck formulation of the 1D ODE equation. A good approximate solution is obtained using $H1$- 2^{nd} order Lagrange Polynomial shape functions. The accuracy of the solutions can be verified by finding the norm of the difference between the numerical solution and the analytical solution. Figure 6 and Figure 7 shows the convergence analysis performed. These results provide motivation to extend this analysis to multi-dimensional cases, namely the Linear Fokker-Planck formulations of the VanDerPol oscillator and the Lorenz attractor.

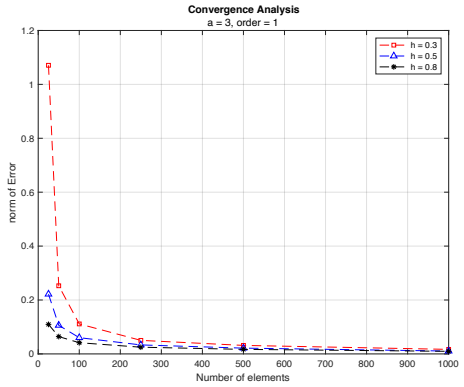


Figure 6. Converge of error norm – 1st order Shape Functions

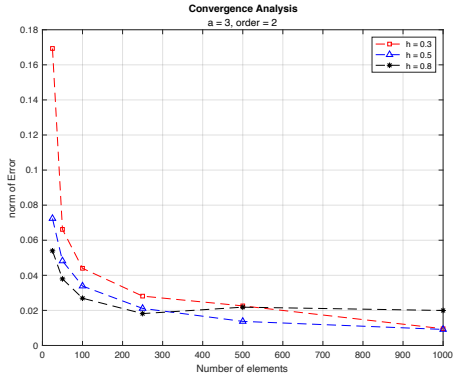


Figure 7. Converge of error norm – 2nd order Shape Functions

It can be observed that the norm of the error is, 1 order of magnitude lesser if 2nd order interpolating polynomials are used as opposed to 1st order interpolating polynomials.

2-D Fokker-Planck Equation

Before the Fokker-Planck equation is solved, it is important to look at the deterministic solution to these ordinary differential equations and the chaotic nature of these equations.

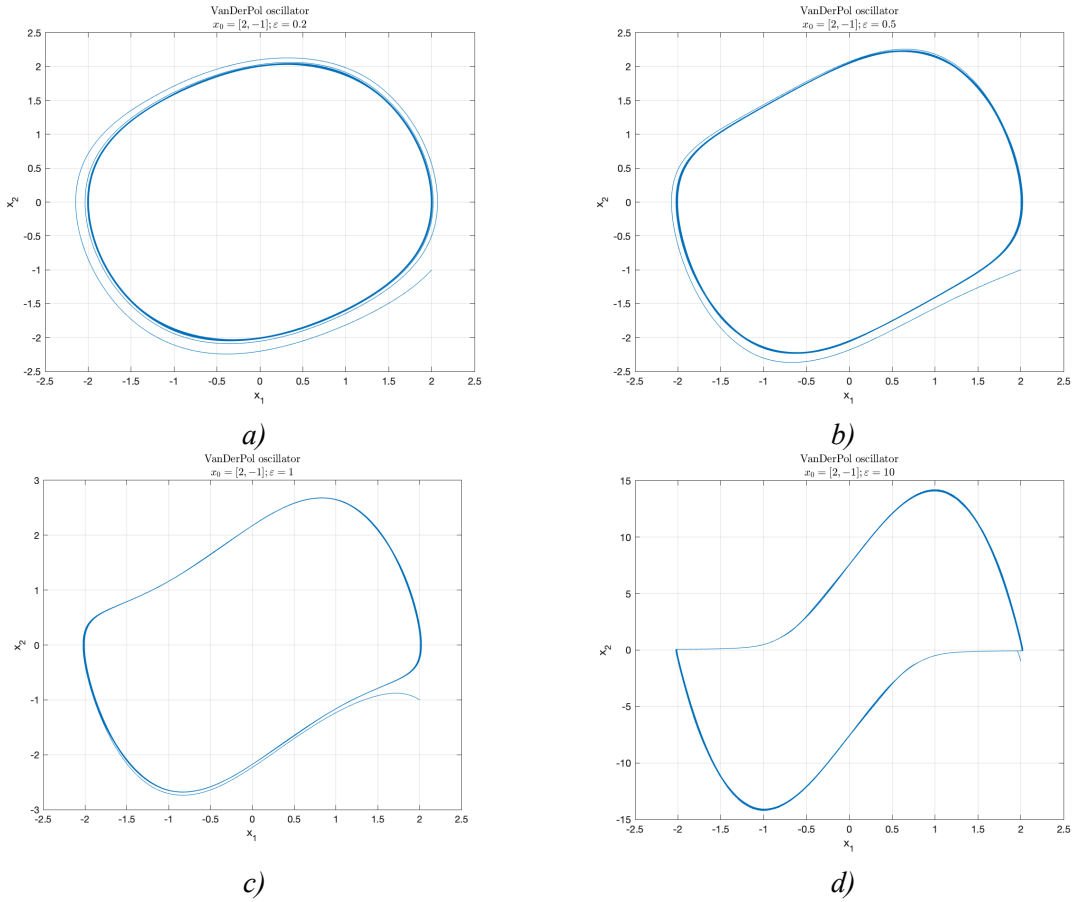


Figure 8. Deterministic Solution Trajectories of VanDerPol Oscillator

For the same initial conditions, the trajectory of the state-space variables over long periods of time change drastically with respect to the parameters used, in this case the parameter is ε. Both the shape of the phase-space trajectory and the range of values it attains over long periods of time change drastically even with a small change in the value taken by the parameter ε. This proves the chaotic nature of this oscillator and its sensitivity to parameters.

Three approaches were taken to arrive at the solution of VanDerPol FPE. Finite Difference approach, Finite Element Method code written in MATLAB, developed by me, and MFEM. The source code for all three methods have been added in the appendix of this project. In all three cases, the following mesh was generated and utilized to get the numerical solution.
Domain of phase-space $x \in [-3, 3] \times [-3, 3]$, with a resolution of $dx_1 = dx_2 = 0.06$. 10201 nodes are present, and 10000 elements are present when looking at this mesh from a Finite Element Method perspective.

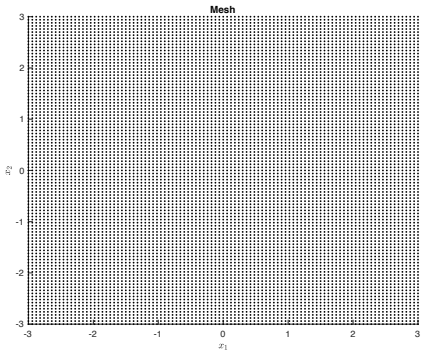


Figure 9. Domain and the nodes present in the mesh

For the same mesh, the numerical solutions obtained for different parameters and stochastic forcing for the VanDerPol FPE is discussed below. The solutions obtained are presented below.

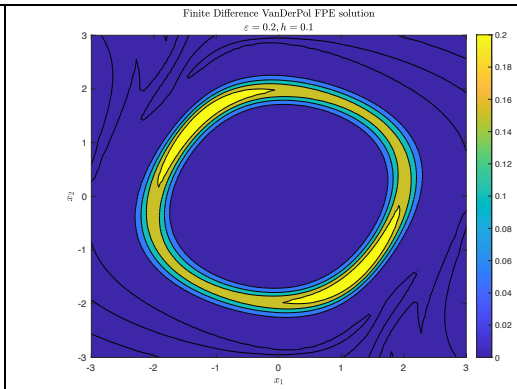
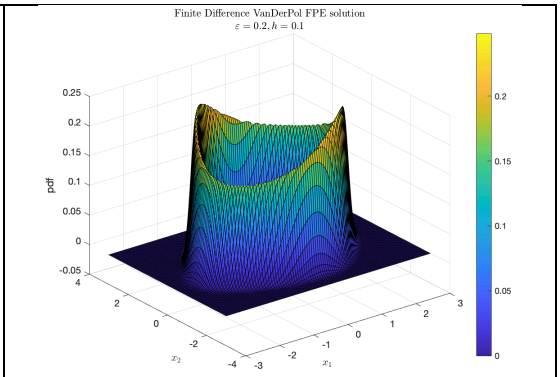
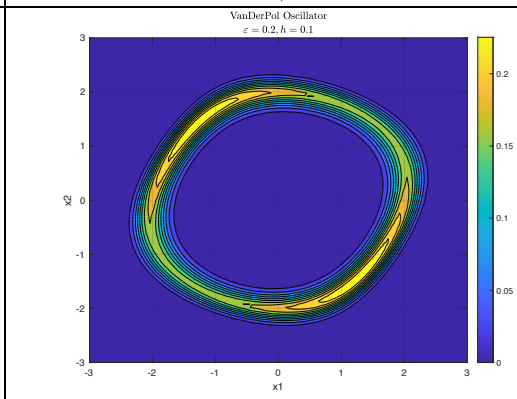
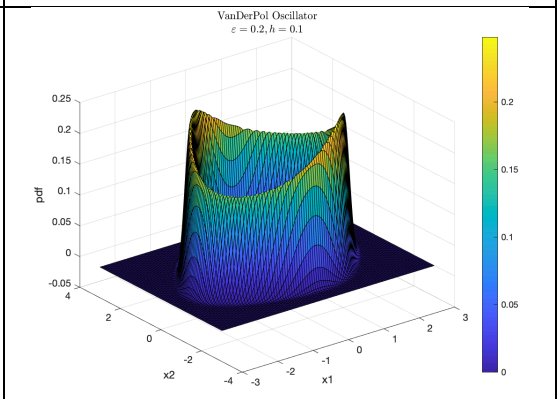
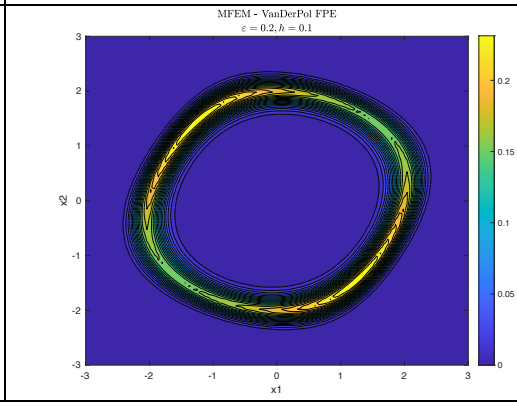
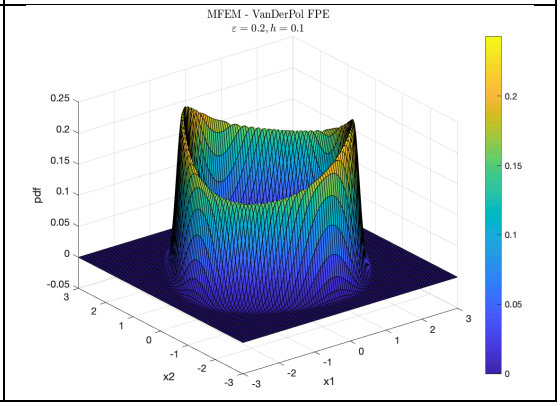
Finite Difference Method		
MATLAB FEM code		
MFEM		

Figure 10. FPE null space solution to VanDerPol Oscillator, $\varepsilon = 0.2, h = 0.1$

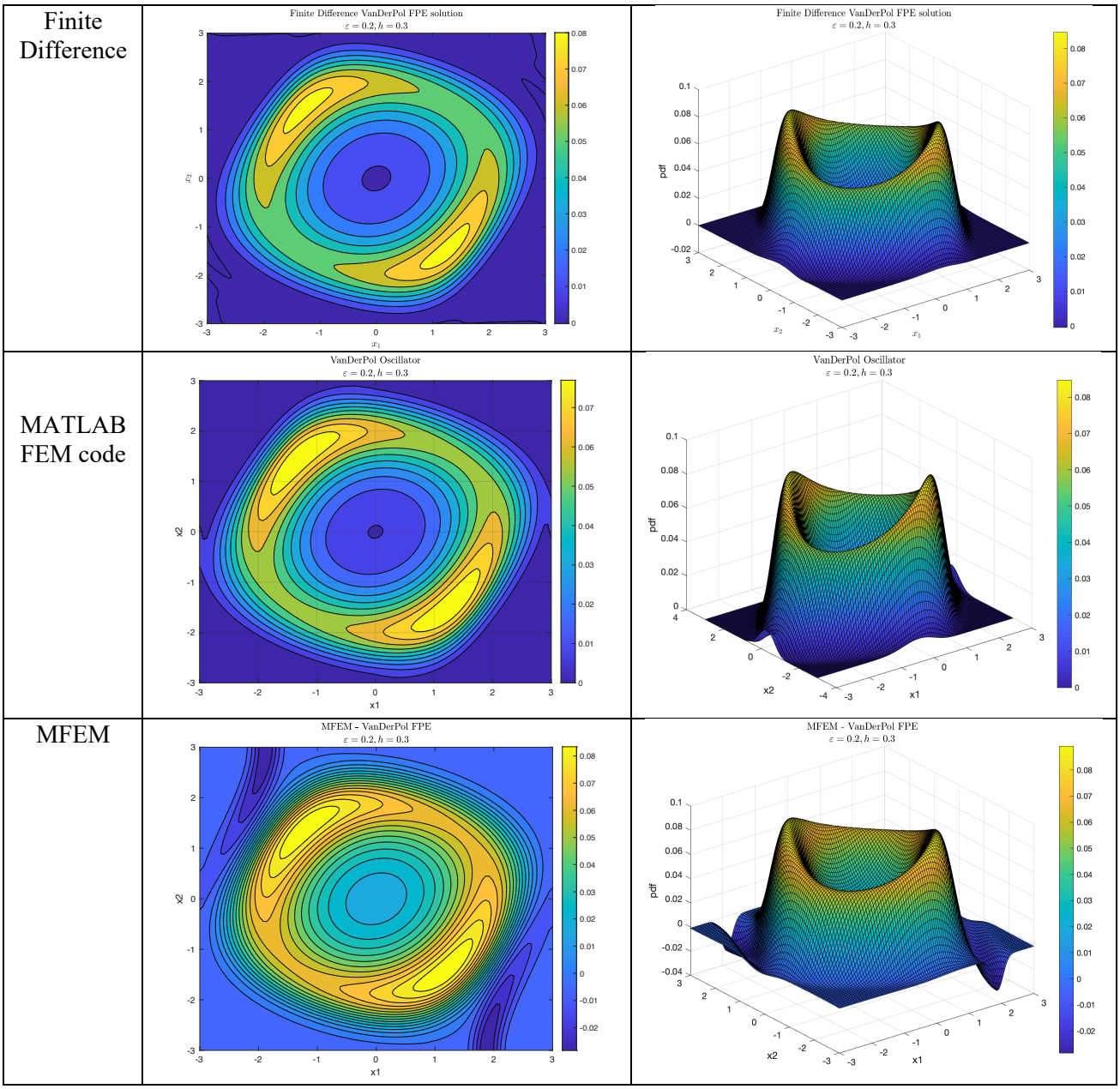


Figure 11. FPE null space solution to VanDerPol Oscillator, $\varepsilon = 0.2, h = 0.3$

From these graphs, some initial observations can be made. It can clearly be noticed how diffusion dominates as the diffusion coefficient increases. Increasing h, from 0.1 to 0.3, the probability density is a lot more spread and smeared out. Due to extensive diffusion, the maximum probability density value is also seen to drop. So, with stronger multiplicative noise affecting the system, the probability of finding the state variables within the limit cycle is smeared out.

An analytical solution to Van Der Pol equation can be obtained if the parameters and the initial condition belong to the region of attraction which eventually takes it to its limit cycle. Multiple approaches have been taken to find the analytical solution to its various limit cycles. There are some oscillators whose Fokker-Planck formulations have analytical solutions. This has been discussed in [6], and an analytical solution has been developed for duffing type oscillators.

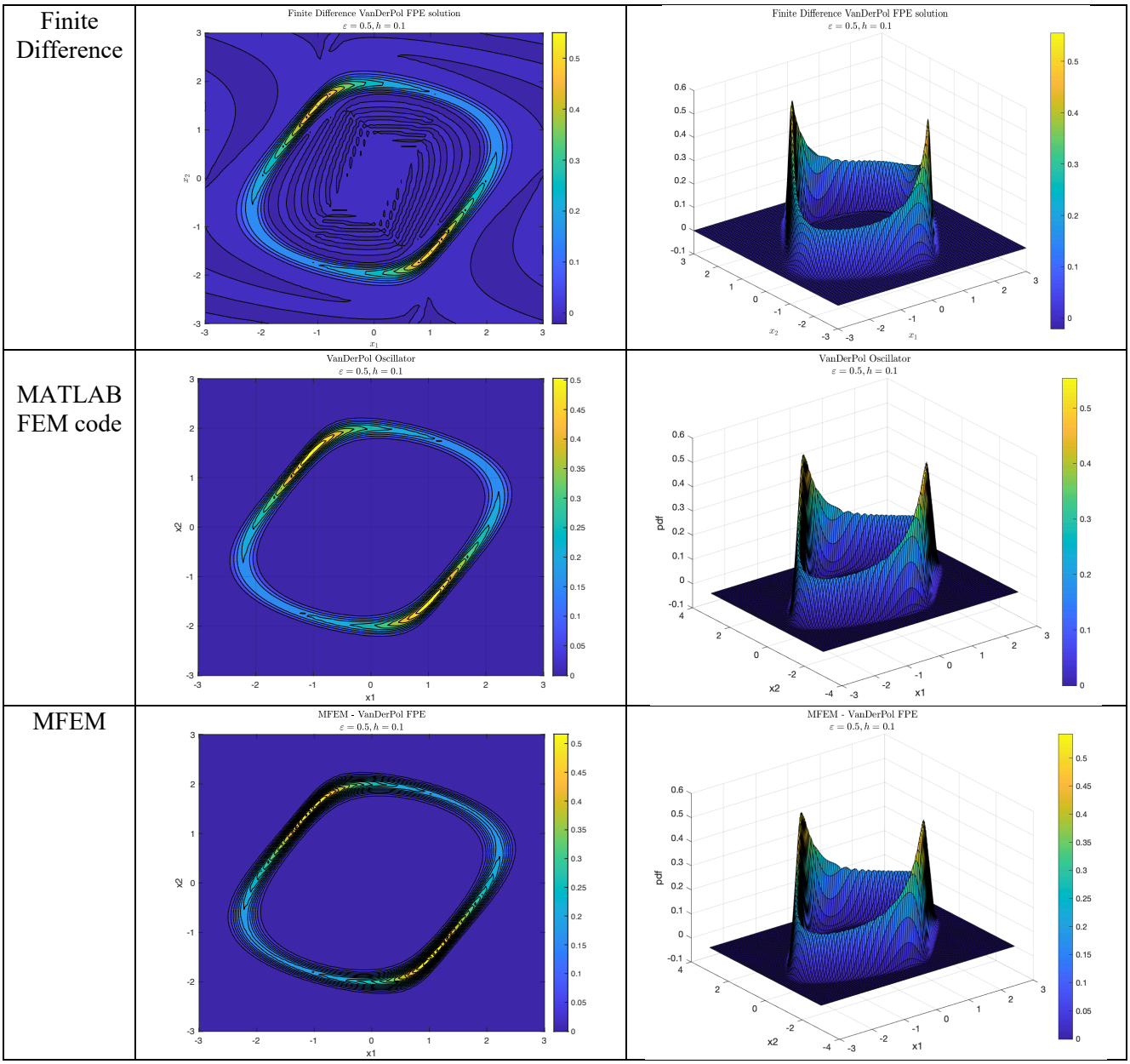


Figure 12. FPE null space solution to VanDerPol Oscillator, $\varepsilon = 0.5, h = 0.1$

[6], states that the probability density for a duffing oscillator with dynamics given by:

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix} = \begin{bmatrix}x_2\\ -2\omega_n\xi x_2 - \omega_n^2\gamma x_1 - \beta x_1^3\end{bmatrix} + \begin{bmatrix}0 & 0\\ 0 & \pi^{-0.5}\end{bmatrix}\begin{bmatrix}0\\ w(t)\end{bmatrix}$$

Stochastic forcing with white noise of variance $\sigma = 1/\pi$.

Equation 18

Its analytical solution is given by Equation 19.

$$p(x_1, x_2) = C \exp\left\{-2\xi\omega_n\left[0.5x_2^2 + \frac{1}{2}\omega_n^2\gamma x_1^2 + \frac{1}{4}\omega_n^2\beta x_1^4\right]\right\}$$

C – normalization constant.

Equation 19

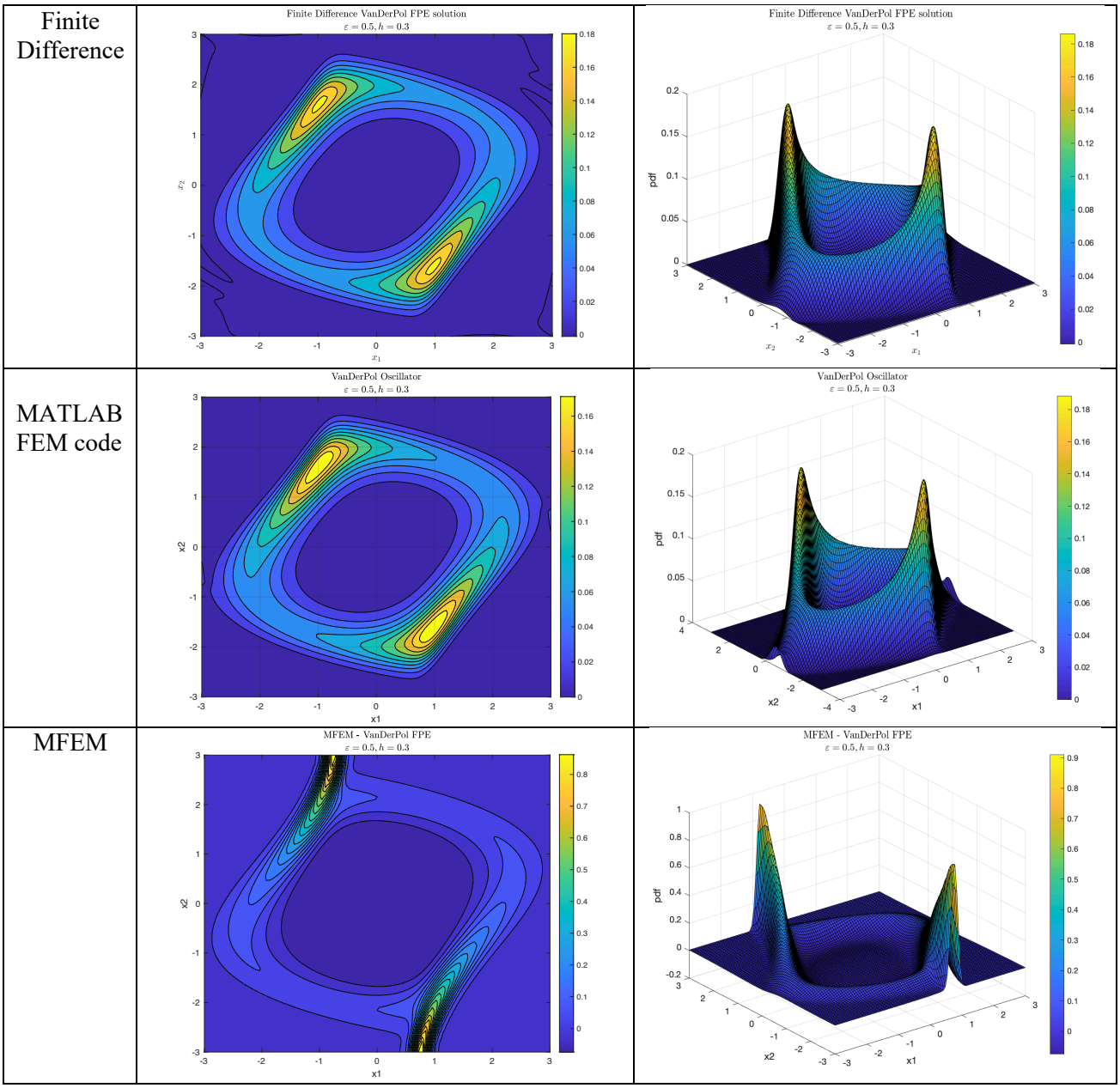


Figure 13. FPE null space solution to VanDerPol Oscillator, $\varepsilon = 0.5, h = 0.3$

A Duffing Oscillator was also designed to check the performance of my Finite Element MATLAB code with the analytical solution, and it is discussed in future sections.

Looking at the case of $\varepsilon = 0.5, h = 0.3$, it is quite evident that the MFEM result does not agree with the Finite Difference, or the FEM code written in MATLAB. There is one possible explanation as to why the results disagree. In the MATLAB FEM code written, the element transformation and the Jacobian is calculated for each element whereas in MFEM that is not the case. Performing Element Transformation and computing the Jacobian and its determinant is a very costly step and hence it is usually only computed only if there is an absolute necessity. This improves the overall computational efficiency and saves on computation time as well. This is shown in Table 4. MFEM code executes the fastest and the MFEM code on MATLAB executes the slowest. It is important to find out if the MATLAB code is more accurate than MFEM in order to justify the computation of Jacobian for each element and in every iteration.

Finite Difference Code	16.1513362 sec
MATLAB FEM code	198.211876 sec
MFEM	7.11238873 sec

Table 4. Computational time of the 3 Numerical Methods

The oversimplification of not computing Jacobian can prove to be costly in cases where diffusion weak form needs to be integrated. This can be seen in the MFEM solution on Figure 13. Diffusion weak form requires proper element transformation and Jacobian calculation.

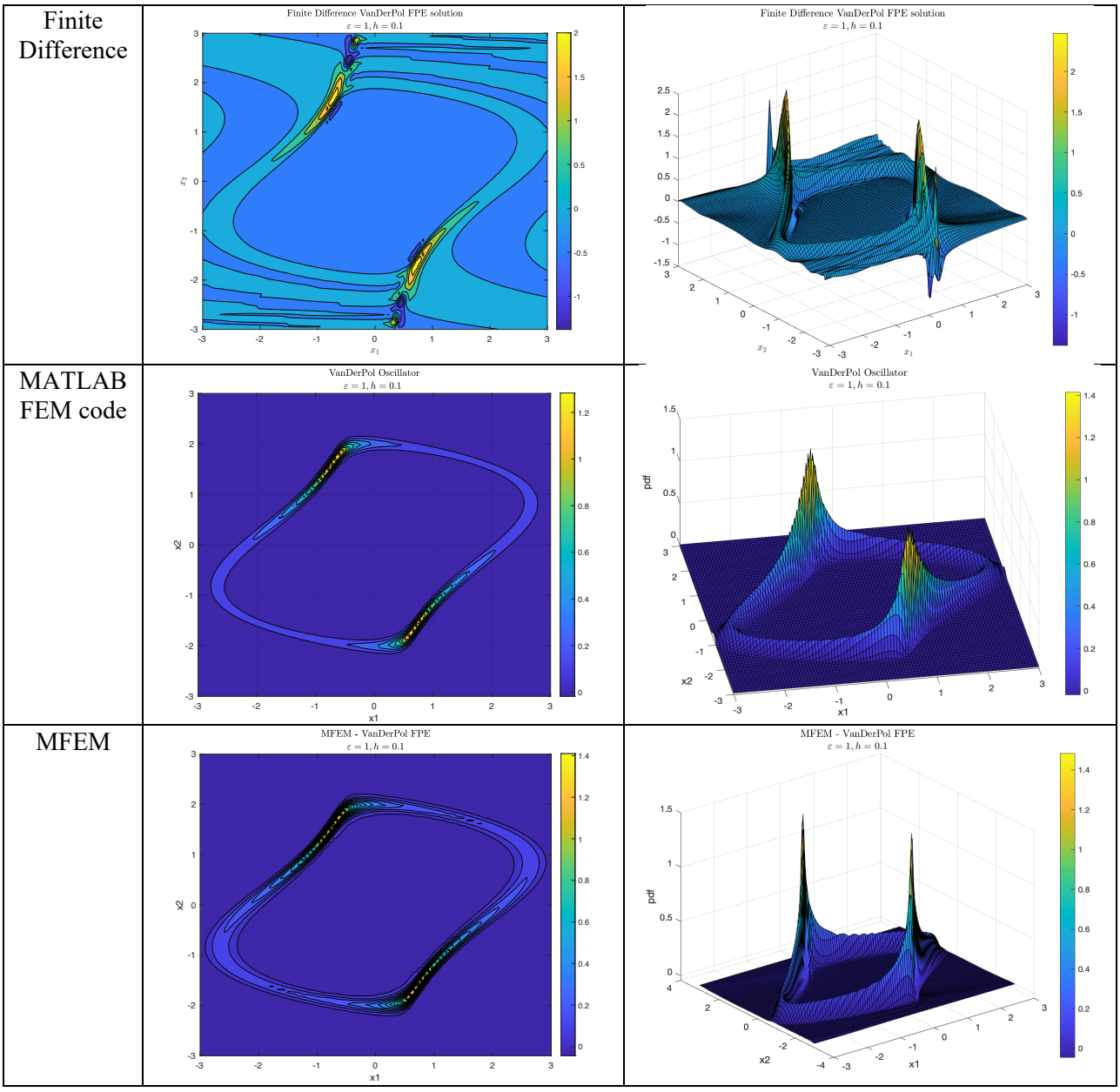


Figure 14. FPE null space solution to VanDerPol Oscillator, $\varepsilon = 1, h = 0.1$

The Finite Difference code is not doing well in this case where non-linearity dominates ($\varepsilon = 1$) with significantly less stochastic forcing (Figure 14). This shows how finite difference approximation for

convection with regions having extremely high nonlinearity can cause stability issues and hence lead to inaccurate results.

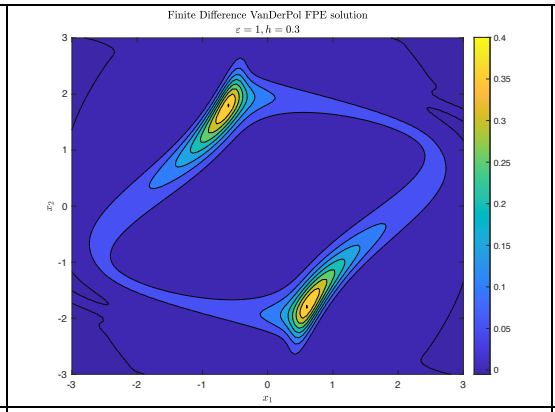
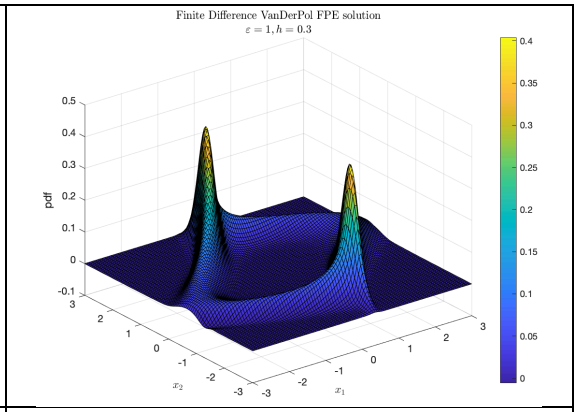
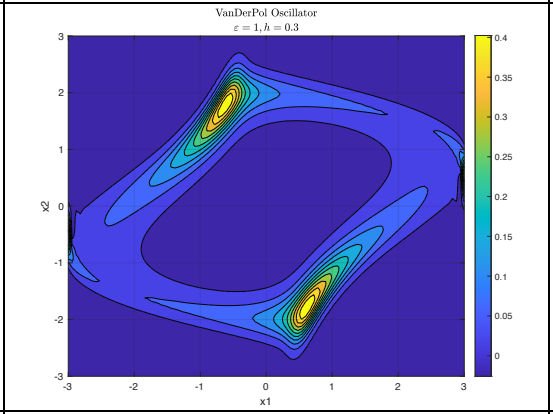
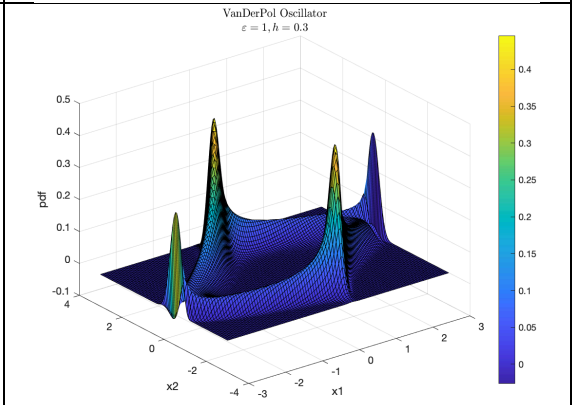
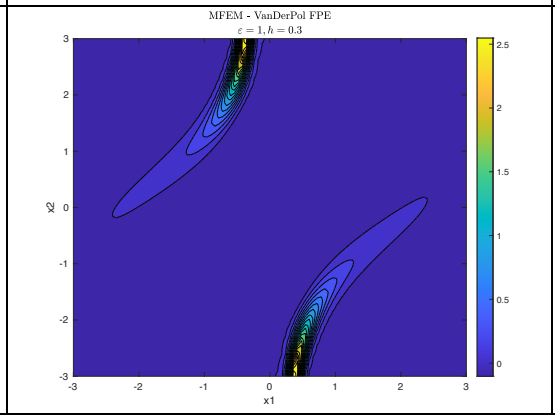
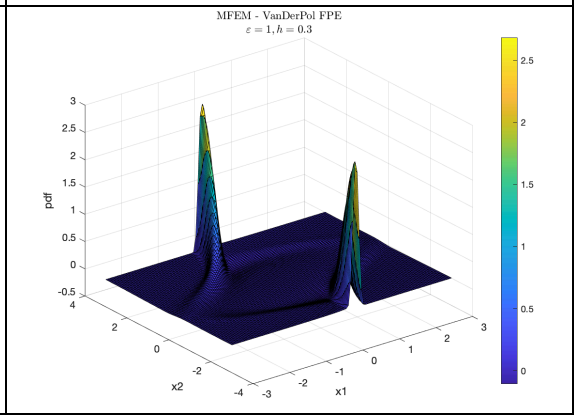
Finite Difference		
MATLAB FEM code		
MFEM		

Figure 15. FPE null space solution to VanDerPol Oscillator, $\varepsilon = 1, h = 0.3$

As it can be noticed, MFEM performs the worst in cases with higher stochastic forcing ($h = 0.3$) applied to the system of nonlinear equations. This is because of the diffusion integrator performing a poor job due to extensive Jacobian approximations. Second-order accurate Finite Differencing is finding it extremely difficult to handle cases with high non-linearity ($\varepsilon = \{1,0.5\}, h = 0.1$) involved.

To analyze the performance of the FEM code written on MATLAB, it is important to compare its results with an analytical result for a system of Nonlinear Ordinary Differential Equations – for example the Duffing oscillator explained in Equation 18 and Equation 19. The comparison is performed by computing the analytical results and the numerical results and finding the norm of the error between the two results. The pictures below show the comparison of MATLAB FEM code and the analytical results.

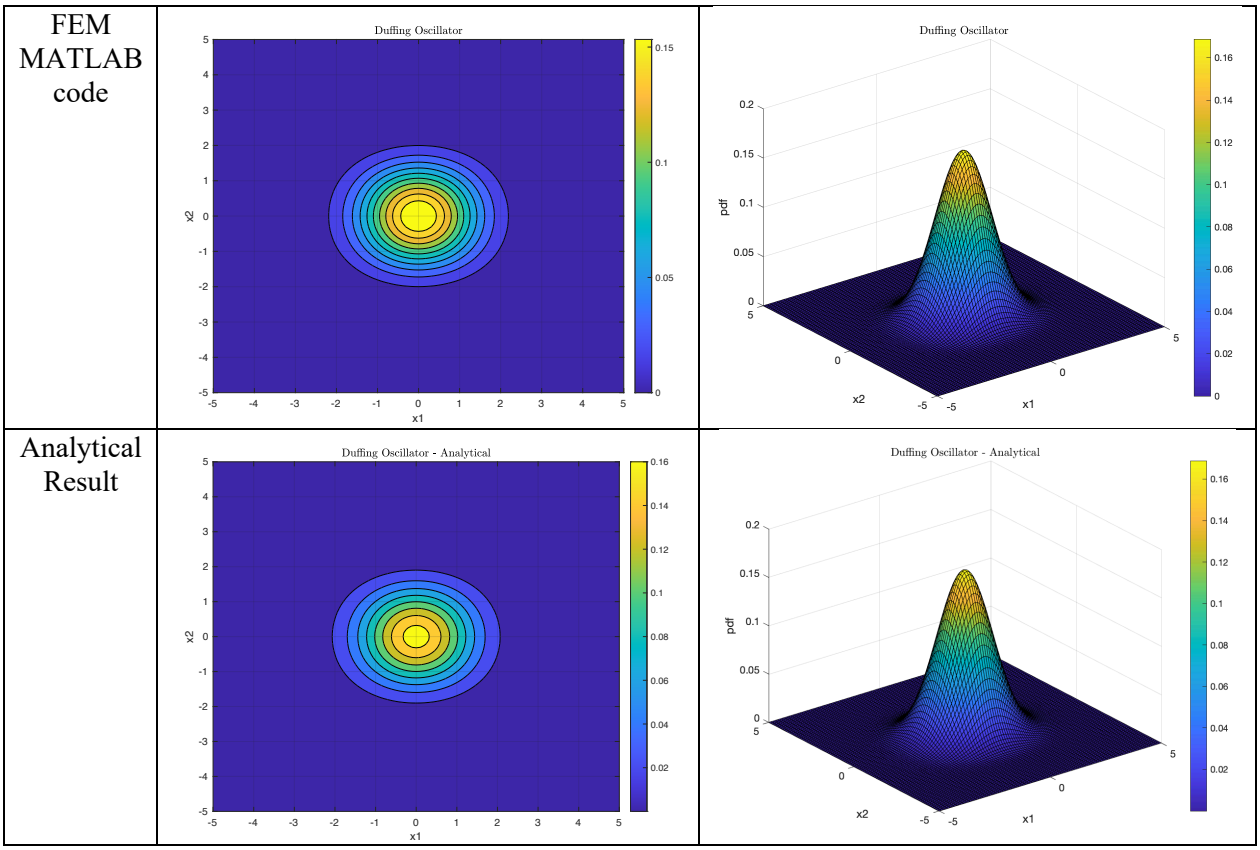


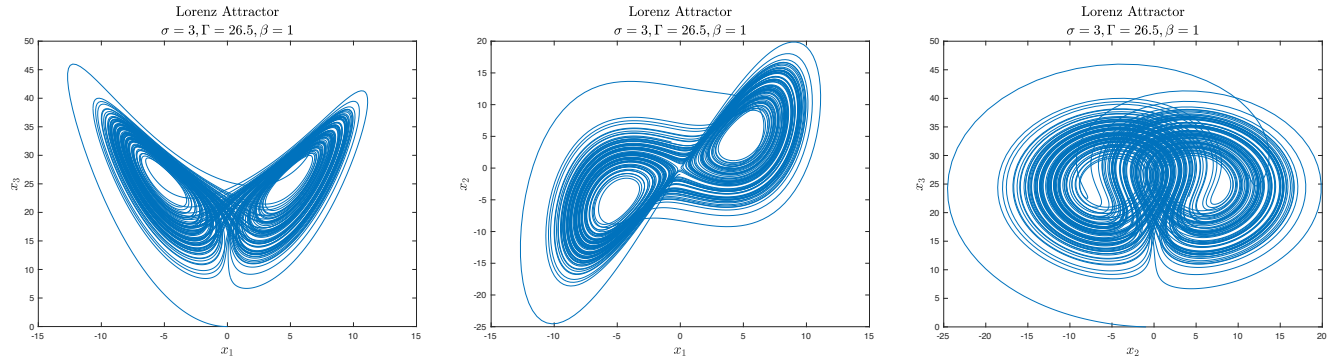
Figure 16. MATLAB FEM - FPE solution comparison with Analytical FEM solution for a Duffing Oscillator

The norm of the error between the MATLAB FEM code and the analytical result is 0.001310630881520. This is a testament to the accuracy achieved through the MATLAB FEM code. Hence, the observations and the comparison made about the behavior of MFEM code or Finite Difference methods with the MATLAB code is justified.

One way to improve results for MFEM for diffusion dominated problems is to increase the order of the polynomials used. This will also increase the computation time as there are more nodes for the same number of elements, but the ratio of rise in accuracy compared to computation time is very high, so it is justified. If the mesh is refined a little more and if the finite difference order is also increased from second to third or fourth order, the finite difference approach can perform better in areas of high nonlinearity. Given the computational time it takes to perform a 2D FPE solution in MATLAB FEM code, MFEM was preferred over the MATLAB code for the 3D case.

3D Fokker-Planck Equation

Simulation of the 3D FPE of the Lorenz Attractor is the next step in this project. MFEM code to generate the Sparse Global stiffness matrix can be found in the appendix. Unfortunately, I did not have the computational power in my computer to successfully get solution to the 3D Lorenz attractor using Linear iterative solvers like $gmres()$. A couple of reasons could be the root cause to this issue. I ~~couldn't land on~~ the right preconditioner to use on the sparse matrices. The domain of the 3D case is a lot bigger than the 2D case and it can be seen from the deterministic solution to the Lorenz attractor shown in Figure 17.



a) *b)* *c)*

Figure 17. Deterministic solution to 3D FPE- Lorenz Equation

A fine discretization of the 3D Lorenz system produces a huge sparse matrix and solving this with a proper preconditioner is very difficult. The solution is not unattainable as it has been done in [2]. A computer with better computational capability and a better understanding of computational linear algebra is required to arrive at a solution.

Future Work

First step is to arrive at a solution to the 3D FPE solution of the Lorenz attractor. Once this is done, an adjoint code should be developed to compute the rate of change of the null-space solutions with respect to the parameters that define the nonlinearity of the ODEs. This is useful for applications like computing the sensitivities of objectives that has perturbations in it, like turbulence. A Fokker-Planck formulation is a cheaper way to compute these sensitivities and a detailed explanation is provided in [7]. Its applications are extensive in the field of climate sensitivity analysis. Its applications can be tested in the field of aerodynamic shape optimization where the cost of the optimization depends on chaotic dynamics like turbulence. It is an area of active research, and this method could be a cheap way of formulating the objective compared to currently used cost functions.

Bibliography

[1] R. K. Jirí Náprstek, "Finite element method analysis of Fokker–Plank equation in stationary and evolutionary versions," *Advances in Engineering Software,* no. 72, pp. 28-38, 19 July 2013.

[2] J. B. M. Altan Allawala, "Statistics of the stochastically-forced Lorenz attractor by the Fokker-Planck equation and cumulant expansions," *Physical Review E,* vol. 94, no. 5, 23 November 2016.

[3] M. A. B. L. Pichler L., "Numerical Solution of the Fokker–Planck Equation by Finite Difference and Finite Element Methods—A Comparative Study," *Computational Methods in Stochastic Dynamics,* vol. 26, pp. 69-85, 2013.

[4] N. S. Kumar P., "Solution of Fokker-Planck equation by finite element and finite difference methods for nonlinear systems," *Sadhana,* vol. 31, pp. 445-461, 2006.

[5] T. J. Hughes, The Finite Element Method, Linear Static and Dynamic Finite Element Analysis, Garden City, New York: Dover Publications, 1987.

[6] C. S. H, "Perturbation techniques for random vibration of nonlinear systems.," *The Journal of the Acoustical Society of America,* vol. 35, 1963.

[7] T. J., "Climate sensitivities via a Fokker–Planck adjoint approach," *Quarterly Journal of the Royal Meterorological Society,* vol. 131, no. 73, p. 92, 2005.

Appendix

1-D Finite Element MATLAB code

```
function [rho,rho_a] = FPE_1D(Nelem,h)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This is script to solve the 1D Fokker Planck equation using Finite Elements
% 1D FPE: Drho/Dt = -a*rho + h^2/2 (D2rho/Dx^2)
% Assume boundary conditions = 0 on either end of it
% Inputs: Nelem : Number of elements on the mesh
%         h     : The diffusion coefficient parameter
% Output: rho   : FEM solution to FPE
%         rho_a : Analytical solution of FPE

x1_lim1 = -3;
x1_lim2 = 3;
mesh = generate1Dmesh(Nelem,x1_lim1,x1_lim2);

order = 1;
c_coeff = 1;
fespace = FiniteElementSpace(mesh,order);

% Go through each element and compute Bilinear forms - diffusion and
% convection
% mass integrator choice -1 , convection - 2, diffusion - 3

Nnodes = fespace(end).ElemDOF(end);

D = single(zeros(Nnodes)); % diffusion overall matrix
C = single(zeros(Nnodes)); % convection overall matrix
W = single(zeros(Nnodes));

for i=1:Nelem
    d = Diffusion_Integrator(-h^2/2,order,fespace(i));
    c = Convection_Integrator(c_coeff,order,fespace(i));
    e = fespace(i).ElemDOF(1);
    D = Assemble(D,d,e);
    C = Assemble(C,c,e);
end



for i=1:Nnodes
    if i == 1 || i == Nnodes
        w(1,i) = 0.5 * (x1_lim2-x1_lim1)/Nnodes;
    else
        w(1,i) = (x1_lim2-x1_lim1)/Nnodes;
    end
end

for i=1:Nnodes
    W(i,:) = w;
end

A = D+C+W;
X = ones(Nnodes,1);

rho = A\X;

% plotting
R = @(x) sqrt(3/(pi*h^2))*exp((-3*x.^2)/(h^2));
```

```
x = (x1_lim1:(x1_lim2-x1_lim1)/(length(rho)-1):x1_lim2)' ;
rho_a = R(x);
plot(x,rho,'r','lineWidth',2);
hold on;
grid on;
plot(x,rho_a,'k--','lineWidth',2);
xlabel('x');
ylabel('$\rho$','Interpreter','latex');
legend('FEM approximate solution','Exact solution');
title('FEM solution to 1D FPE','a = 3, h = 0.8');

end
```

```
function MeshData = generate1Dmesh(Nelem,x1_lim1,x1_lim2)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This function generates a 1D mesh.
% Inputs: Nelem      : Number of elements required on the 1D mesh
%         x1_lim1    : the lower limit of the domain considered
%         x2_lim2    : the upper limit of the domain considered
% Output: MeshData   : This is a structure that contains all the necessary
%                      information about the domain being meshed
%         dim        : The dimension of the domain - 1D
%         num_elem   : Number of elements present in the domain
%         num_node   : Number of nodes present in the domain
%         DOF        : Array containing the DOF ID of each node in the mesh
%         BoundaryDOF: Array containing the DOF assigned to the boundary nodes
%         GridFn     : A cell array containing the spatial location of the
%                      nodes present in each element

    x = x1_lim1: (x1_lim2 - x1_lim1)/Nelem : x1_lim2;

    Nnodes = Nelem + 1;
    GridFn = cell(1,Nnodes);
    DOF = zeros(1,Nnodes);
    boundary_dof = zeros(1,2);
    b_dof = 1;


    for i=1:Nnodes
        DOF(i) = i;
        GridFn{i} = x(i); % storing the position data in a GridFn
        if i==1 || i==Nnodes
            boundary_dof(b_dof) = i;
            b_dof = b_dof + 1;
        end
    end

    MeshData.dim = 1;
    MeshData.num_elem = Nelem;
    MeshData.num_node = Nnodes;
    MeshData.DOF = DOF;
    MeshData.BoundaryDOF = boundary_dof;
    MeshData.GridFn = GridFn;

end
```

```
function [FE_space] = FiniteElementSpace(mesh,order)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This function is used to generate the Finite Element Space assigning each
% element with its corresponding nodal DOF and spatial grid function
% Inputs : mesh     : An input structure of the mesh information of domain
%          order    : The order of interpolating polynomials to generate
% Output : FE_space : A structure that contains the DOF and GridFn
%                     information for each element in the mesh with added
%                     node points based on the polynomial order required
%          ElemDOF  : Array containing the DOF of each node in the element
%          LocDOF   : Cell containing location of the DOF in space

    Nelem = mesh.num_elem;

    % FE_space stores the following:
    % Element ID
    % DOF attached to Element ID (ElemDOF)
    k = 1;
    for i=1:Nelem
        FE_space(i).ID = i;
        t = zeros(order+1,1);
        for j=1:order+1
            t(j,1) = k+j-1;
        end
        FE_space(i).ElemDOF = t; k = k+order;
        int_pt = ComputeIntGridPt(mesh.GridFn{i},mesh.GridFn{i+1},order);
        FE_space(i).LocDOF = [mesh.GridFn{i};int_pt;mesh.GridFn{i+1}];
    end

end
```

```
function int_pt = ComputeIntGridPt(st_pt,end_pt,order)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This function is used to generate points internal to the element
% depending on the order of polynomial of shape functions
% Inputs: st_pt  : starting node point
%         end_pt : ending node_pt
% Output: int_pt : internal point in space newly generated

    % points to be added
        n = order - 1;
        dn = (end_pt - st_pt)/order;
        int_pt = [];
        for i=1:n
            int_pt = st_pt + i*dn;
        end


end
```

```
function D_sub = Diffusion_Integrator(diff_coeff,order,fespace)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This funciton performs Diffusion integration over the element and
% generates the element stiffness matrix for this Bilinear operation.
% Inputs : diff_coeff : The co-efficient of diffusion in governing equation
%          order      : Order of polynimial degree used for interpolation
%          fespace    : Elements finite element space structure that
%                       contains its DOF array and gird function of its nodes
% Output : D_sub      : Element Stiffness matrix for the diffusion bilinear
%                       operation

    LocalGrid = fespace.LocDOF;
    len = length(LocalGrid);
    D_sub = zeros(len);
    choice = 3; % diffusion

    for i=1:len
        for j=1:len
            fIdx = [i j];
            f = Eval_ShapeFn(fIdx,order,choice);
            val = NumInt(f,order,LocalGrid,choice);
            D_sub(i,j) = diff_coeff*val;
        end
    end

end
```

```
function C_sub = Convection_Integrator(conv_coeff,order,fespace)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This funciton performs Convection integration over the element and
% generates the element stiffness matrix for this Bilinear operation.
% Inputs : conv_coeff : The co-efficient of convection in governing equation
%          order      : Order of polynimial degree used for interpolation
%          fespace    : Elements finite element space structure that
%                       contains its DOF array and gird function of its nodes
% Output : C_sub      : Element Stiffness matrix for the convection bilinear
%                       operation

    LocalGrid = fespace.LocDOF;
    len = length(LocalGrid);
    C_sub = zeros(len);
    choice = 2; % convection

    for i=1:len
        for j=1:len
            fIdx = [i j];
            f = Eval_ShapeFn(fIdx,order,choice);
            val = NumInt(f,order,LocalGrid,choice);
            C_sub(i,j) = conv_coeff*val;
        end
    end
end
```

```
function fval = Eval_ShapeFn(fIdx,order,choice)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This function evaluates the Shape functions at the integration points
% chosen for the element. It performs this operation for bilinear
% operations, Diffusion and Convection. For Diffusion, the derivative of
% the shape functions are evaluated and for convection, a derivative and a
% shape function is evaluated.
% Inputs : fIdx   : Takes the indices of the the shape functions to be
%                   evaluated at the integration points
%          order  : order of interpolating polynomials to use
%          choice : The choice of integration operation to perform -
%                   Diffusion or convection
% Output : fval   : Array of function evaluations at integration points for
%                   the required indices

    [ShapeFn,DShapeFn] = H1_FECollection(order);
    [Quad_pts,~] = IntRules();
    fval = zeros(length(fIdx),length(Quad_pts));

    if choice == 2 % convection
        for i=1:2
            for j=1:length(Quad_pts)
                if i == 1
                    fval(i,j) = DShapeFn{fIdx(i)}(Quad_pts(j));
                else
                    fval(i,j) = ShapeFn{fIdx(i)}(Quad_pts(j));
                end
            end
        end
    end

    if choice == 3 % diffusion
        for i=1:2
            for j=1:length(Quad_pts)
                fval(i,j) = DShapeFn{fIdx(i)}(Quad_pts(j));
            end
        end
    end

end
```

```
function [Quad_pts, Quad_wts] = IntRules()
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This function outputs the integration rule used in this project
% Outputs: Quad_pts : An array containing the location of the integration
%                     points in the \zeta coordinate system
%          Quad_wts : An array containing the weights associated to each
%                     integration point in the \zeta coordinate system

    Quad_pts = [-1/sqrt(3) 1/sqrt(3)];
    Quad_wts = [1 1]';

end
```

```
function [ShapeFn, DShapeFn] = H1_FECollection(order)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This function outputs a cell array containing the shape functions to use
% based on the order of polynimial to use. These are H1 continuous Lagrange
% polynomials. This list can be updated for higher order polyninomials.
% Inputs : order   : order of polynomial required
% Outputs: ShapeFn : A cell array containing the Shape Functions at nodes
%          DShapeFn: A cell array containing the derivative of Shape
%                    Functions at nodes.

    if order == 1
        ShapeFn{1} = @(eta) 0.5 * (1 - eta);
        ShapeFn{2} = @(eta) 0.5 * (1 + eta);

        DShapeFn{1} = @(eta) -0.5;
        DShapeFn{2} = @(eta) 0.5;
    elseif order == 2
        ShapeFn{1} = @(eta) 0.5 * eta .* (eta - 1);
        ShapeFn{2} = @(eta) 1 - eta.^2;
        ShapeFn{3} = @(eta) 0.5 * eta .* (eta + 1);

        DShapeFn{1} = @(eta) 0.5 * (2*eta - 1);
        DShapeFn{2} = @(eta) -2 * eta;
        DShapeFn{3} = @(eta) 0.5 * (2*eta + 1);
    else
        disp("Orders higher than 2 not supported as of now! sorry");
        ShapeFn{1} = @(eta) 0;

        DShapeFn{1} = @(eta) 0;
    end

end
```

```
function flux = Calc_Flux(GlobalPt)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This function evaluates the flux at the global point location of the
% integration points of the element.
% Input : GlobalPt : The point where flux needs to be computed
% Output: fulx     : Evaluated flux at the global point location

    par = 3;
    flux = -par*GlobalPt;
end
```

```
function I = NumInt(g,order,LocalNp,choice)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This function performs the numerical integration over the element space
% for the function array sent as input.
% Inputs : g       : The function array evaluated at integration points for
%                    the appropriate shape functions
%          order   : order of polynomials used in the function evaluations
%          LocalNp : The Local node locations for the shape functions that
%                    are attached and evaluated at the nodes of the element
%          choice  : choice of integration performed - diffusion or
%                    convection
% Outputs: I       : Integrated value

    [Quad_pts,Quad_wts] = IntRules();
    [m,n] = size(g);
    s = 0;
    for i=1:n
        p = 1;
        for j=1:m
            p =  p* g(j,i);
        end
        if choice == 2 % convection

            [pt,~] = ElementTransformation(order,LocalNp,Quad_pts(i));
            flux = Calc_Flux(pt);
            s = s + Quad_wts(i)*p*flux;

        elseif choice == 3 % diffusion

            [~,jac] = ElementTransformation(order,LocalNp,Quad_pts(i));
            s = s + Quad_wts(i)*(1/jac)*p;

        else % as of now
            s = s + Quad_wts(i)*(1/jac)*p;
        end
    end

    I = s;
end
```

```
function [pt,Jacobian] = ElementTransformation(order,LocalNp,loc_intPt)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This function performs transformation of the evaluated shape functions to
% find the jacobian and the location of the global point in the local
% coordinate (\zeta) system.
% Inputs : order     : order of polynomials used to perform integration
%          LocalNp   : Local nodal locations of the shape functions (attached
%                      to nodes) are evaluated at.
%          loc_intPt : Local Integration point at which the shape function
%                      is evaluated at.
% Outputs: pt        : The transformed point from global location to the
%                      local \zeta coordinate system
%          Jacobian  : The evaluated Jacobian of this element
%                      transformation

    [ShapeFn,DShapeFn] = H1_FECollection(order);
    J = 0; p = 0;
    for i=1:length(DShapeFn)
        p = p + ShapeFn{i}(loc_intPt)*LocalNp(i);
        J = J + DShapeFn{i}(loc_intPt)*LocalNp(i);
    end

    zero_tol = 1e-4;
    if J <= zero_tol
        J = zero_tol; % avoid jacobian to become 0
    end

    pt = p;
    Jacobian = J;
end
```

```
function M_out = Assemble(M,m_sub,e)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This function is used to Assemble the local Element Stiffness matrix in
% the right places on the Global Stiffness matrix
% Inputs : M     : Input Global Matrix to modify
%          m_sub : Local Element Stiffness matrix that should be integrated
%                  to the Global Stiffness matrix
%          e     : the element ID being handled
% Outputs: M_out : The modified Global Stiffness matrix

    len = length(m_sub);
    M_out = M;

    for i=1:len
        for j=1:len
            M_out(e+i-1,e+j-1) = M_out(e+i-1,e+j-1) + m_sub(i,j);
        end
    end

end
```

2D Finite Element code MATLAB

Contents


```
% Script to do FEM Analysis on VanDerPol FPE
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 30, 2021$
% $Code Version: 1.1$
% Does Partial Assembly process while assembling Element Stiffness Matrix
% VanDerPol FPE : 0 = -(f1\rho),x1 -(f2\rho),x2 + (h^2/2)[(rho),x1x1 + (rho),x2x2]
% f1 = x2, f2 = -x1 + \epsilon*(1-x1^2)*x2;
clc
clear all

% generate Rectangular mesh
mesh = generateRecMesh(0.06,0.06,-3,3,-3,3);

% specify order of Finite Elements to use
order = 1;

% Generate Finite Element Space
fespace = FiniteElementSpace(mesh,order);

% parameter used in VanDerPol FPE
eps = 0.2;
```

Generate Stiffness Matrix

```
NumElem = mesh.num_elem;
NumNode = fespace(end).ElemDOF(end);

% computing B and S matrix in z-n coordinate system (all integration points)
[B_zn,S_zn] = Eval_ShapeFn(order);

% multiplicative noise covariance value
h = 0.3;
I1 = []; J1 = []; V1 = [];
% generate sparse matrix of size (NumNode+1) x (NumNode+1)
M = sparse(NumNode+1,NumNode+1);
k = 1;
for i=1:NumElem
    n = length(fespace(i).ElemGrid);
    LocGrid = fespace(i).ElemGrid;
    LocGridArr = zeros(n,2); % 2D

    for j=1:n
        LocGridArr(j,:) = LocGrid{j};
    end

    d = DiffusionIntegrator(-h^2/2,B_zn,LocGridArr);
    c = ConvectionIntegrator(-1,B_zn,S_zn,LocGridArr,eps);
    [i_a,j_a,v_a] = Assemble_NoBC(c+d,fespace(i));
    I1 = [I1;i_a];
    J1 = [J1;j_a];
    V1 = [V1;v_a];
```

```
    if (k > 100)
        M = partialAssemble(M,I1,J1,V1);
        I1 = []; J1 = []; V1 = [];
        k = 0;
    end
    k = k + 1;
end

M = partialAssemble(M,I1,J1,V1);

W = CalcTrapzWts(mesh);
M(NumNode+1,1:NumNode) = W;
M(1:NumNode,NumNode+1) = W';
```

Generate RHS

```
RHS = [zeros(NumNode,1);1];
tol = 1e-6;
X = gmres(M,RHS,[],tol,5000);
%X = M\RHS;

rho = reshape(X(1:NumNode),mesh.DimLen(1),mesh.DimLen(2));
```

Plotting

```
[X,Y] = meshgrid(-3:mesh.DX(1):3,-3:mesh.DX(2):3);

figure
surf(X,Y,rho);
colorbar
xlabel('x1');
ylabel('x2');
zlabel('pdf');
title('VanDerPol Oscillator','$\varepsilon = 0.2, h = 0.3$',...
    'Interpreter','latex');

figure
contourf(X,Y,rho,10);
grid on
colorbar
xlabel('x1');
ylabel('x2');
title('VanDerPol Oscillator','$\varepsilon = 0.2, h = 0.3$',...
    'Interpreter','latex');
```

```
function MeshData = generateRecMesh(dx1,dx2,x1_lim1,x1_lim2,x2_lim1,x2_lim2)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 21, 2021$
% $Code Version: 1.0$
% Inputs: dx1 - discretization along x1-axis
%         dx2 - discretization along x2-axis
%         x1_lim1 - lower limit of x1 dimension of the domain to mesh
%         x1_lim2 - upper limit of x1 dimension of the domain to mesh
%         x2_lim1 - lower limit of x2 dimension of the domain to mesh
%         x2_lim2 - upper limit of x2 dimension of the domain to mesh
% Outputs: struct mesh
%          mesh.dim        - holds the dimension of the domain = 2
%          mesh.num_elem   - Number of rectangular elements present in the
%                            domain
%          mesh.num_node   - Number of nodal elements present in the domain
%          mesh.DOF        - 2D Matrix with each node holding its DOF value
%          mesh.CornerDOF  - 1D array holding the DOF values of domain
%                            corners
%          mesh.BoundaryDOF- 1D array holding the DOF values of domain
%                            boundary
%          mesh.GridFn     - 2D cell array with each cell holding the
%                            domain location of each nodal DOF
%          mesh.DimLen     - 1x2 array that holds total number of points
%                            along x1 and x2 direction
%          mesh.DX         - [dx1 dx2]: discretization along x1 and x2

    % Generate [X1 X2] - Values along which to generate rectangular mesh
    x1 = x1_lim1:dx1:x1_lim2;
    x2 = x2_lim1:dx2:x2_lim2;

    % dimension of domain
    dim = 2;

    Nnodes = length(x1)*length(x2);
    Nelem = (length(x1)-1)*(length(x2)-1);

    MeshDOF = zeros(length(x2),length(x1));
    GridFn = cell(length(x2),length(x1));
    k = 1;
    b_dof = 1;
    c_dof = 1;
    corner_dof = zeros(2*dim,1);
    per = 2*(length(x1)-1) + 2*(length(x2)-1);
    boundary_dof = zeros(per,1);
    for i=1:length(x2)
        for j=1:length(x1)
            MeshDOF(i,j) = k;
            GridFn{i,j} = [x1(j),x2(i)];
            if i==1 && j==1 % corner 1
                corner_dof(c_dof) = k;
                c_dof = c_dof + 1;
            elseif i==1 && j== length(x1) % corner 2
                corner_dof(c_dof) = k;
                c_dof = c_dof + 1;
            elseif i==length(x2) && j==1 % corner 3
                corner_dof(c_dof) = k;
                c_dof = c_dof + 1;
            elseif i== length(x2) && j==length(x1) % corner 4
                corner_dof(c_dof) = k;
```

```
                c_dof = c_dof + 1;
            end
            if i==1
                boundary_dof(b_dof) = k;
                b_dof = b_dof + 1;
            elseif j==1 || j==length(x1)
                boundary_dof(b_dof) = k;
                b_dof = b_dof + 1;
            elseif i == length(x2)
                boundary_dof(b_dof) = k;
                b_dof = b_dof + 1;
            end

            k = k + 1;
        end
    end
    MeshData.dim = dim;
    MeshData.num_elem = Nelem;
    MeshData.num_node = Nnodes;
    MeshData.DOF = MeshDOF;
    MeshData.CornerDOF = corner_dof;
    MeshData.BoundaryDOF = boundary_dof;
    MeshData.GridFn = GridFn;
    MeshData.DimLen = [length(x1) length(x2)];
    MeshData.DX = [dx1 dx2];
end
```

```
function fespace = FiniteElementSpace(mesh,order)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 21, 2021$
% $Code Version: 1.0$
% This function is used to generate the Finite Element Space for order 1
% polynomials. Higher order polynomials cant be handled in this code.
% Inputs : mesh - structure mesh which holds all mesh information generated
%                 using generateRecMesh function
%          order - order of polynomial used for Finite Elements
% Outputs: fespace         - structure
%          fespace.Element - Holds Element ID
%          fespace.ElemDOF - 1D array which holds the DOF of the nodes
%                            attached to the element
%          fespace.ElemGrid- 2D array which holds the GridLocation of the
%                            nodes attached to the element
% this code considers only rectangular elements
% if order is increased, it will add more nodes to the mesh.
    [m,n] = size(mesh.DOF);
    Nodes = mesh.num_node;
    k = 1;
    for i=1:m-1
        for j=1:n-1
            LocalDOF(1,1) = mesh.DOF(i,j);
            LocalDOF(2,1) = mesh.DOF(i,j+1);
            LocalDOF(3,1) = mesh.DOF(i+1,j);
            LocalDOF(4,1) = mesh.DOF(i+1,j+1);

            LocalGridFn{1,1} = mesh.GridFn{i,j};
            LocalGridFn{2,1} = mesh.GridFn{i,j+1};
            LocalGridFn{3,1} = mesh.GridFn{i+1,j};
            LocalGridFn{4,1} = mesh.GridFn{i+1,j+1};
            [locNodes,extraNodes] = AccuElemNodeData(LocalGridFn,order,Nodes);
            Nodes = Nodes + extraNodes;

            fespace(k).Element = k;
            fespace(k).ElemDOF = [LocalDOF;locNodes.locDOF];

            fespace(k).ElemGrid = [LocalGridFn;locNodes.pt];
            k = k + 1;
        end
    end

end
```

```
function [elemNodes,extraNodes] = AccuElemNodeData(LocalGridFn,order,TDof)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 21, 2021$
% $Code Version: 1.0$
% This function is used to generate internal grid points for additional
% nodes that will be added due to increase in polynomial order.
% Inputs : LocalGridFn - 1D array that contain the corner grid values of
%                        the corner nodes of the element
%          order       - order of polynomial used
%          TDof        - Total Degrees of freedom currently in the mesh
% Outputs: elemNodes        - structure
%          elemNodes.locDOF - Adds DOF values of the added nodes on element
%          elemNodes.pt     - Adds grid values as 1D array for added nodes
%                             on element
%          extraNodes       - Total Number of added nodes
% since only quadrilateral nodes are considered and lagrange
% polynomials, the way to find out num of added nodes is :

    extraNodes = (order + 1)^2 - 4; % 4 nodes are already present at the edge of each element
    loc_x1_lim1 = LocalGridFn{1,1}(1);
    loc_x1_lim2 = LocalGridFn{2,1}(1);

    loc_x2_lim1 = LocalGridFn{1,1}(2);
    loc_x2_lim2 = LocalGridFn{3,1}(2);

    l_dx1 = (loc_x1_lim2 - loc_x1_lim1)/order;
    l_dx2 = (loc_x2_lim2 - loc_x2_lim1)/order;

    n = order + 1;
    k = 1;
    locDOF = [];
    pt = {};
    for i=1:n % x2
        for j=1:n %x1
            if i==1 && j==1
                continue
            elseif i==1 && j==n
                continue
            elseif i==n && j==1
                continue
            elseif i==n && j==n
                continue
            else
                locDOF(k,1) = TDof + k;
                pt{k,1}(1,1) = loc_x1_lim1 + (j-1)*l_dx1;
                pt{k,1}(1,2) = loc_x2_lim1 + (i-1)*l_dx2;
                k = k + 1;
            end
        end
    end

    elemNodes.locDOF = locDOF;
    elemNodes.pt = pt;
end
```

```
function [B_zn,S_zn] = Eval_ShapeFn(order)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 21, 2021$
% $Code Version: 1.0$
% Shape Functions attached to each node of the element and its
% corresponding Gradients with respect to \zeta and \eta
% Inputs : order - order of polynomial to generate shape functions for
% Outputs: B_zn  - [N1,zeta N2,zeta N3,zeta N4,zeta]
%                  [N1,eta  N2,eta  N3,eta  N4,eta ]
%                  evaluated at each integral point - [2x4x4] array
%          S_zn  - [N1 N2 N3 N4]
%                  [N1 N2 N3 N4]
%                  evaluated at each integral point - [2x4x4] array

    [ShapeFn,DShapeFn] = H1_FECollection(order);
    [Quad_pts,~] = IntRules();
    num_IntPts = length(Quad_pts);
    [dim,n] = size(DShapeFn);
    B_zn = zeros(dim,n,num_IntPts);
    S_zn = zeros(dim,n,num_IntPts);


    for i=1:dim
        for j=1:n
            B_zn(i,j,:) = DShapeFn{i,j}(Quad_pts(:,1),Quad_pts(:,2)); % 2D
            S_zn(i,j,:) = ShapeFn{j}(Quad_pts(:,1),Quad_pts(:,2)); %2D
        end
    end


end
```

```
function [ShapeFn, DShapeFn] = H1_FECollection(order)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 21, 2021$
% $Code Version: 1.0$
% H1_FECollection - collection of H1 continuous Finite Element shape
% functions and its derivatives are sent as output based on order
% Inputs : order    - order of polynomials required
% Outputs: ShapeFn  - Shape function at each node based on order
%          DShapeFn - Derivatives of shape function at each node on order

    if order == 1
        ShapeFn{1} = @(zeta,eta) 0.25* (1 - zeta).* (1 - eta);
        ShapeFn{2} = @(zeta,eta) 0.25* (1 + zeta).* (1 - eta);
        ShapeFn{3} = @(zeta,eta) 0.25* (1 - zeta).* (1 + eta);
        ShapeFn{4} = @(zeta,eta) 0.25* (1 + zeta).* (1 + eta);

        DShapeFn{1,1} = @(zeta,eta) -0.25* (1 - eta); % DN1/Dzeta
        DShapeFn{2,1} = @(zeta,eta) -0.25* (1 - zeta); % DN1/Deta

        DShapeFn{1,2} = @(zeta,eta) 0.25* (1 - eta); % DN2/Dzeta
        DShapeFn{2,2} = @(zeta,eta) -0.25* (1 + zeta); % DN2/Deta

        DShapeFn{1,3} = @(zeta,eta) -0.25* (1 + eta); % DN3/Dzeta
        DShapeFn{2,3} = @(zeta,eta) 0.25* (1 - zeta); % DN3/Deta

        DShapeFn{1,4} = @(zeta,eta) 0.25* (1 + eta); % DN4/Dzeta
        DShapeFn{2,4} = @(zeta,eta) 0.25* (1 + zeta); % DN4/Deta

    else
        ShapeFn{1} = 0;
        DShapeFn{1} = 0;
        disp("sorry not supported");
    end

end
```

```
function [Quad_pts,Quad_wts] = IntRules()
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 21, 2021$
% $Code Version: 1.0$
% This function outputs the Quadrature points and Quadrature weights for a
% \zeta-\eta coordinate system.
% Outputs : Quad_pts - 4 grid locations of integration point/ quadrature
%                      points [4x2] array
%           Quad_wts - weights attached to each of the quadrature points

    Quad_wts = [1 1 1 1];

    Quad_pts = [-0.5774 -0.5774; 0.5774 -0.5774; -0.5774 0.5774; 0.5774 0.5774];

end
```

```
function [B,det_J] = ElementTransformation(Eval_DShapeFn,GridPts,choice)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 24, 2021$
% $Code Version: 1.0$
% This function evaluates the transformation of evaluated local Shape
% Function Gradients from local coordinate system to global coordinate
% system and also compute the determinant of jacobian at the integration
% points.
% Inputs : Eval_DShapeFn - Gradient of Shape functions evaluated at
%                          Quadrature or integration points
%          GridPts       - global GridPts of nodes of the elements
%          choice        - choice of integrator.
% Outputs: B             - Transformed Gradient of Shape functions at
%                          Quadrature points
%          det_J         - determinant of Jacobians at global nodal
%                          locations of nodes of the element

    [dim,n,num_IntPts] = size(Eval_DShapeFn);
    B = zeros(dim,n,num_IntPts);
    det_J = zeros(1,num_IntPts);
    det_tol = 1e-4;

    if choice == 3 % diffusion
        for i=1:num_IntPts
            t = Eval_DShapeFn(:,:,i);
            J = t*GridPts;
            J = J';
            det_J(1,i) = det(J);
            if det_J(1,i) <= det_tol
                det_J(1,i) = det_tol;
            end
            cofJ = (adjoint(J));
            B(:,:,i) = (1/det_J(1,i))*cofJ*Eval_DShapeFn(:,:,i);
        end
    end

    if (choice == 2 || choice == 1) % convection - 2, mass integrator - 1
        for i=1:num_IntPts
            t = Eval_DShapeFn(:,:,i);
            J = t*GridPts;
            J = J';
            det_J(1,i) = det(J);
            if det_J(1,i) <= det_tol
                det_J(1,i) = det_tol;
            end
            cofJ = (adjoint(J));
            B(:,:,i) = (1/det_J(1,i))*cofJ*Eval_DShapeFn(:,:,i);
        end
    end

end
```

```
function d_sub = DiffusionIntegrator(diff_coeff,B_zn,LocGridArr)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 21, 2021$
% $Code Version: 1.0$
% This function performs the weak form of diffusion integration
% Inputs : diff_coeff   - coefficient matrix or scalar integration
%                         constants
%          B_zn         - Shape Function gradients evaluated at integration
%                         points
%          LocGridArr   - The Grid Locations of the nodes present in the
%                         element being integrated
% Outputs: d_sub        - The diffusion element stiffness sub-matrix

    choice = 3; % diffusion

    [B,detJ] = ElementTransformation(B_zn,LocGridArr,choice);
    d_sub = diff_coeff*NumInt(B,detJ,choice);

end
```

```
function ElemStiffness = NumInt(B,detJ,choice)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 21, 2021$
% $Code Version: 1.0$
% This function performs numerical integration of the desired choice
% Inputs : choice        - Diffusion = 3, Convection = 2
% Output : ElemStiffness - Outputs the integrated Element Stiffness Matrix

    [~,n,num_IntPts] = size(B);
    [~,Quad_wts] = IntRules();
    ElemStiffness = zeros(n);

    if choice == 3
        for i=1:num_IntPts
            ElemStiffness = ElemStiffness + B(:,:,i)'*B(:,:,i)*detJ(i)*Quad_wts(i);
        end
    end

    if choice == 2
    end
end
```

```
function c_sub = ConvectionIntegrator(conv_coeff,B_zn,S_zn,LocalGridArr,par)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 21, 2021$
% $Code Version: 1.0$
% This function performs the convection weak integration.
% Inputs : conv_coeff   - coefficient matrix or scalar integration
%                         constants
%          B_zn         - Shape Function gradients evaluated at integration
%                         points
%          S_zn         - Shape Function evaluated at integration points
%          LocGridArr   - The Grid Locations of the nodes present in the
%                         element being integrated
%          par          - Parameter that influences the flux
% Outputs: c_sub        - Element Stiffness matrix for convection
%                         integration

    choice = 2; % convection
    flux = Calc_Flux(S_zn,LocalGridArr,par);
    [B,detJ] = ElementTransformation(B_zn,LocalGridArr,choice);
    num_Int = length(detJ);
    [~,n,~] = size(B);
    c_sub = zeros(n);

    for i=1:num_Int
        t = flux(:,:,i).*S_zn(:,:,i);
        c_sub = c_sub + B(:,:,i)'*t*detJ(i);
    end

    c_sub = conv_coeff*c_sub;

end
```

```
function flux = Calc_Flux(ShapeFn,GridPt,par)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 21, 2021$
% $Code Version: 1.0$
% This function computes the flux values at the integration points on the
% global coordinate system.
% Inputs : ShapeFn  - Shape Function evaluated at integration points
%          GridPt   - global Grid Location of the nodes of the element
%          par      - parameter determining the dynamic equation
% Outputs: flux     - flux values computed at integration points

    % Function that calculates the -flux values at all integration points
    [dim,n,num_Int] = size(ShapeFn);
    flux = zeros(dim,n,num_Int);

    for i=1:num_Int
        x = ShapeFn(1,:,i)*GridPt;
        flux(1,:,i) = -x(2);
        flux(2,:,i) = x(1) - par*(1 - x(1)^2)*x(2);
    end

end
```

```
function [i_append, j_append, val_append] = Assemble_NoBC(m,fespace)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 21, 2021$
% $Code Version: 1.0$
% This function performs the act of assembling the local element stiffness
% matrix to the Global Stiffness matrix.
% Inputs : m          - local element stiffness matrix to assemble
%          fespace    - the finite element space of the corresponding element
% Outputs: i_append   - the 'i' idx of the Global Sparse Matrix to append
%          j_append   - the 'j' idx of the Global Sparse Matrix to append
%          val_append - the value to add at the 'i,j' location of the
%                       Global Sparse Matrix

% No Boundary conditions are applied in this assembly process

i_append = [];
j_append = [];
val_append = [];

n = length(m);
k = 1;

for i=1:n
    eq_num = fespace.ElemDOF(i);
        for j=1:n
            col = fespace.ElemDOF(j);
            i_append(k,1) = eq_num;
            j_append(k,1) = col;
            val_append(k,1) = m(i,j);
            k = k + 1;
        end
end

end
```

```
function M_out = partialAssemble(M_in,I,J,V)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 21, 2021$
% $Code Version: 1.0$
% This function Assemble the Element stiffness matrix in the following
% manner.
% M_out = M_in + append(M_in(I,J) = V)

M_out = M_in;
for i=1:length(I)
    M_out(I(i),J(i)) = M_out(I(i),J(i)) + V(i);
end

end
```

2D MFEM FPE code

```cpp

#include "mfem.hpp"
#include <fstream>
#include <iostream>

using namespace std;
using namespace mfem;

double eps = 0.2;

void flux_function (const Vector &x, Vector &f);
double g_function(const Vector &x);

int main(int argc, char *argv[])
{
   // 1. Parse command line options
   // mesh generated from -3 to 3 of the whole domain in x1 and x2 direction
   const char *mesh_file = "./FPE_2D_new.mesh";
   int order = 1;
   int ref_lev = 0;

   OptionsParser args(argc, argv);
   args.AddOption(&mesh_file, "-m", "--mesh", "Mesh file to use.");
   args.AddOption(&order, "-o", "--order", "Finite element polynomial degree");
   args.AddOption(&ref_lev, "-r", "--refine", "refinement level");
   args.ParseCheck();

   // 2. Read the mesh from the given mesh file, and refine once uniformly.
   Mesh mesh(mesh_file);
   for (int i=0; i< ref_lev; i++)
   {
      mesh.UniformRefinement();
   }

   // 3. Define a finite element space on the mesh. Here we use H1 continuous
   //    high-order Lagrange finite elements of the given order.
   H1_FECollection fec(order, mesh.Dimension());
   FiniteElementSpace fespace(&mesh, &fec);
   cout << "Number of unknowns: " << fespace.GetTrueVSize() << endl;

   // 4. Set up the linear form b(.) corresponding to the right-hand side.
   ConstantCoefficient zero(0.0);
   ConstantCoefficient one(1.0);

   // 5. Set up the bilinear form a(.,.) corresponding to the -Delta operator.
   double h = 0.3;
   int dim = mesh.Dimension();
   ConstantCoefficient cov(-1*pow(h,2)/2);
   VectorFunctionCoefficient flux(dim,flux_function);
   FunctionCoefficient g(g_function);
```

```
51
52      BilinearForm a(&fespace);
53      // splitting the advection term into convection term and mass term
54      a.AddDomainIntegrator(new ConvectionIntegrator(flux));
55      a.AddDomainIntegrator(new DiffusionIntegrator(cov));
56      a.AddDomainIntegrator(new MassIntegrator(g));
57
58      a.Assemble();
59      a.Finalize();
60      SparseMatrix AA;
61      AA = a.SpMat();
62
63      // save sparse matrix and use MATLAB to perform linear algebra after
64      // adding the weight constraints to the sparse matrix
65      ofstream ksave("k.txt");
66      AA.PrintMatlab(ksave);
67      ksave.close();
68
69      return 0;
70  }
71
72  void flux_function(const Vector &x, Vector &f)
73  {
74      int dim = x.Size();
75      // computing the negative flux in this problem
76
77      f(0) = -1*x(1);
78      f(1) = -1* (-x(0) + eps*(1 - pow(x(0),2))*x(1));
79  }
80
81  double g_function(const Vector &x)
82  {
83      return -1*eps*(1 - pow(x(0),2));
84  }
```

Contents


```
% Post Process the generated Element Stiffness Matrix to find FPE solution
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 30, 2021$
% $Code Version: 1.1$

clc
clear all

IDX = readmatrix('k.txt');
nx = 100; ny = 100;
x1 = linspace(-3,3,nx+1);
x2 = linspace(-3,3,ny+1);
TDof = (nx+1)*(ny+1);

M = sparse(IDX(:,1),IDX(:,2),IDX(:,3),...
    TDof+1,TDof+1);
```

Calculate trapezoidal weight constraints

```
w = CalcTrapzWts(nx,ny,-3,3,-3,3);

M(TDof+1,1:TDof) = w;
M(1:TDof,TDof+1) = w';

figure
spy(M)
xlabel('Dof...');
ylabel('Dof...');
title('Sparse Element Stiffness Matrix')
```

Find solution rho

```
RHS = [zeros(TDof,1);1];

X = M\RHS;

rho = reshape(X(1:TDof),ny+1,nx+1);
```

Plotting

```
[X,Y] = meshgrid(x1,x2);
figure
surf(X,Y,rho);
colorbar
xlabel('x1');
ylabel('x2');
zlabel('pdf');
title('MFEM - VanDerPol FPE','$\varepsilon = 0.2, h = 0.3$','Interpreter','latex');
```

```
figure
contourf(X,Y,rho,20);
colorbar
xlabel('x1');
ylabel('x2');
title('MFEM - VanDerPol FPE','$\varepsilon = 0.2, h = 0.3$','Interpreter','latex');
```

```
function w = CalcTrapzWts(nx,ny,x1_lim1,x1_lim2,x2_lim1,x2_lim2)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 30, 2021$
% $Code Version: 1.1$
% This function is to generate the trapezoidal weights associated to each
% node of the mesh (integration weight associated with the nodes).
% Inputs: nx        : x direction discretization
%         ny        : y direction discretization
%         x1_lim1   : lower limit of x direction in domain
%         x1_lim2   : upper limit of x direction in domain
%         x2_lim1   : lower limit of y direction in domain
%         x2_lim2   : upper limit of y direction in domain
% Output: w         : weight array associated to each DOF

n = (nx+1)*(ny+1);
w = zeros(1,n);

m1 = nx+1;
m2 = ny+1;

dx1 = (x1_lim2-x1_lim1)/(nx-1);
dx2 = (x2_lim2-x2_lim1)/(ny-1);
k = 1;
for i=1:m1
    for j=1:m2
        if (i==1 && j==1) || (i==1 && j==m2) ||...
                (i==m1 && j==1) || (i==m1 && j==m2)
            w(k) = dx1*dx2/4;
            k = k + 1;
        elseif (i==1) || (i==m1) || ...
                (j==1) || (j==m2)
            w(k) = dx1*dx2/2;
            k = k + 1;
        else
            w(k) = dx1*dx2;
            k = k+1;
        end
    end
end

end
```

2D Finite Difference FPE – MATLAB code

Contents


```
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% Script to perform Finite Difference FPE simulation
clc
clear all

dx1 = 0.06; dx2 = 0.06;
meshData = generateRecMesh(dx1,dx2,-3,3,-3,3);
[m,n] = size(meshData.DOF);
len = meshData.nDOF;
h = 0.3;

Diff = Diffusion_Assemble(meshData,h^2/2);
flux = Calc_Flux(meshData.GridFn,0.2);

Conv = Convection_Assemble(meshData,-1,flux);
Wts = CalcTrapzWeights(meshData);
```

Generate FD LHS matrix

```
A = Diff + Conv;
A(len+1,len+1) = 0;
A(len+1,1:len) = Wts;
A(1:len,len+1) = Wts';
clear Diff Conv Wts meshData
```

Set up RHS and solve for rho

```
B = [zeros(len,1);1];
X = A\B;
pdf_f = reshape(X(1:len),n,m);
clear X A B flux
```

Plotting

```
[X,Y] = meshgrid(-3:dx1:3,-3:dx2:3);
figure
surf(X,Y,pdf_f);
colorbar
xlabel('$x_1$','Interpreter','latex');
ylabel('$x_2$','Interpreter','latex');
zlabel('pdf')
title('Finite Difference VanDerPol FPE solution',...
    '$\varepsilon = 0.2, h = 0.3$',...
    'Interpreter','latex')

figure
```

```
contourf(X,Y,pdf_f);
colorbar
xlabel('$x_1$','Interpreter','latex');
ylabel('$x_2$','Interpreter','latex');
title('Finite Difference VanDerPol FPE solution',...
    '$\varepsilon = 0.2, h = 0.3$',...
    'Interpreter','latex')
```

```
function MeshData = generateRecMesh(dx1,dx2,x1_lim1,x1_lim2,x2_lim1,x2_lim2)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% Inputs: dx1 - discretization along x1-axis
%         dx2 - discretization along x2-axis
%         x1_lim1 - lower limit of x1 dimension of the domain to mesh
%         x1_lim2 - upper limit of x1 dimension of the domain to mesh
%         x2_lim1 - lower limit of x2 dimension of the domain to mesh
%         x2_lim2 - upper limit of x2 dimension of the domain to mesh
% Outputs: struct mesh
%          mesh.nDOF   - Number of nodal elements present in the domain
%          mesh.DOF        - 2D Matrix with each node holding its DOF value
%          mesh.CornerDOF  - 1D array holding the DOF values of domain
%                            corners
%          mesh.BoundaryDOF- 1D array holding the DOF values of domain
%                            boundary
%          mesh.GridFn     - 2D cell array with each cell holding the
%                            domain location of each nodal DOF
%          mesh.nx         - dx1 discretization along x1
%          mesh.ny         - dx2 discretization along x2

    % Generating Mesh
    x1 = x1_lim1:dx1:x1_lim2;
    x2 = x2_lim1:dx2:x2_lim2;

    dim = 2;

    nDOF = length(x1)*length(x2);

    MeshDOF = zeros(length(x2),length(x1));
    GridFn = cell(length(x2),length(x1));
    k = 1;
    b_dof = 1;
    c_dof = 1;
    corner_dof = zeros(2*dim,1);
    per = 2*(length(x1)-1) + 2*(length(x2)-1);
    boundary_dof = zeros(per,1);
    for i=1:length(x2)
        for j=1:length(x1)
            MeshDOF(i,j) = k;
            GridFn{i,j} = [x1(j),x2(i)];
            if i==1 && j==1 % corner 1
                corner_dof(c_dof) = k;
                c_dof = c_dof + 1;
            elseif i==1 && j== length(x1) % corner 2
                corner_dof(c_dof) = k;
                c_dof = c_dof + 1;
            elseif i==length(x2) && j==1 % corner 3
                corner_dof(c_dof) = k;
                c_dof = c_dof + 1;
            elseif i== length(x2) && j==length(x1) % corner 4
                corner_dof(c_dof) = k;
                c_dof = c_dof + 1;
            end
            if i==1
                boundary_dof(b_dof) = k;
                b_dof = b_dof + 1;
            elseif j==1 || j==length(x1)
```

```
                boundary_dof(b_dof) = k;
                b_dof = b_dof + 1;
            elseif i == length(x2)
                boundary_dof(b_dof) = k;
                b_dof = b_dof + 1;
            end

            k = k + 1;
        end
    end

    MeshData.nDOF = nDOF;
    MeshData.DOF = MeshDOF;
    MeshData.GridFn = GridFn;
    MeshData.CornerDOF = corner_dof;
    MeshData.BoundaryDOF = boundary_dof;
    MeshData.nx = dx1;
    MeshData.ny = dx2;
end
```

```
function D = Diffusion_Assemble(mesh,diff_coeff)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This function performs the diffusion finite differencing operation and
% assembles the values in the right place of the matrix
% Inputs - mesh       - structure mesh generated by generateRecMesh function
%          diff_coeff - diffusion constant coefficient
% Output - D          - Diffusion matrix

    len = mesh.nDOF;
    D = zeros(len);
    [m,n] = size(mesh.DOF);
    dx1 = mesh.nx;
    dx2 = mesh.ny;
    for i=1:m
        for j=1:n
            bool1 = chk_dof(mesh.CornerDOF,mesh.DOF(i,j));
            bool2 = chk_dof(mesh.BoundaryDOF,mesh.DOF(i,j));
            r = mesh.DOF(i,j);
            D(r,r) = diff_coeff*(-2/dx1^2 + -2/dx2^2);
            if bool1
                if i==1 && j==1
                    cp = mesh.DOF(i,j+1);
                    rp = mesh.DOF(i+1,j);
                    D(r,cp) = diff_coeff*(1/dx1^2);
                    D(r,rp) = diff_coeff*(1/dx2^2);
                elseif i==1 && j==n
                    cm = mesh.DOF(i,j-1);
                    rp = mesh.DOF(i+1,j);
                    D(r,cm) = diff_coeff*(1/dx1^2);
                    D(r,rp) = diff_coeff*(1/dx2^2);
                elseif i==m && j==1
                    cp = mesh.DOF(i,j+1);
                    rm = mesh.DOF(i-1,j);
                    D(r,cp) = diff_coeff*(1/dx1^2);
                    D(r,rm) = diff_coeff*(1/dx2^2);
                else
                    cm = mesh.DOF(i,j-1);
                    rm = mesh.DOF(i-1,j);
                    D(r,cm) = diff_coeff*(1/dx1^2);
                    D(r,rm) = diff_coeff*(1/dx2^2);
                end
            elseif bool2
                if i==1
                    cp = mesh.DOF(i,j+1);
                    cm = mesh.DOF(i,j-1);
                    rp = mesh.DOF(i+1,j);
                    D(r,cp) = diff_coeff*(1/dx1^2);
                    D(r,cm) = diff_coeff*(1/dx1^2);
                    D(r,rp) = diff_coeff*(1/dx2^2);
                elseif j==1 || j==n
                    if j==1
                        cp = mesh.DOF(i,j+1);
                        rp = mesh.DOF(i+1,j);
                        rm = mesh.DOF(i-1,j);
                        D(r,cp) = diff_coeff*(1/dx1^2);
                        D(r,rp) = diff_coeff*(1/dx2^2);
                        D(r,rm) = diff_coeff*(1/dx2^2);
```

```
                else
                    cm = mesh.DOF(i,j-1);
                    rp = mesh.DOF(i+1,j);
                    rm = mesh.DOF(i-1,j);
                    D(r,cm) = diff_coeff*(1/dx1^2);
                    D(r,rp) = diff_coeff*(1/dx2^2);
                    D(r,rm) = diff_coeff*(1/dx2^2);
                end
            elseif i==m
                cp = mesh.DOF(i,j+1);
                cm = mesh.DOF(i,j-1);
                rm = mesh.DOF(i-1,j);
                D(r,cp) = diff_coeff*(1/dx1^2);
                D(r,cm) = diff_coeff*(1/dx1^2);
                D(r,rm) = diff_coeff*(1/dx2^2);
            end
        else
            cp = mesh.DOF(i,j+1);
            cm = mesh.DOF(i,j-1);
            rp = mesh.DOF(i+1,j);
            rm = mesh.DOF(i-1,j);
            D(r,cp) = diff_coeff*(1/dx1^2);
            D(r,cm) = diff_coeff*(1/dx1^2);
            D(r,rp) = diff_coeff*(1/dx2^2);
            D(r,rm) = diff_coeff*(1/dx2^2);
        end
    end
  end
end
```

```
function bool = chk_dof(Vec,dof)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 11, 2021$
% $Code Version: 1.0$
% This function returns a boolean which checks if the input dof is present
% in the input vector array

    bool = any(Vec == dof);
end
```

```
function flux = Calc_Flux(GridFn, par)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This function calculates the flux values at the nodal locations
% Inputs - GridFn - global coordinate location of the nodes
%          par    - parameter influencing the dynamics
% Output - flux   - cell array with the flux vector at each nodal location

    [m,n] = size(GridFn);
    flux = cell(m,n);
    f = zeros(1,2);
    for i=1:m
        for j=1:n
            x = GridFn{i,j};
            f(1) = x(2);
            f(2) = -x(1) + par*(1 - x(1)^2)*x(2);
            flux{i,j} = f;
        end
    end

end
```

```
function C = Convection_Assemble(mesh,conv_coeff,flux)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 10, 2021$
% $Code Version: 1.0$
% This function performs the convection finite differencing operation and
% assembles the values in the right place of the matrix
% Inputs - mesh       - structure mesh generated by generateRecMesh function
%          conv_coeff - convection constant coefficient
%          flux       - flux computed at each grid location
% Output - C          - Convection matrix

    len = mesh.nDOF;
    C = zeros(len);
    [m,n] = size(mesh.DOF);
    dx1 = mesh.nx;
    dx2 = mesh.ny;
    for i=1:m
        for j=1:n
            bool1 = chk_dof(mesh.CornerDOF,mesh.DOF(i,j));
            bool2 = chk_dof(mesh.BoundaryDOF,mesh.DOF(i,j));
            r = mesh.DOF(i,j);
            if bool1
                if i==1 && j==1
                    cp = mesh.DOF(i,j+1);
                    f_cp = flux{i,j+1}(1);
                    rp = mesh.DOF(i+1,j);
                    f_rp = flux{i+1,j}(2);
                    f_c = flux{i,j}(1);
                    f_r = flux{i,j}(2);
                    C(r,r) = conv_coeff* (-f_c/dx1 + -f_r/dx2);
                    C(r,cp) = conv_coeff* f_cp/dx1;
                    C(r,rp) = conv_coeff* f_rp/dx2;
                elseif i==1 && j==n
                    cm = mesh.DOF(i,j-1);
                    f_cm = flux{i,j-1}(1);
                    rp = mesh.DOF(i+1,j);
                    f_rp = flux{i+1,j}(2);
                    f_c = flux{i,j}(1);
                    f_r = flux{i,j}(2);
                    C(r,r) = conv_coeff* (f_c/dx1 + -f_r/dx2);
                    C(r,cm) = conv_coeff* -1* f_cm/dx1;
                    C(r,rp) = conv_coeff* f_rp/dx2;
                elseif i==m && j==1
                    cp = mesh.DOF(i,j+1);
                    f_cp = flux{i,j+1}(1);
                    rm = mesh.DOF(i-1,j);
                    f_rm = flux{i-1,j}(2);
                    f_c = flux{i,j}(1);
                    f_r = flux{i,j}(2);
                    C(r,r) = conv_coeff* (-f_c/dx1 + f_r/dx2);
                    C(r,cp) = conv_coeff* f_cp/dx1;
                    C(r,rm) = conv_coeff* -1* f_rm/dx2;
                else
                    cm = mesh.DOF(i,j-1);
                    f_cm = flux{i,j-1}(1);
                    rm = mesh.DOF(i-1,j);
                    f_rm = flux{i-1,j}(2);
                    f_c = flux{i,j}(1);
                    f_r = flux{i,j}(2);
```

```
            C(r,r) = conv_coeff* (f_c/dx1 + f_r/dx2);
            C(r,cm) = conv_coeff* -1* f_cm/dx1;
            C(r,rm) = conv_coeff* -1* f_rm/dx2;
        end
    elseif bool2
        if i==1
            cp = mesh.DOF(i,j+1);
            f_cp = flux{i,j+1}(1);
            cm = mesh.DOF(i,j-1);
            f_cm = flux{i,j-1}(1);
            rp = mesh.DOF(i+1,j);
            f_rp = flux{i+1,j}(2);
            f = flux{i,j}(2);
            C(r,cp) = conv_coeff* f_cp/(2*dx1);
            C(r,cm) = conv_coeff* -1* f_cm/(2*dx1);
            C(r,rp) = conv_coeff* f_rp/dx2;
            C(r,r) = conv_coeff* -1* f/dx2;
        elseif j==1 || j==n
            if j==1
                cp = mesh.DOF(i,j+1);
                f_cp = flux{i,j+1}(1);
                f = flux{i,j}(1);
                rp = mesh.DOF(i+1,j);
                f_rp = flux{i+1,j}(2);
                rm = mesh.DOF(i-1,j);
                f_rm = flux{i-1,j}(2);
                C(r,cp) = conv_coeff* f_cp/dx1;
                C(r,r) = conv_coeff* -1* f/dx1;
                C(r,rp) = conv_coeff* f_rp/(2*dx2);
                C(r,rm) = conv_coeff* -1* f_rm/(2*dx2);
            else
                cm = mesh.DOF(i,j-1);
                f_cm = flux{i,j-1}(1);
                f = flux{i,j}(1);
                rp = mesh.DOF(i+1,j);
                f_rp = flux{i+1,j}(2);
                rm = mesh.DOF(i-1,j);
                f_rm = flux{i-1,j}(2);
                C(r,cm) = conv_coeff* -1*f_cm/dx1;
                C(r,r) = conv_coeff*  f/dx1;
                C(r,rp) = conv_coeff* f_rp/(2*dx2);
                C(r,rm) = conv_coeff* -1* f_rm/(2*dx2);
            end
        elseif i==m
            cp = mesh.DOF(i,j+1);
            f_cp = flux{i,j+1}(1);
            cm = mesh.DOF(i,j-1);
            f_cm = flux{i,j-1}(1);
            rm = mesh.DOF(i-1,j);
            f_rm = flux{i-1,j}(2);
            f = flux{i,j}(2);
            C(r,cp) = conv_coeff* f_cp/(2*dx1);
            C(r,cm) = conv_coeff* -1* f_cm/(2*dx1);
            C(r,rm) = conv_coeff* -1* f_rm/dx2;
            C(r,r) = conv_coeff* f/dx2;
        end
    else
        cp = mesh.DOF(i,j+1);
        f_cp = flux{i,j+1}(1);
        cm = mesh.DOF(i,j-1);
        f_cm = flux{i,j-1}(1);
```

```
                rp = mesh.DOF(i+1,j);
                f_rp = flux{i+1,j}(2);
                rm = mesh.DOF(i-1,j);
                f_rm = flux{i-1,j}(2);
                C(r,cp) = conv_coeff* f_cp/(2*dx1);
                C(r,cm) = conv_coeff* -1* f_cm/(2*dx1);
                C(r,rp) = conv_coeff* f_rp/(2*dx2);
                C(r,rm) = conv_coeff* -1* f_rm/(2*dx2);
            end
        end
    end
end
```

```
function w = CalcTrapzWeights(mesh)
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 11, 2021$
% $Code Version: 1.0$
% This function generates the trapezoidal weights associated with each node
% of the mesh, which is the area around the node.
% Input : mesh - structure mesh generated using generateRecMech function
% Output: w    - trapezoidal weight array at each node location

    len = mesh.nDOF;
    w = zeros(1,len);
    dx1 = mesh.nx;
    dx2 = mesh.ny;
    [m,n] = size(mesh.DOF);

    for i=1:m
        for j=1:n
            bool1 = chk_dof(mesh.CornerDOF,mesh.DOF(i,j));
            bool2 = chk_dof(mesh.BoundaryDOF,mesh.DOF(i,j));
            idx = mesh.DOF(i,j);
            if bool1
                w(idx) = dx1*dx2/4;
            elseif bool2
                w(idx) = dx1*dx2/2;
            else
                w(idx) = dx1*dx2;
            end
        end
    end

end
```

Lorenz Attractor Deterministic solution – MATLAB code

Contents


```
% $Author : Vignesh Ramakrishnan$
% $RIN : 662028006$  $Date : November 15, 2021$
% $Code Version: 1.0$
% Script that generates the Deterministic FPE solution for a Lorenz system
t0 = 0;
tend = 200;
tstep = 0.01;
tspan = t0:tstep:tend;
X0 = [0;-1; 0];
[tp,Xp] = ode45(@Lorenz,tspan,X0);
```

plotting

```
figure(1)
plot(Xp(:,1),Xp(:,3));
xlabel('$x_1$','Interpreter','latex','FontSize',16);
ylabel('$x_3$','Interpreter','latex','FontSize',16);
title('Lorenz Attractor','$\sigma = 3, \Gamma = 26.5, \beta = 1$','Interpreter','latex','FontSize',16);

figure(2)
plot(Xp(:,1),Xp(:,2));
xlabel('$x_1$','Interpreter','latex','FontSize',16);
ylabel('$x_2$','Interpreter','latex','FontSize',16);
title('Lorenz Attractor','$\sigma = 3, \Gamma = 26.5, \beta = 1$','Interpreter','latex','FontSize',16);

figure(3)
plot(Xp(:,2),Xp(:,3));
xlabel('$x_2$','Interpreter','latex','FontSize',16);
ylabel('$x_3$','Interpreter','latex','FontSize',16);
title('Lorenz Attractor','$\sigma = 3, \Gamma = 26.5, \beta = 1$','Interpreter','latex','FontSize',16);
```

ODE45 simulation

```
function Xdot = Lorenz(t,X)
sig = 3;
rho = 26.5; beta = 1;
Xdot(1) = sig*(X(2)-X(1));
Xdot(2) = -X(2) + X(1)*(rho - X(3));
Xdot(3) = X(1)*X(2) - beta*X(3);
Xdot = Xdot';
end
```