Quantum Tornado II

Colin Fjelsted
Professor Mikhail Shvartsman
University of St. Thomas
CAM Final Report Summer 2024

I. Introduction/History

On March 18, 1925, one of the deadliest tornado outbreaks in recorded history generated at least twelve significant tornadoes and spanned a large portion of the midwestern and southern United States. In all, at least 751 people were killed and more than 2,298 were injured, making it the deadliest tornado outbreak. If tornado warnings were issued earlier, potentially 100s of lives could have been saved. Since 1925, many technological advances have made predicting tornadoes more accurate, but are still not perfect. Currently, tornado warnings are issued with an average lead time of 18 minutes, but even these are only effective 80% of the time. Hence, improving the understanding of tornado formation could lead to better prediction and potentially saving lives. This project aimed to understand tornado formation by employing properties of quantum mechanics, thermodynamics, fluid mechanics, and probability theory.

Advances over the past three decades, such as Doppler radar, allow for the detection of wind speed and direction. Doppler radar works by sending a beam of electromagnetic radiation waves, with a specific frequency, at the objects and analyzing how their motion changes the frequency of the reflected waves (this is also called the Doppler effect). The frequency variations allow meteorologists to find the radial component of the object's velocity relative to the radar. In the picture below (Fig. 1), low velocity is associated with the colors closer to violet or blue, and high velocity is associated with colors closer to red on the color spectrum. This assists in identifying storm circulations, which are often linked to tornadoes. However, Doppler radar is not perfect because it struggles to take into account all the subtle properties of our atmosphere, in particular, it struggles to accurately track turbulence.

Turbulence is the unsteady and almost random nature of air or any fluid. For instance, when water flows out of a sink, it initially follows a relatively predictable, deterministic path, but after the first few inches, the flow becomes seemingly random. This randomness is turbulence. Even in the present day, physicists still struggle with this concept. Many even argue that understanding turbulence is a more challenging problem to solve than quantum mechanics itself. Thus, classical physics itself does not have an accurate model of this turbulent behavior, which, again, is a major culprit in producing tornado-like vortices.

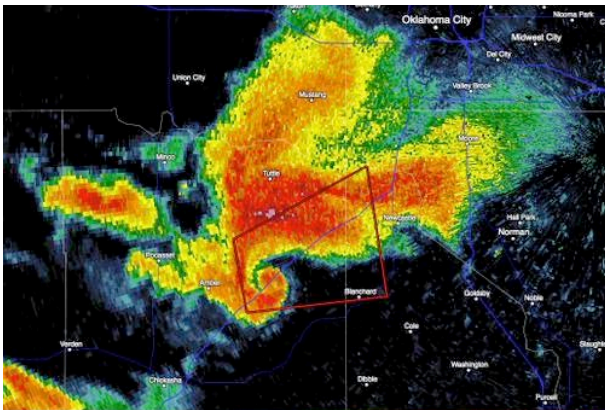


Fig 1. Doppler Radar image

One of the central issues regarding tornadoes is how numerous small vortices with high entropy evolve into a larger, faster vortex with lower entropy. Entropy is a concept that measures the level of disorder or randomness in a system. According to the second law of thermodynamics, entropy should increase over time in absence of other influences on a system. However, in thermodynamics it states that the system must be in equilibrium for that law of thermodynamics to hold true. Hence, since the atmosphere is not in equilibrium, the problem with entropy becomes much more complicated (Sasaki, p. 2107). Due to the complex nature of the atmosphere itself, physicists and metrologists alike theorize turbulence could be a culprit of allowing tornadoes to seemingly go against these laws of thermodynamics.

Tornadoes, in particular, are challenging to predict for many reasons, one being due to their small scale compared to hurricanes. Hence, many tornadoes are difficult to observe and track accurately, making the data less reliable and harder to obtain. This challenge is further exacerbated by the irregular and discontinuous nature of tornado vortices themselves. If tornadoes exhibited continuous behavior,

conventional fluid dynamics could describe them more accurately. The equations that govern classical fluids are known as the Navier-Stokes equations, which will be discussed in the next section of this paper. However, one of the million-dollar math problems is centered around the Navier-Stokes equations, underscoring the complexity and unpredictability of fluid dynamics. This makes predicting tornadoes using classical methods even more unreliable.

Given these challenges, this project employed the nonlinear quantum mechanical Gross-Pitaevskii equation, which models the dynamics of the wave function of a Bose-Einstein condensate. Through extensive simulations that varied numerous parameters, a substantial amount of data was collected and analyzed. This analysis aimed to draw connections between the findings and actual tornado behavior while also illuminating the properties of classical fluids. The primary objective was to model quantum vortices and explore their relationship with classical vortices, despite the inherent differences between them. Further details on this analysis will be provided in the subsequent sections of this paper.

II. Mathematical Equations

For classical fluid dynamics, Navier-Stokes equations are used. These equations are derived from the principles of conservation of mass and momentum (a restatement of Newton's 2nd law). They take into account viscosity of the fluid and describe how velocity, pressure, and density of a fluid change over time and space. These equations are used to better understand fluid flow.

$$(\partial \boldsymbol{v}_x/\partial x)+(\partial \boldsymbol{v}_y/\partial y)+(\partial \boldsymbol{v}_z/\partial z)=0$$

$$\rho((\partial \boldsymbol{v}_x/\partial t)+\boldsymbol{v}_x(\partial \boldsymbol{v}_x/\partial x)+\boldsymbol{v}_y(\partial \boldsymbol{v}_x/\partial y)+\boldsymbol{v}_z(\partial \boldsymbol{v}_x/\partial z))= -(\partial P/\partial x)+\mu((\partial^2 \boldsymbol{v}_x/\partial x^2)+(\partial^2 \boldsymbol{v}_x/\partial y^2)+(\partial^2 \boldsymbol{v}_x/\partial z^2))+\rho g_x$$

$$\rho((\partial \boldsymbol{v}_y/\partial t)+\boldsymbol{v}_x(\partial \boldsymbol{v}_y/\partial x)+\boldsymbol{v}_y(\partial \boldsymbol{v}_y/\partial y)+\boldsymbol{v}_z(\partial \boldsymbol{v}_y/\partial z))= -(\partial P/\partial y)+\mu((\partial^2 \boldsymbol{v}_y/\partial x^2)+(\partial^2 \boldsymbol{v}_y/\partial y^2)+(\partial^2 \boldsymbol{v}_y/\partial z^2))+\rho g_y$$

$$\rho((\partial \boldsymbol{v}_z/\partial t)+\boldsymbol{v}_x(\partial \boldsymbol{v}_z/\partial x)+\boldsymbol{v}_y(\partial \boldsymbol{v}_z/\partial y)+\boldsymbol{v}_z(\partial \boldsymbol{v}_z/\partial z))= -(\partial P/\partial z)+\mu((\partial^2 \boldsymbol{v}_z/\partial x^2)+(\partial^2 \boldsymbol{v}_z/\partial y^2)+(\partial^2 \boldsymbol{v}_z/\partial z^2))+\rho g_z$$

(ρ = density, $\boldsymbol{v}$ = velocity, P = pressure, μ = dynamic viscosity, g = gravitational field, x, y, z = Cartesian coordinates, t = time)

Implicit differentiation can be used to track the position of a ladder leaning against a wall as it falls to the ground. However, when solving the equation for the rate of change in the y-direction over time, the result suggests that this change approaches negative infinity, when logically it should be zero once the ladder reaches the ground. This paradox has puzzled mathematicians for decades. Despite this issue, the Navier-Stokes equation remains valuable for modeling real-world situations. The challenge of resolving this inconsistency is one reason why the solution is so highly sought after.

While Navier-Stokes equations model classical fluids, these equations can be related to quantum mechanics. "Ordinary turbulence, computed by solving the Navier-Stokes equation, also contains tubular coherent structures or vortex tubes of intense vorticity (Fig. 2), somewhat similar to the vortex lines" (Barenghi, p. 15). By solving these equations, researchers can gain insights into the complex interactions between air pressure, velocity, and vorticity (the measure of the rotation of a fluid element). This helps in understanding the mechanisms that drive tornado formation, sustain their rotation, and determine their intensity.

Navier-Stokes equations alone are not sufficient to fully capture the intricacies of tornadoes, as was discussed in the introduction. Thus, this project will dove into the quantum realm. The motivations of appealing to quantum mechanics were mentioned above. More specifically, this project turned to an analysis of the Gross-Pitaevskii equation (Barenghi, equation 7). The Gross-Pitaevskii equation (GPE) is a nonlinear partial differential equation (a variant of the Schrodinger equation) that describes the behavior of a quantum mechanical system of particles, specifically Bose-Einstein condensates (BECs). Let's note that classical Navier-Stokes equations can be derived from the Gross-Pitaevskii equation using the so-called Madelung transform of the wave function. The GPE can be written in different forms to model the phenomena under consideration.

Example Form: $i\hbar(\partial\psi/\partial t) = (-\hbar^2/2m)\nabla^2\psi + V(x)\psi + g|\psi|^2\psi$

($\hbar$ = reduced Planck's constant, ψ = wave function, m = particle mass, ∇ = gradient, g = interaction parameter)

At temperatures below 2.17 Kelvin, some fluids behave as super fluids and acquire quantum properties. These super fluids are Bose-Einstein condensates. In this state of matter, a large number of particles may occupy the same quantum state. Bose-Einstein condensates are instrumental in studying

quantum vortices due to their unique features. BECs exist in a state where all particles occupy the lowest energy level. The behavior of particles in this state is not individual-centric, but rather described via a wavefunction that encompasses the entire particle ensemble.

These fluids form quantum lines, around which these quantum vortices form. If these lines are close enough and aligned, they undergo spontaneous partial polarization, causing them to become parallel (share a phase) and combine into a larger vortex with increased velocity. “The ideal superfluid has zero entropy and zero viscosity, whereas the normal fluid, consisting of thermal excitations, is viscous and carries the entire heat content of the liquid” (Barenghi, p. 5). Hence, quantum vortices have quantum properties, including a non-zero phase change in rotation and the lack of viscosity.

Quantum vortices and classical vortices both are rotating structures and influence flow patterns. However, the behavior of quantum vortices differs from their classical counterparts due to quantum effects, leading to phenomena such as an empty core around which rotation happens and the quantization of circulation.. Studying these vortices within the context of BECs offers a deeper understanding of vortex physics and could shed light on the behavior of vortices in classical fluids, despite their essential differences.

According to the Heisenberg uncertainty principle, in quantum mechanics, particles are described by wave functions. The wave function in quantum mechanics provides insights into the probable positions of particles. The magnitude of the wave function squared represents the probability density function of a particle or a wave being at a certain area. A probability density function is a function whose volume integral equals one. These justifications above lead to studying the solutions of the GPE equation as possible models of tornado vortices.

III. Research Methods

To find and display solutions of the GPE equations, this project used Xavier Antoine and Romain Duboscq’s GPELab™, a MATLAB package designed specifically for the Gross-Pitaevskii equation. This code allowed us to solve the Gross-Pitaevskii equation using fast Fourier transforms (FFTs). FFTs are an efficient algorithm for computing the discrete Fourier transform (DFT) and its inverse, which are essential for converting complex wave functions between spatial and momentum space. This technique enables the handling of the nonlinear interactions and kinetic terms in the GPE more effectively, facilitating precise and computationally efficient simulations.

While GPELab simplified the process of solving the wave function, much of the output processing still required extensive custom coding. Beyond the GPELab code, the project’s code grew to thousands of lines, with a portion focused on reducing computation time—a critical challenge when running a large number of simulations. Through these optimizations, the computation time for a single simulation was reduced from approximately thirty minutes to just around ten seconds. This efficiency boost enabled me to save significantly more data and conduct a more comprehensive analysis.

Given that research on quantum vortices is relatively new, I decided to develop various live image outputs, GIFs, and multiple simulation results to gain a clearer understanding of the simulations. For each live image output, values were calculated for each pixel in the image. For example, in a 100 by 100 grid, calculating the vorticity norm would require 100 * 100 = 10,000 calculations—one for each pixel in the image. Each calculation corresponded to a specific color, with each color representing a particular numerical value. I saved each image from each iteration in a designated folder on my desktop, which allowed me to generate GIFs. These GIFs, akin to flip books, provided a dynamic visualization of how the simulations progressed over time, significantly aiding in the comparison of different simulations. Every output from a given simulation was stored in this folder, with the folder name and save structure easily adaptable based on how the code was written. In total, the live image outputs included: phase of the wave function, magnitude of the wave function squared, magnitude of the gradient (velocity), x-directional velocity, y-directional velocity, imaginary part of the wave function, real part of the wave function, vorticity contours, and velocity vector fields. The x-directional and y-directional velocities were particularly useful in understanding the overall direction of the simulation, especially when compared to the magnitude of the gradient, which can be interpreted as velocity

For many of the simulation outputs, I had to create custom user-defined functions, often incorporating FFT techniques to calculate specific values. This was a challenging process that required navigating multiple interfaces and understanding the intricate workings of GPELab. Calculating specific metrics, such as the vorticity norm, involved employing specialized algorithms. For instance, the vorticity norm is derived from the two-dimensional live image space, but this space isn't continuous—it's composed of discrete pixels, much like a digital screen. In a 100 by 100 grid, calculating the vorticity norm would require 10,000 individual calculations—one for each pixel in the image. An example calculation for the vorticity norm is illustrated by the following equations: $\| \text{Vorticity} \|_2 \approx (\sum_{i,j} |\text{Vorticity}(x_i,y_j)|^2 \Delta x \Delta y)^{(1/2)}$, where $\text{Vorticity}=\text{Grad}_x - \text{Grad}_y$. This process was repeated for each simulation output, particularly those that yielded single values, such as energy, angular momentum, and the vorticity norm. In summary, the simulation outputs I calculated include: energy, angular momentum, square at the origin, gradient norm, vorticity norm, x-rms, y-rms, energy evolution, helicity, maximum gradient magnitude, maximum x-directional velocity, and maximum y-directional velocity. Each of these values were saved as arrays in text files for every iteration of each simulation. This approach provided a clearer understanding and quantification of these pioneering simulations.

Moreover, once the code was fully operational in terms of saving data and images, then the testing proceeded. The initial tests involved varying the potential function, which in the Gross-Pitaevskii equation represents the external forces acting on the Bose-Einstein condensate, such as trapping potentials or interactions with other particles. By adjusting this potential function, I investigated how various external conditions influenced the behavior of quantum vortices in the simulations. This potential function corresponds to the $V(x)\psi$ term in the Gross-Pitaevskii equation, which plays a crucial role in modeling the external forces acting on the system. For instance, one potential function was defined as: PotentialFunction = @(X, Y) A * log(sqrt(X.^2 + Y.^2) + epsilon) + B * atan2(Y, X) + C * sqrt(X.^2 + Y.^2) + Omega * (X.^2 + Y.^2); where the parameters A, B, C, Omega, and epsilon could be modified. Numerous other potential functions were simulated with different scenarios and conditions as well.

Next, a specific potential function was selected to conduct a large number of additional simulations. The Gaussian centers, which represent localized minima, were then varied. In the context of classical vortices, these centers can be likened to the drain of a sink, where water forms a vortex as it spirals down. For these tests, I altered the number of centers and their distances from the origin, ensuring that the centers were equally spaced at a specified radius. Additionally, the width and ellipticity of these centers were modified, with an ellipticity of one corresponding to a perfect circle, and values less than one creating an oval-like shape. For each potential function, I systematically varied the number, radius, width, and ellipticity of the centers, while saving all the live images and other outputs described earlier. Examples of varied first iteration outputs are given in Fig. 4. This process was repeated for multiple potential functions, including both time-independent and time-dependent potentials.

Once the data was collected, I developed a system to efficiently process the large volume of text files containing the array data. This was made possible in part by the way I initially coded the data-saving process. For instance, when running a simulation with a given potential and varying the number, radii, width, and ellipticity of Gaussian centers, these parameters were saved in a text file with a name corresponding to the specific values used. I designed the code to allow easy modification of text file names, further optimizing the efficiency of the simulations. This data was then read by another MATLAB script, which utilized for loops to generate specific graphs and extract results, as will be detailed in the results section.

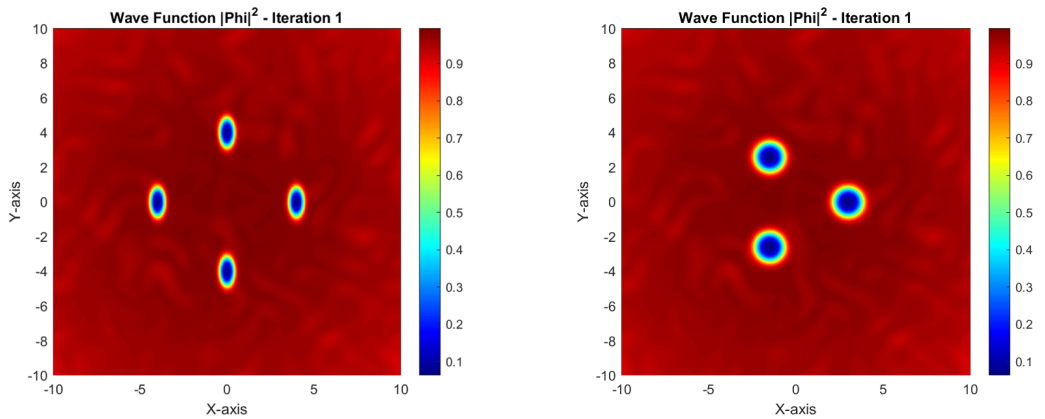


Fig. 4 $|\psi|^2$ of a simulation after the first iteration with 4 centers, ellipticity of 0.5, width of .4, and radii of 4 (left). $|\psi|^2$ of a simulation after the first iteration with 3 centers, ellipticity of 1, width of .4, and radii of 3 (right)

IV. Results

Since our system can only study something that exists in a reasonably long time, experiments can only be done for steady state solutions since non-steady states cannot be measured because they disappear too fast to be tracked by available measurement devices. Using these equations, one can find steady state solutions. For this example, the solution was sought of the linearized equation (11) in the Barenghi paper where $f(r) = r^b$ with f(r) being the magnitude of the wave function. $\psi(r,\theta) = f(r)e^{i\theta}$, where $f(r)=|\psi|$ satisfies $-(\hbar^2/2m)((d^2f/dr^2)+((1/r)df/dr)-(f/r^2))+gf^3-\mu f = 0$ (full nonlinear form). The linearized version has the only possible solutions of the form $f(r) = r^b$ with b = 1 and b = -1 which produces a Rankine combine vortex solution (Fig. 7 in Barenghi). For the full nonlinear equation and assumptions of regularity (f(r) is twice continuously differentiable) let's note that $f(r) \to 0$ as $r \to 0$ and $f(r) \to \sqrt{(\mu/g)}$ as $r \to \infty$, where r is the distance to the origin.

Initially, experiments were conducted on vortices under different potentials. One of the first observations was that as the magnitude of the wave function squared evolved into a concentrated center, the magnitude of the gradient took on a donut-like shape (Fig. 5). Upon comparing the gradient magnitude with the x and y-directional velocities, it became evident that this donut shape was, in fact, rotating. Therefore, the gradient magnitude's donut shape, combined with the concentrated area of the wave function's squared magnitude, can be interpreted as a quantum vortex—one of the more exciting discoveries in this research.

To gain a better understanding of which aspects to alter in the simulations, I closely examined GIFs generated for each one. However, since this is a paper, I was unable to include these GIFs. These visualizations guided my decisions on which potential functions and other variables to keep constant when conducting tests with varying Gaussian centers. Additionally, I created a random generator for Gaussian centers with various conditions and ran it through multiple scenarios. This approach further helped to identify the specific simulation scenarios I wanted to mass run for data analysis.

Another key observation from the GIFs was the visualization of spontaneous polarization when analyzing the phase of the wave function. Over time, or through iterations, the colors representing different phases gradually merged, providing a clear visual representation of spontaneous polarization—a fascinating outcome discussed earlier in this paper (Fig. 7). It's important to note that iterations can be interpreted as time, as I programmed the code to account for a specific delta T, or change in time, between each iteration; thus, as iterations progress, time advances proportionally.

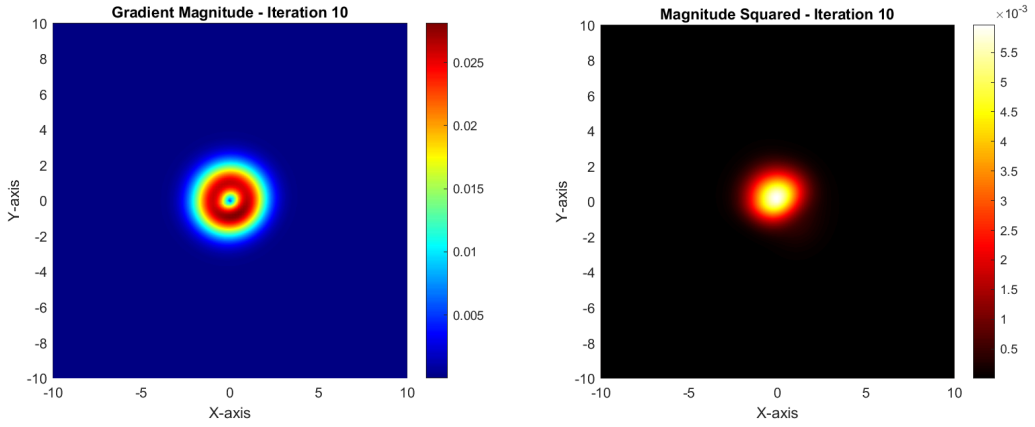


Fig. 5 Image Output of magnitude gradient of ϕ (left). Corresponding $|\psi|^2$ (right).

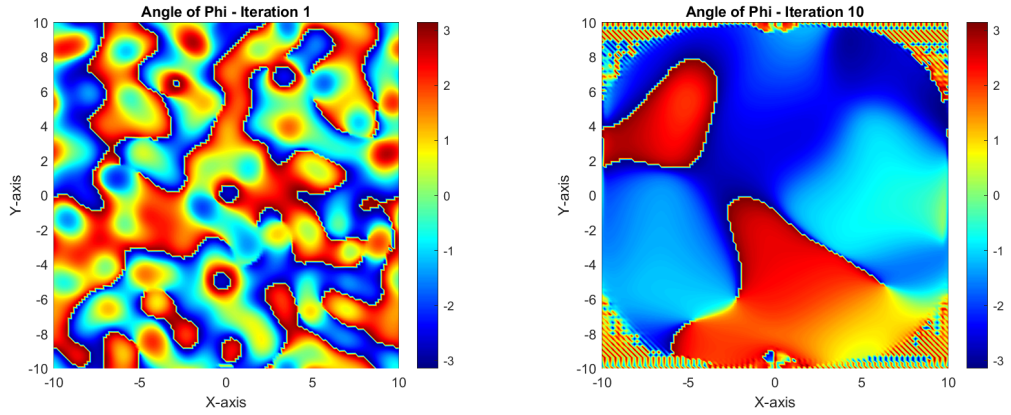


Fig. 7 Example phase of the ϕ for the first iteration (left) and the tenth iteration (right).

In addition to the mass data analysis conducted on varying Gaussian centers, their number, radii, width, and ellipticity were examined. The results for minimum energy and minimum angular momentum are presented below. Here, minimum angular momentum is understood as the maximum absolute value, since the max negative absolute value of angular momentum was always greater than the max positive value (at least for these tests). The graphs are annotated for clarity. It's important to note that 'radii' refers to the distance of the Gaussian centers from the origin. The potential function and all other variables were held constant during these tests to isolate the effects of the dependent variable on the data. Although additional data was recorded, such as vorticity norm and maximum gradient magnitude, this paper will focus on analyzing angular momentum and energy (unless otherwise stated) to conserve space.

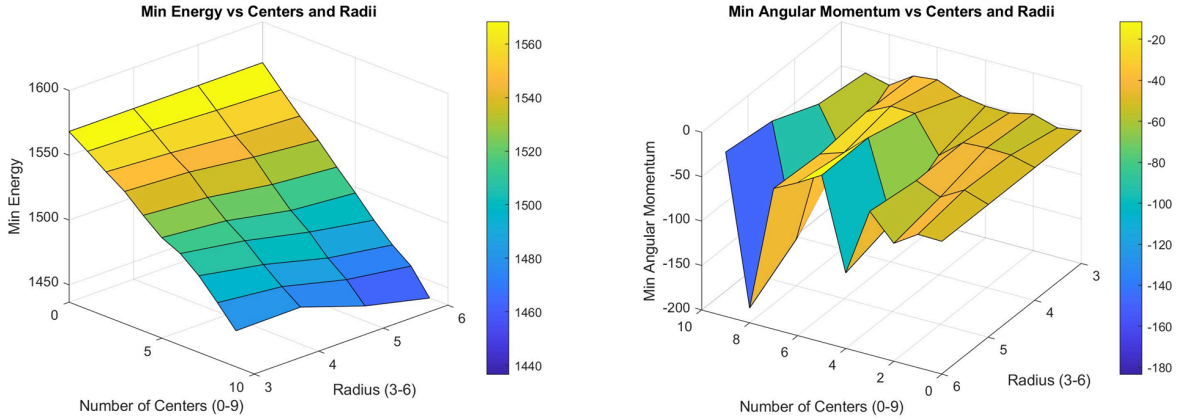


Fig. 8 Minimum Value of Energy (Left) and Angular Momentum (Right) vs. Number and Radii of Gaussian Centers. The Gaussian Width was kept constant at 0.3, and the Ellipticity at 1.

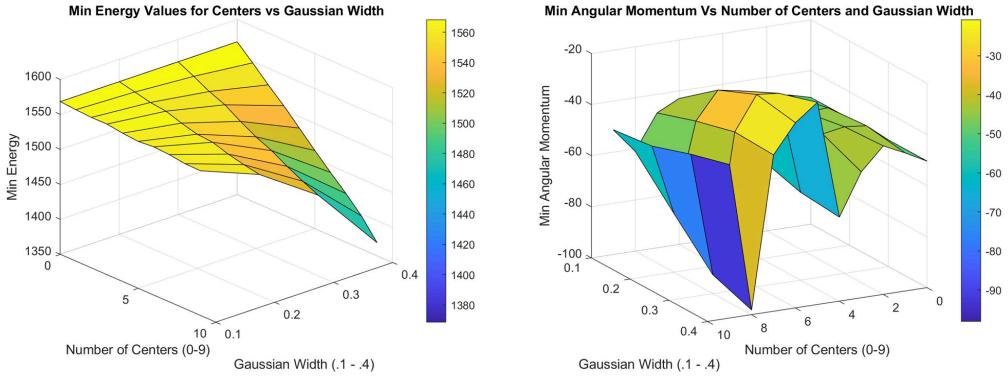


Fig. 9 Minimum Value of Energy (Left) and Angular Momentum (Right) vs. Number and Width of Gaussian Centers. The Radius was kept constant at 4, and the Ellipticity at 1.

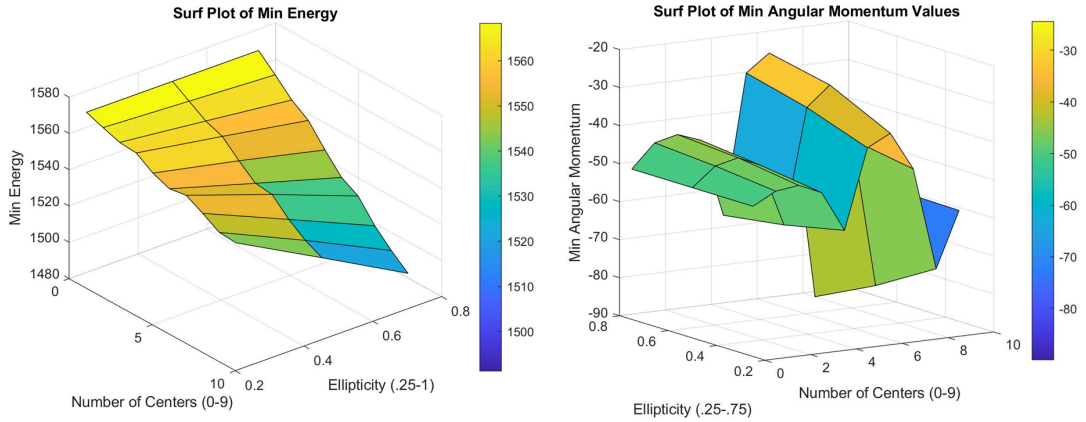


Fig. 10 Minimum Value of Energy (Left) and Angular Momentum (Right) vs. Number and Ellipticity of Gaussian Centers. The Radius was kept constant at 4, and the Width at 0.3.

Moreover, additional graphs were created to track the variation of certain values over the iterations (time). These graphs were essential for gaining a deeper understanding of how each variable influences the simulation outcomes. The annotated graphs are presented below for clarity. It's important to note that the energy values are represented by the lines closely grouped together in an exponential decay pattern, while the angular momentum values are more spaced out and resemble a quadratic function.

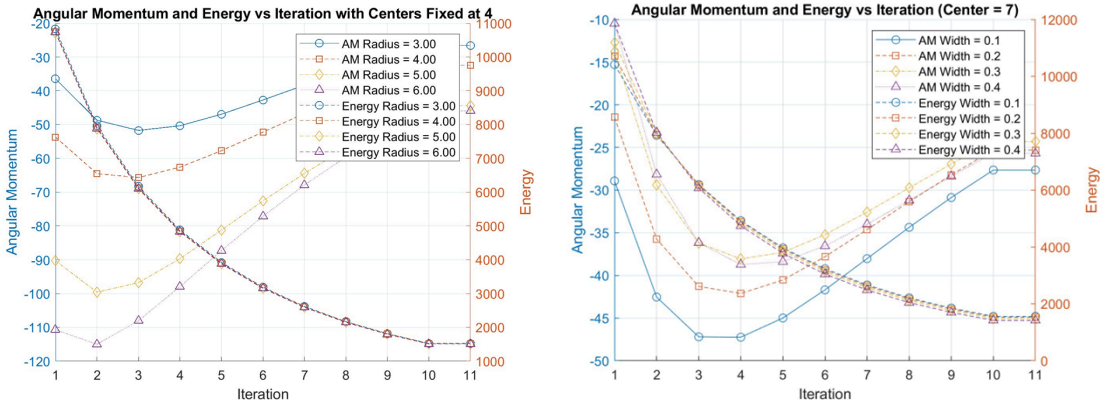


Fig. 11 Energy and Angular Momentum Values vs. Iterations.

In the examples above, the graph on the left (Fig. 11) shows simulations with four centers, a Gaussian width of 0.3, and an ellipticity of one. Each line represents a different radii value, allowing the observation of how radii affect the results under these specific conditions. The graph on the right (Fig. 11) illustrates simulations with seven centers, a radius of four, and an ellipticity of one. Here, each line corresponds to a different Gaussian width, providing insight into how width influences the outcomes when the radii, number, and ellipticity of the centers are held constant. Although hundreds of graphs were generated, only two are presented here due to space constraints. Despite this, these graphs were invaluable for deriving meaningful observations.

Moreover, a line of best quadratic fit was generated for each simulation. These lines were created for every simulation output, including energy, angular momentum, maximum magnitude gradient, and others. For instance, 44 simulations were run with varying numbers of centers (ranging from 0 to 10) and different radii distances (ranging from 3 to 6). The angular momentum value was calculated for each

iteration, and a line of best quadratic fit was produced for each simulation (Fig. 12). It's important to note that in the figure below, only one line of best fit is shown. This approach provided additional insights and observations.

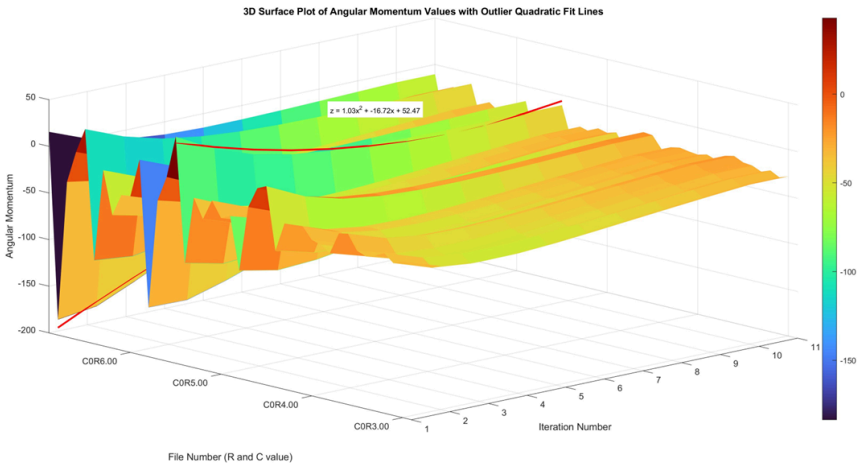


Fig. 12 Angular Momentum Values vs. Iterations and File Numbers. Each file corresponds with a different number and radius of centers.

One of the central goals of this project was to connect these GPE simulations to actual tornado data, and the figures below demonstrate this connection—an incredibly exciting finding. The left graph (Fig. 14) presents real tornado data, with the x-axis representing the radius from the center of the tornado and the y-axis representing the mean tangential velocity. Each line in the graph corresponds to data taken at various heights, where higher altitudes are associated with lower pressure. The right graph (Fig. 14) displays the results of the Gross-Pitaevskii simulation after the formation of a quantum vortex. The x-axis represents the distance from the origin, with the potential function centering the vortices at the origin. The y-axis shows the mean magnitude of the gradient, which can be interpreted as velocity. By referencing additional academic papers, the potential function was adjusted to theoretically mimic increasing pressure, with each line corresponding to a different pressure scaling. As shown in the graphs, the GPE simulations closely follow real-life trends, with higher pressure resulting in an earlier, more pronounced peak followed by a steeper decline. This alignment with real-world data is an extremely exciting discovery.

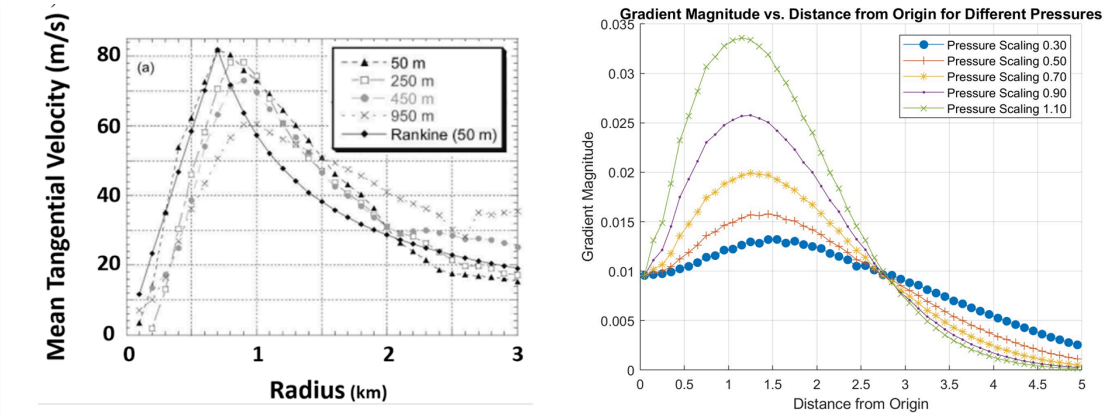


Fig. 14 Real Life Tornado Data (Left). GPE Data (Right).

Connections between the two graphs below (Fig. 15) highlight how helicity, both in simulated quantum vortices and real-world tornadoes, varies based on specific conditions. Helicity in fluid dynamics is mathematically defined as $H=\int v \cdot (\nabla \times v)\, dV$, v is the velocity vector, $\nabla \times v$ is the vorticity vector, and dV represents a volume element. In the below right plot (Fig. 15) of calculated helicity, the helicity values fluctuate depending on the number of Gaussian centers and their radii. This sensitivity to parameters in the quantum simulation mirrors how tornado helicity might change based on physical characteristics like the structure and distribution of vortices. Similarly, the storm-relative helicity box plot (real life tornado data) (Fig. 15) demonstrates that helicity changes with height (0-1 km vs. 0-3 km) and storm type (isolated vs. multiple), indicating that real-world tornado helicity is influenced by atmospheric conditions. These quantum simulations potentially provide a controlled environment to explore and observe helicity behavior, offering insights that could enhance our understanding of helicity in natural tornadoes. Let's note that the lower right graph features different variable values compared to the previous graphs, but the number of centers and radii were still varied.

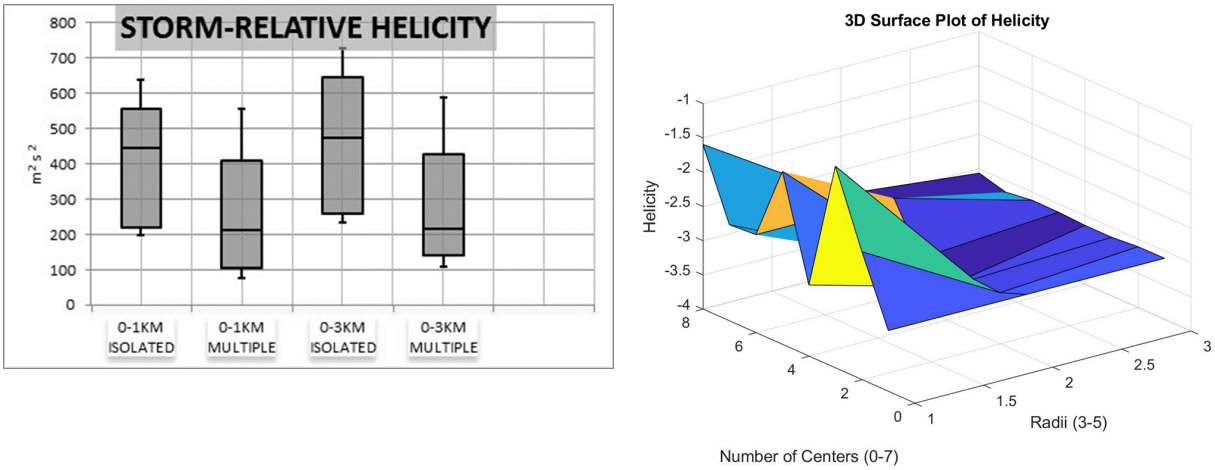


Fig. 15 Real Life Tornado Data (Left). GPE Data (Right).

V. Future Plans

Using the phase of the wave function, this project theorized how the idea of spontaneous partial polarization could exactly be related to vortices formed. Using the simulation models, one could attempt to achieve a better estimate on how close the position and phase of vortices must be for this polarization to occur. Furthermore, the change in velocity and momenta could be calculated with the mergence. In the future, more simulations need to be tested and observed.

In addition, in the future, a three-dimensional model could be useful to continue the research. A three-dimensional simulation could give outputs of a live figure of the magnitude of $|\psi|^2$, the angle ϕ, and the gradient of ϕ. Thus, three-dimensional columns (gaussian centers in two dimensions) could be programmed with varying direction, width, and length. More conclusions could be made based on how the columns interacted, and the wave function would be calculated accordingly. Hence, further deepening our understanding of vortices.

Using Bose-Einstein statistics, one could potentially find a connection between quantum vortices and the density of tornadoes. Using the storm data from the storm prediction center, the density can be calculated using classical methods. Density, using Bose-Einstein statistics, could be represented by the amount of particles at a given energy level. Temperature data can be brought into the model using the energy conservation and switching to a baroclinic model of the flow where density depends not only on pressure but also at temperature. Thus, based on what number of particles most resemble a tornado (or the atmosphere), using Bose-Einstein statistics, one could calculate the number of configurations that would

yield a given amount of particles at a specific energy level. Furthermore, comparing simulations of differing number of particles could give us results on the differences in lower and higher density tornadoes, where a lower number of particles represent a lower density and higher number of particles represent higher density. Also, these results may give more insight on the relationship between quantum and classical pressure.

Moreover, access to accurate past live feeds of tornadoes would further increase the accuracy of the research. Observing a tornado's past live feed with the directional velocities could give us more insight on vortices merging. Thus, one can calculate the experimental and theoretical values. Then, from the difference in values, attempt to predict the quantum effects in the tornado. More comprehensive data on tornadoes would be invaluable, as there is currently limited information available online. This data would greatly enhance the accuracy and depth of future research, allowing for more precise comparisons.

On top of this, utilizing AI and supercomputers to compare simulations could significantly enhance our understanding by analyzing an extreme number of simulations. This approach could deepen our comprehension of the relationship between the Gross-Pitaevskii equation, classical vortices, and tornadogenesis. Additionally, incorporating Gaussian center maxima into the simulations could yield intriguing results. This feature has been successfully coded, and further exploration could provide valuable insights. It's important to note that the Gaussian centers used in this study were localized minima, not necessarily localized maxima.

Upgrading the Navier-Stokes equation model could also produce interesting outcomes. For this project, an existing MATLAB Navier-Stokes equation simulation was initially altered from online sources, but work on it was suspended due to time constraints. Improving this model would be an exciting direction for future research. Given that both models are programmed in MATLAB, further connections between the Gross-Pitaevskii equation and classical fluid dynamics could be explored. However, this would require significant additional time to refine the Navier-Stokes equation model.

VI. Conclusion / Findings

To conclude this research, several key findings were observed. Minimum energy generally decreased as the number of Gaussian centers increased, with the only exception occurring when ellipticity was small. Additionally, minimum energy consistently decreased with increasing Gaussian width and ellipticity. The study also revealed that if the maximum absolute value of angular momentum increases with the number of centers, there is a corresponding increase in radii distance, Gaussian width, and ellipticity. Conversely, the reverse trend was also observed.

In certain scenarios, angular momentum was found to be inversely proportional to energy. Moreover, the pressure trends in velocity versus radii mirrored real-life tornado data, highlighting the relevance of the simulations. The study successfully visualized spontaneous polarization and quantum vortices by varying the potential function, number, radii, and types of centers. The number and characteristics of Gaussian centers were shown to significantly influence angular momentum, emphasizing their importance in these simulations. Overall, this research establishes a strong foundation for future studies, with extensive data saved to enable detailed observations and further exploration. Ultimately, this project hopes to aid in improving tornado warning systems, with the ultimate goal of potentially saving lives.

Bibliography/References

Antoine, X., & Duboscq, R. (2014). *GPELab, a Matlab toolbox to solve Gross-Pitaevskii equations I: Computation of stationary solutions*. Université de Lorraine, Institut Elie Cartan de Lorraine, UMR 7502, Vandoeuvre-lès-Nancy, F-54506, France; Inria Nancy Grand-Est/IECL - ALICE; Inria Nancy Grand-Est/IECL - CORID.

Barenghi, Carlo F. Tangled Vortex Lines: Dynamics, Geometry and Topology of Quantum Turbulence.

Brachet, M., Sadaka, G., Zhang, Z., Kalt, V., & Danaila, I. (2023). Coupling Navier-Stokes and Gross-Pitaevskii equations for the numerical simulation of two-fluid quantum flows. ENS, Université PSL, CNRS, Sorbonne Université, Université de Paris, Laboratoire de Physique de l'École Normale Supérieure, F-75005 Paris, France; Univ Rouen Normandie, CNRS, Laboratoire de Mathématiques Raphaël Salem, UMR 6085, F-76000 Rouen, France.

Chopra, Abhishek. "Can Quantum Fluids Explain Turbulence in Classical Fluids?" BosonQ Psi, 20 Jan. 2022, www.bosonqpsi.com/post/can-quantum-fluids-explain-turbulence-in-classical-fluids.

Chorin, Alexandre, and Jerrold E Marsden. A Mathematical Introduction to Fluid Mechanics. Third ed., Springer-Verlag Publishing Company, Inc., 1992.

Gamillo, Elizabeth. "MIT Physicists Formed Quantum Tornadoes by Spinning Ultra-Cold Atoms." Smithsonian Magazine, Smithsonian Institution, 13 Jan. 2022, www.smithsonianmag.com/smart-news/mit-physicists-formed-quantum-tornados-by-spinning-ultra-cold-atoms-180979388/.

Griffiths, David J. Introduction to Quantum Mechanics. Edited by Rose Kernan and Fred Dahl, Prentice Hall, Inc., 1995.

"NOAA/NWS Storm Prediction Center." NOAA/NWS Storm Prediction Center, www.spc.noaa.gov/. Accessed 17 Aug. 2023.

Sasaki, Yoshi K. "Entropic Balance Theory and Variational Field Lagrangian Formalism: Tornadogenesis." Journal of the Atmospheric Sciences, vol. 71, 30 Jan. 2014, pp. 2104–2113.

"12.3 Second Law of Thermodynamics: Entropy." OpenStax, https://openstax.org/books/physics/pages/12-3-second-law-of-thermodynamics-entropy#:~:text=The%20second%20law%20of%20thermodynamics%20states%20that%20the,objects%2C%20but%20never%20spontaneously%20in%20the%20reverse%20direction. Accessed 17 Aug. 2023.

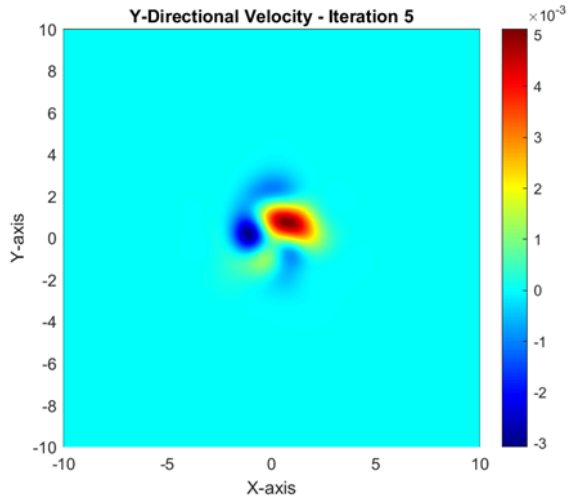
"2D Unsteady Navier-Stokes." *File Exchange - MATLAB Central*, www.mathworks.com/matlabcentral/fileexchange/60869-2d-unsteady-navier-stokes. Accessed 13 Aug. 2024.

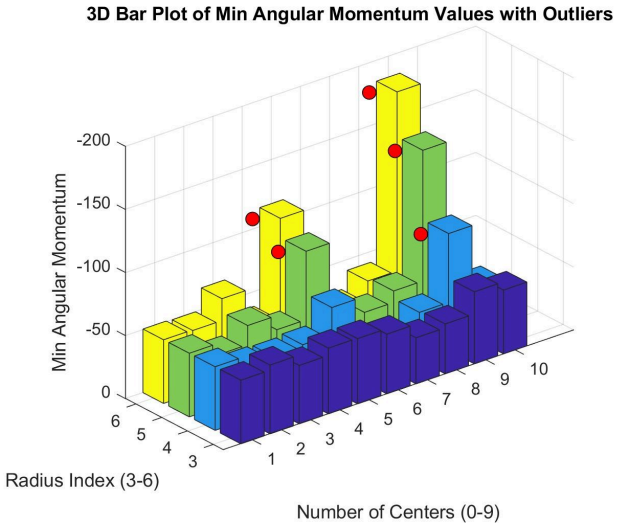
The attached file includes many of the live image outputs generated during the course of this research, providing visual insights into the quantum vortex simulations. Additionally, it contains images and descriptions that explore various alternative directions the research could have taken, illustrating the range of possibilities considered and the breadth of the study's potential impact.

Appendix / Addition File

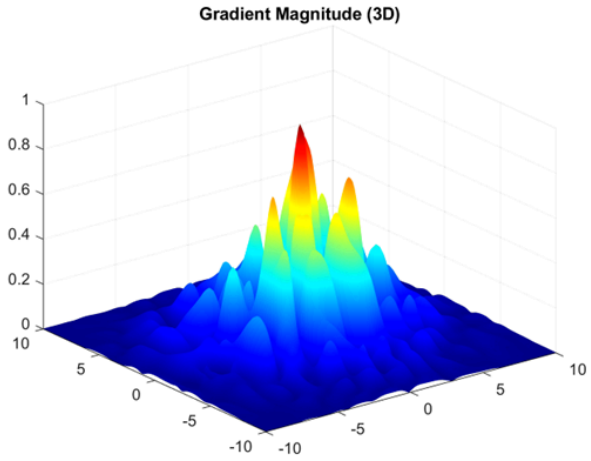
Colin Fjelsted & Professor Mikhail Shvartsman

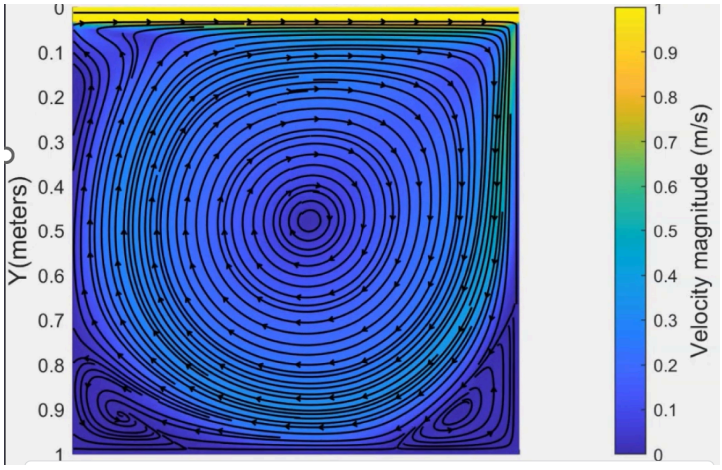
This appendix includes images referenced in the main paper, along with additional outputs generated from various ideas explored during the research process. Following the appendix, example codes are also provided.



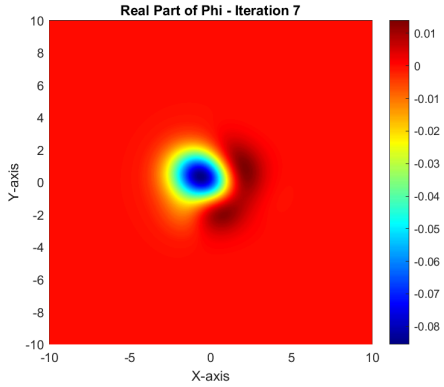


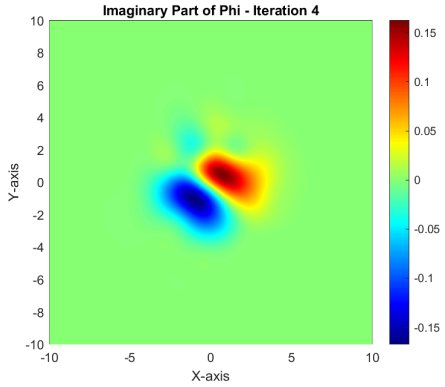
App. 1. Example Live Image Output of Y-directional Velocity (Left), Alternate view of Interpreting Angular Momentum Values (Right)



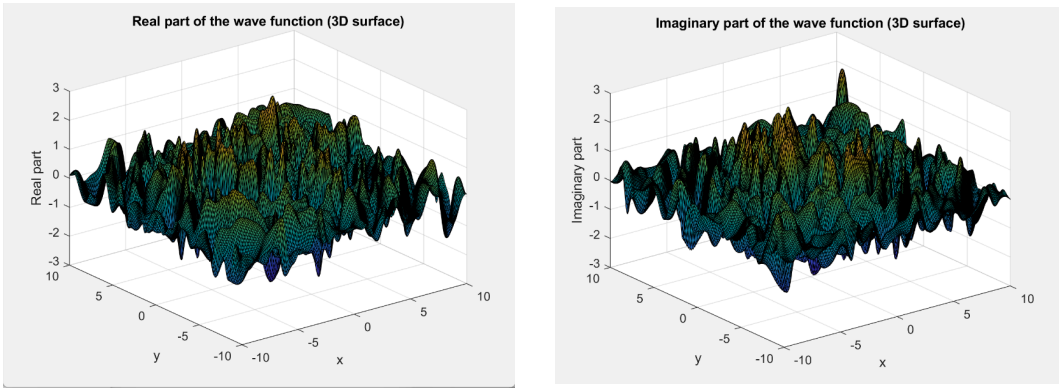


App. 2. 3-Dimensional Version of Gradient Magnitude Plot (Left), MATLAB Navier-Stokes Equation Simulator (Right)

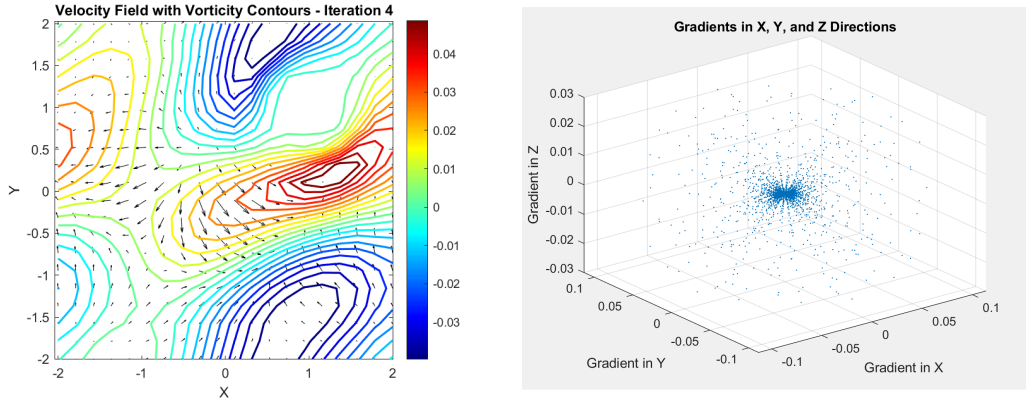




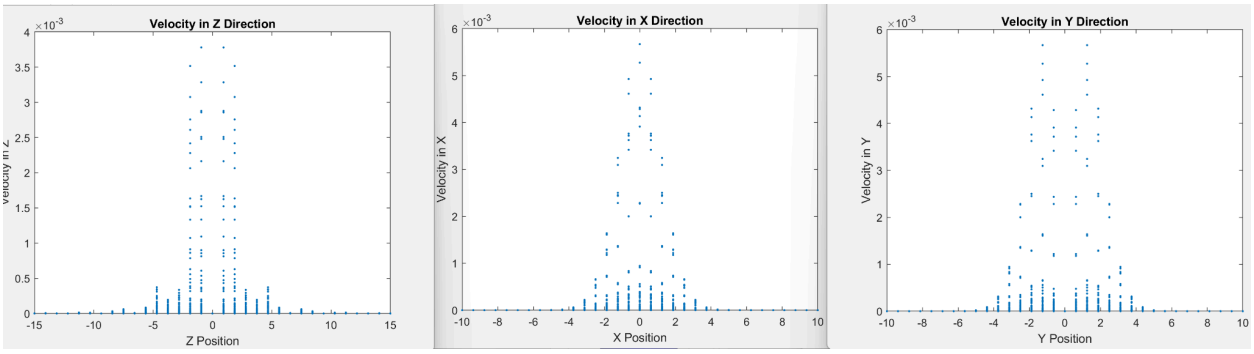
App. 3. Real Part (Left) and Imaginary Part (Right) of the Wave Function



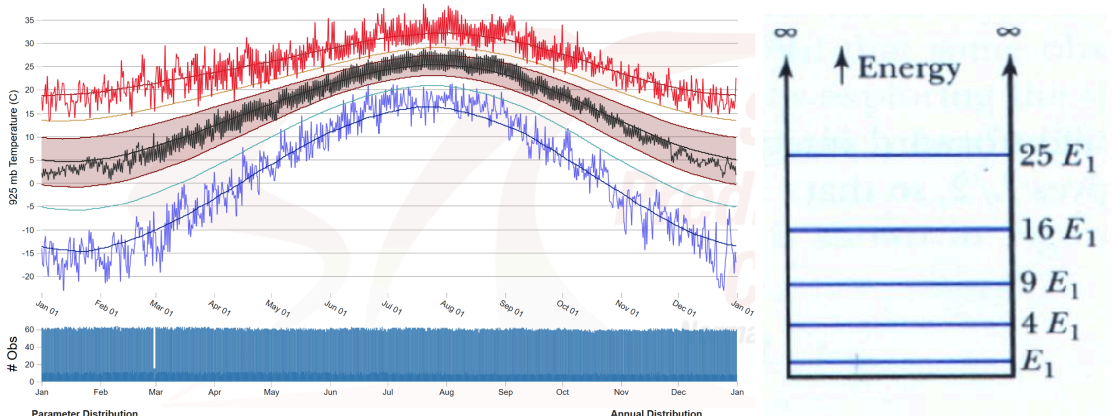
App. 4. 3-Dimensional View of Real Part (Left) and Imaginary Part (Right) of the Wave Function



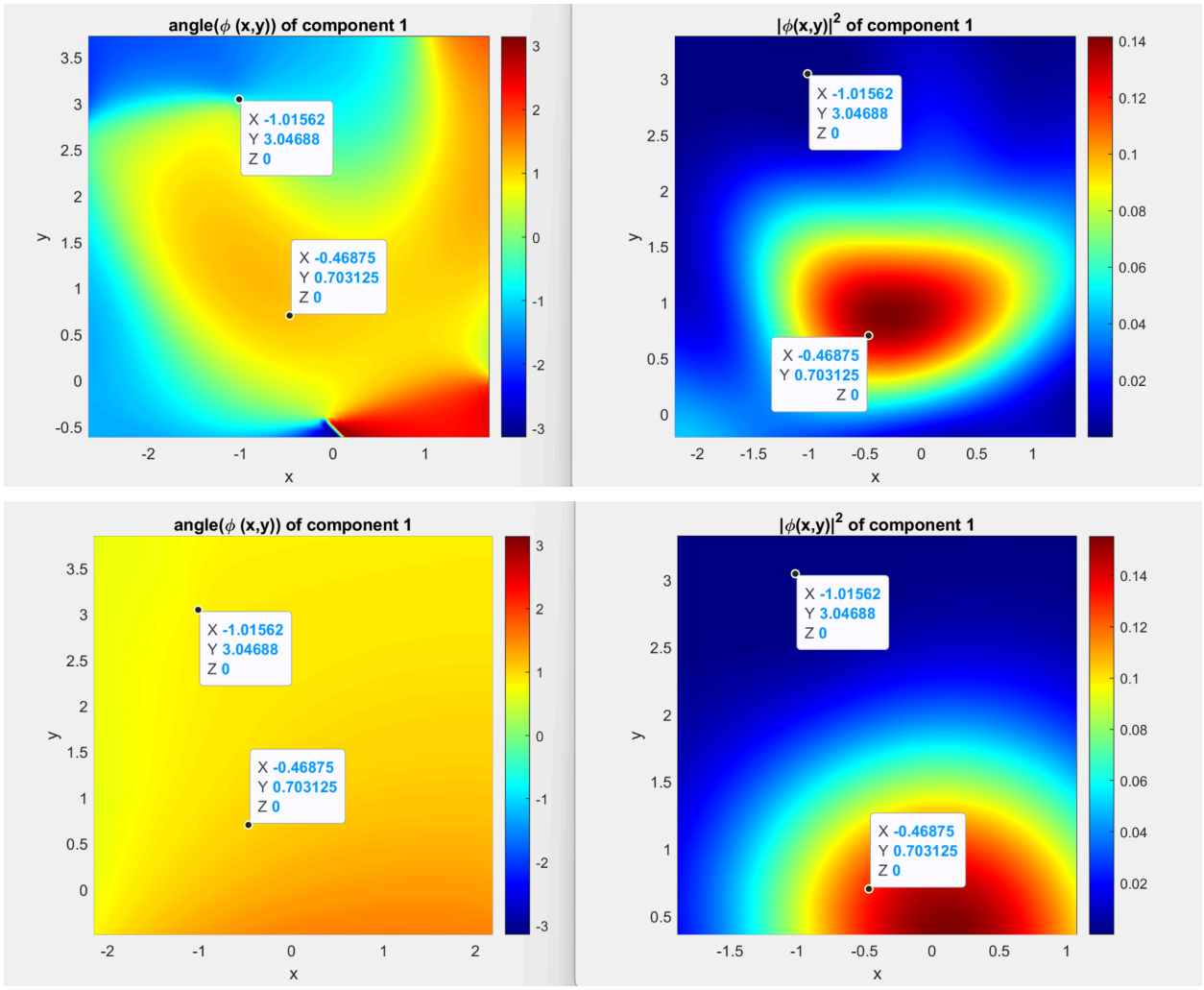
App. 5. Velocity Field with Vorticity Contours (Left), Gradient Calculation for a Three-Dimensional Simulation (Right)



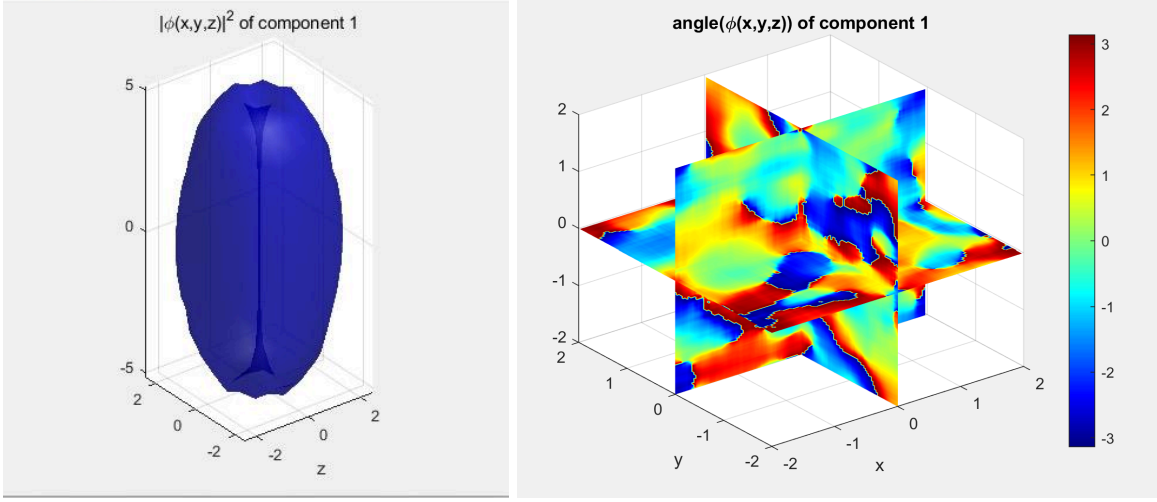
App. 6. Z-directional (Left), X-directional (Center), and Y-directional (Right) Velocities of a Three-Dimensional Simulation



App. 7. Storm data from SPC (Storm Prediction Center) of the temperature at 925 mb for the year 2001 (Left); energy levels within an atom (BEC particles can occupy any energy state)(Right).



App. 8. Example of simulation tracking the angle ϕ and $|\psi|^2$ comparing an earlier iteration (Top) with a later iteration (Bottom).



App. 9. Three-Dimensional Simulation

Example Code for Simulation

Please note that the code relies on functions and interfaces that are not explicitly defined within the code itself.

```
% % Description of what the code does
description = [
  "This MATLAB script performs a simulation of a quantum system using the
Gross-Pitaevskii Equation (GPE) in two dimensions. ", newline, ...
  "It begins by initializing directories and filenames for saving output
images, videos, and text files on the desktop. ", newline, ...
  "The script sets up the computational parameters, including the type of
computation (dynamic), the number of components, ", newline, ...
  "time step, and total simulation time. It defines the geometry of the
simulation domain and sets up the physical problem ", newline, ...
  "parameters, such as dispersion, nonlinearity, and a potential function with
random Gaussian centers.", newline, newline, ...
  "The script generates an initial random phase and runs the simulation for a
specified number of iterations, storing the solutions ", newline, ...
  "at each step. During each iteration, it calculates various quantities like
the magnitude and angle of the wave function, x- and y-directional ", newline,
...
  "velocities, and the gradient magnitude. These quantities are saved as images
with corresponding iteration numbers.", newline, newline, ...
  "The script then creates GIFs from the saved images and generates videos of
the magnitude of the wave function and the gradient magnitude ", newline, ...
  "using 2D and 3D plots. It also plots and saves various physical quantities
like energy, angular momentum, and RMS values as a combined ", newline, ...
  "subplot figure, which is saved as a PDF.", newline, newline, ...
  "Finally, the script writes the entire simulation code to a text file and
includes references to the generated images, GIFs, videos, and the ", newline,
...
  "subplot figure. This text file serves as a simple, accessible report of the
simulation outputs and the corresponding code, with bold headings for
clarity.", newline, newline
];
rng(42); % Use a fixed seed value
% Initialize directories and filenames
output_dir = fullfile(getenv('USERPROFILE'), 'Desktop',
'phi_imagesfortestin2g');
gif_filename_magnitude = fullfile(output_dir, 'phi_squared_video.gif');
gif_filename_angle = fullfile(output_dir, 'phi_angle_video.gif');
gif_filename_velocity_x = fullfile(output_dir, 'phi_velocity_x_video.gif');
gif_filename_velocity_y = fullfile(output_dir, 'phi_velocity_y_video.gif');
gif_filename_gradient_magnitude = fullfile(output_dir,
'phi_gradient_magnitude_video.gif');
gif_filename_spin_vector = fullfile(output_dir, 'phi_spin_vector_video.gif');
video_filename = fullfile(output_dir, 'phi_video.avi');
video_filename_gradient_magnitude = fullfile(output_dir,
'phi_gradient_magnitude_video.avi');
```

```
video_filename_gradient_magnitude_surf = fullfile(output_dir,
'phi_gradient_magnitude_surf_video.avi');
text_filename = fullfile(output_dir, 'simulation_outputs.txt');
code_filename = fullfile(output_dir, 'simulation_code.txt');
pdf_filename = fullfile(output_dir, 'simulation_outputs.pdf');
%results_filename = fullfile(output_dir, 'simulation_results.txt'); % Define
the text file filename for the results
%results_filename = fullfile(output_dir,
sprintf('longernewresults_width%.2f_decay%.2f_ellipticity%.2fnumCenters%.2fradi
us%.2f.txtellispeangle%.2f.txt', gaussian_width, decay_time, ellipticity,
numCenters, radius, ellipse_angle));
% Ensure output directory exists
if ~exist(output_dir, 'dir')
  mkdir(output_dir);
end
% Settings for the method and geometry
Computation = 'Dynamic';
Ncomponents = 1;
Type = 'BESP';
Deltat = .003;
Stop_time = .03;
Stop_crit = [];
xmin = -10;
xmax = 10;
ymin = -10;
ymax = 10;
Nx = 2^7+1;
Ny = 2^7+1;
% Physical problem settings
Delta = 1;
Beta = 10;
Omega = 100;
V0 = 10000;
d = 0.3;
W0 = 0;
Z0 = 0;
numCenters = 4;
radius = 4; % Radius of the circle around the origin
% Save the initial part of the script as a text file
code_text = [
  "%% Initialize directories and filenames", newline, ...
  "output_dir = fullfile(getenv('USERPROFILE'), 'Desktop', 'phi_images');",
newline, ...
  "gif_filename_magnitude = fullfile(output_dir, 'phi_squared_video.gif');",
newline, ...
  "gif_filename_angle = fullfile(output_dir, 'phi_angle_video.gif');", newline,
...
  "gif_filename_velocity_x = fullfile(output_dir,
'phi_velocity_x_video.gif');", newline, ...
```

```
  "gif_filename_velocity_y = fullfile(output_dir,
'phi_velocity_y_video.gif');", newline, ...
  "gif_filename_gradient_magnitude = fullfile(output_dir,
'phi_gradient_magnitude_video.gif');", newline, ...
  "gif_filename_spin_vector = fullfile(output_dir,
'phi_spin_vector_video.gif');", newline, ...
  "video_filename = fullfile(output_dir, 'phi_video.avi');", newline, ...
  "video_filename_gradient_magnitude = fullfile(output_dir,
'phi_gradient_magnitude_video.avi');", newline, ...
  "video_filename_gradient_magnitude_surf = fullfile(output_dir,
'phi_gradient_magnitude_surf_video.avi');", newline, ...
  newline, ...
  "%% Setting the method and geometry", newline, ...
  "Computation = '" + Computation + "';", newline, ...
  "Ncomponents = " + num2str(Ncomponents) + ";", newline, ...
  "Type = '" + Type + "';", newline, ...
  "Deltat = " + num2str(Deltat) + ";", newline, ...
  "Stop_time = " + num2str(Stop_time) + ";", newline, ...
  "Stop_crit = [];", newline, ...
  "Method = Method_Var2d(Computation, Ncomponents, Type, Deltat, Stop_time,
Stop_crit);", newline, ...
  newline, ...
  "xmin = " + num2str(xmin) + ";", newline, ...
  "xmax = " + num2str(xmax) + ";", newline, ...
  "ymin = " + num2str(ymin) + ";", newline, ...
  "ymax = " + num2str(ymax) + ";", newline, ...
  "Nx = " + num2str(Nx) + ";", newline, ...
  "Ny = " + num2str(Ny) + ";", newline, ...
  "Geometry2D = Geometry2D_Var2d(xmin, xmax, ymin, ymax, Nx, Ny);", newline,
...
  newline, ...
  "%% Setting the physical problem", newline, ...
  "Delta = " + num2str(Delta) + ";", newline, ...
  "Beta = " + num2str(Beta) + ";", newline, ...
  "Omega = " + num2str(Omega) + ";", newline, ...
  "V0 = " + num2str(V0) + ";", newline, ...
  "d = " + num2str(d) + ";", newline, ...
  "W0 = " + num2str(W0) + ";", newline, ...
  "Z0 = " + num2str(Z0) + ";", newline, ...
  "Physics2D = Physics2D_Var2d(Method, Delta, Beta, Omega);", newline, ...
  "Physics2D = Dispersion_Var2d(Method, Physics2D);", newline, ...
  newline, ...
  "%% Setting a potential function with random Gaussian centers", newline, ...
  "numCenters = randi([1, 8]);", newline, ...
  "PotentialFunction = @(X,Y) (1/2)*(X.^2+Y.^2);", newline, ...
  "for k = 1:numCenters", newline, ...
  "    locX = randi([-10, 10]);", newline, ...
  "    locY = randi([-10, 10]);", newline, ...
```

```
  "    [X, Y] = meshgrid(linspace(xmin, xmax, Nx), linspace(ymin, ymax, Ny));",
newline, ...
  "    PotentialFunction = @(X,Y) PotentialFunction(X,Y) + V0*exp(-((X -
locX).^2+(Y - locY).^2)/d^2);", newline, ...
  "end", newline, ...
  "Physics2D = Potential_Var2d(Method, Physics2D, PotentialFunction);",
newline, ...
  newline, ...
  "Random_phase = Stationary_Gaussian_Field2d(Geometry2D, @(X,Y)
exp(-(X.^2+Y.^2)/2));", newline, ...
  "Random_phase = Random_phase / max(max(Random_phase));", newline, ...
  "Phi_0{1} = exp(-2i*pi*Random_phase);"
];
fileID = fopen(code_filename, 'w');
fprintf(fileID, '%s\n', description, code_text);
fclose(fileID);
% Calculate the positions of the centers
theta = linspace(0, 2*pi, numCenters + 1); % Divide the circle into equal
segments
theta(end) = []; % Remove the last point because it coincides with the first
one
locX_list = radius * cos(theta);
locY_list = radius * sin(theta);
% Define additional parameters for the Gaussian centers
gaussian_width = 0.4; % Width of the Gaussian centers
decay_time = 1; % Time decay factor of the Gaussian centers
ellipticity = .5; % Ellipticity factor (1 for circular, >1 for elliptical)
ellipse_angle = 0; % Angle for all ellipses in radians
% Define the potential function with a constant omega for the harmonic
potential
omega = 1; % Constant omega for harmonic potential
% Define a simple pressure scaling factor
pressure_scaling_factor = .5; % Adjust this factor to simulate different
pressures
% Define the potential function for tornado-like vortex
A = 0.01; % Radial strength constant
B = 0.01; % Tangential strength constant
C = 0.1; % Axial strength constant
epsilon = 1e-6; % Small constant to avoid log(0)
%%%%%%%%%%%%%%%%%%%%%%PotentialFunction = @(X, Y) pressure_scaling_factor *
(1/2) * (X.^2 + Y.^2);
%{ Create the potential function with elliptical Gaussian centers
R = [cos(ellipse_angle), -sin(ellipse_angle); sin(ellipse_angle),
cos(ellipse_angle)];
%{ Create the potential function with elliptical Gaussian centers
PotentialFunction = @(X, Y) 0;
for k = 1:numCenters
  locX = locX_list(k);
  locY = locY_list(k);
```

```
  % Add each elliptical Gaussian center to the potential function
  PotentialFunction = @(X, Y) PotentialFunction(X, Y) + ...
      V0 * exp(-((R(1,1) * (X - locX) + R(1,2) * (Y - locY)).^2 / (ellipticity
* gaussian_width)^2 + ...
                (R(2,1) * (X - locX) + R(2,2) * (Y - locY)).^2 /
gaussian_width^2) / (2 * decay_time^2));
end
%}
results_filename = fullfile(output_dir,
sprintf('timeonlydependentpotentiallongernewresults_width%.2f_decay%.2f_ellipti
city%.2fnumCenters%.2fradius%.2f.txtellispeangle%.2f.txt', gaussian_width,
decay_time, ellipticity, numCenters, radius, ellipse_angle));
% Continue with the rest of the code
Method = Method_Var2d(Computation, Ncomponents, Type, Deltat, Stop_time,
Stop_crit);
Geometry2D = Geometry2D_Var2d(xmin, xmax, ymin, ymax, Nx, Ny);
Physics2D = Physics2D_Var2d(Method, Delta, Beta, Omega);
Physics2D = Dispersion_Var2d(Method, Physics2D);
Physics2D = Potential_Var2d(Method, Physics2D, PotentialFunction);
%Example_Timepotential = @(t,x,y) (1/2+cos(t)).*(x.^2 + y.^2);
% Rotating vortex potential
VortexPotential = @(t, x, y) (1/2) * ((x .* cos(t) - y .* sin(t)).^2 + (x .*
sin(t) + y .* cos(t)).^2);
Physics2D = TimePotential_Var2d(Method, Physics2D, VortexPotential);
Random_phase = Stationary_Gaussian_Field2d(Geometry2D, @(X,Y)
exp(-(X.^2+Y.^2)/2));
Random_phase = Random_phase / max(max(Random_phase));
Phi_0{1} = exp(-2i*pi*Random_phase);
% Define the Figure structure
Figure.label = 1;
Figure.map = 'jet';
Figure.x = 'X-axis';
Figure.y = 'Y-axis';
Figure.title = 'Wave Function |Phi|^2 - Iteration ';
Figure.axis = [];
%% Setting informations and outputs
custom_function = @(phi, x, y, fftx, ffty) Example_outputs(Geometry2D, phi, x,
y, fftx, ffty);
custom_function2 = @(phi, x, y, fftx, ffty)
computeVorticityAtOrigin(Geometry2D, phi, x, y, fftx, ffty);
Outputs = OutputsINI_Var2d(Method, 1, 1, {custom_function, custom_function2},
{'Grad Norm', 'Vorticity Norm'});
Printing = 1;
% Adjust the number of iterations here to make the GIF video shorter
Evo = 1; % Change this value to set the desired number of iterations
Draw = 1;
Print = Print_Var2d(Printing, Evo, Draw);
Physics2D = Nonlinearity_Var2d(Method, Physics2D);
%-----------------------------------------------------------
```

```
% Launching simulation and capturing intermediate states
%-------------------------------------------------------------
Figure.label = 1; % Assuming the figure label is 1
Figure.map = 'jet'; % Define the colormap as 'jet' for rainbow colors
Figure.x = 'X-axis';
Figure.y = 'Y-axis';
Figure.title = 'Wave Function |Phi|^2 - Iteration '; % Base title
Figure.axis = []; % Adjust the axis if necessary
% Initialize storage for solutions, velocities, and gradient magnitudes
Solution_storage = cell(Evo, 1);
Outputs_storage = cell(Evo, 1); % Storage for custom function outputs
% Initialize storage for simulation outputs
spin_vector_magnitudes = zeros(Evo, 1); % Initialize storage for spin vector
magnitudes
max_velocity_x = zeros(Evo, 1); % Initialize storage for max x-directional
velocity
max_velocity_y = zeros(Evo, 1); % Initialize storage for max y-directional
velocity
max_gradient_magnitude = zeros(Evo, 1); % Initialize storage for max gradient
magnitude
% Initialize storage for energy and angular momentum
energy_storage = zeros(Evo, 1);
angular_momentum_storage = zeros(Evo, 1);
% Run the simulation and store solutions at each iteration
for iter = 1:Evo
  [Phi, Outputs] = GPELab2d(Phi_0, Method, Geometry2D, Physics2D, Outputs, [],
Print);
  Solution_storage{iter} = Phi{1}; % Ensure we are storing the correct
solutions
  Outputs_storage{iter} = Outputs.User_defined_local; % Store the custom
function outputs
  % Get the current |phi|^2, angle(phi), x-directional velocity, y-directional
velocity, and gradient magnitude values from Solution_storage
  phi_squared_current = abs(Solution_storage{iter}).^2;
  phi_angle_current = angle(Solution_storage{iter});
   % Calculate the x-directional and y-directional velocities
  [phi_x, phi_y] = gradient(Solution_storage{iter}, Geometry2D.dx,
Geometry2D.dy);
  phi_velocity_x_current = -imag(phi_x .* conj(Solution_storage{iter}));
  phi_velocity_y_current = -imag(phi_y .* conj(Solution_storage{iter}));
  % Calculate the gradient magnitude
  gradient_magnitude_current = sqrt(abs(phi_x).^2 + abs(phi_y).^2);
  % Calculate and store the max x-directional velocity, y-directional velocity,
and gradient magnitude
  max_velocity_x(iter) = max(phi_velocity_x_current(:));
  max_velocity_y(iter) = max(phi_velocity_y_current(:));
  max_gradient_magnitude(iter) = max(gradient_magnitude_current(:));
  % Store the energy and angular momentum
  energy_storage(iter) = Outputs.Energy{1}(1);
```

```
  angular_momentum_storage(iter) = Outputs.Angular_momentum{1}(1);
  % Draw and save the figure for magnitude in the background
  draw_function_2d(phi_squared_current, Geometry2D, Figure, iter, 'Magnitude',
'jet');
   % Save the magnitude figure as an image
  filename_magnitude = fullfile(output_dir, sprintf('frame_magnitude_%04d.png',
iter));
  saveas(gcf, filename_magnitude);
   % Close the figure to avoid displaying it
  close(gcf);
  % Draw and save the figure for angle in the background
  draw_function_2d(phi_angle_current, Geometry2D, Figure, iter, 'Angle',
'jet');
   % Save the angle figure as an image
  filename_angle = fullfile(output_dir, sprintf('frame_angle_%04d.png', iter));
  saveas(gcf, filename_angle);
   % Close the figure to avoid displaying it
  close(gcf);
   % Draw and save the figure for x-directional velocity in the background
  draw_function_2d(phi_velocity_x_current, Geometry2D, Figure, iter, 'Velocity
X', 'jet');
   % Save the x-directional velocity figure as an image
  filename_velocity_x = fullfile(output_dir,
sprintf('frame_velocity_x_%04d.png', iter));
  saveas(gcf, filename_velocity_x);
   % Close the figure to avoid displaying it
  close(gcf);
   % Draw and save the figure for y-directional velocity in the background
  draw_function_2d(phi_velocity_y_current, Geometry2D, Figure, iter, 'Velocity
Y', 'jet');
   % Save the y-directional velocity figure as an image
  filename_velocity_y = fullfile(output_dir,
sprintf('frame_velocity_y_%04d.png', iter));
  saveas(gcf, filename_velocity_y);
   % Close the figure to avoid displaying it
  close(gcf);
   % Draw and save the figure for gradient magnitude in the background
  draw_function_2d(gradient_magnitude_current, Geometry2D, Figure, iter,
'Gradient Magnitude', 'jet');
   % Save the gradient magnitude figure as an image
  filename_gradient_magnitude = fullfile(output_dir,
sprintf('frame_gradient_magnitude_%04d.png', iter));
  saveas(gcf, filename_gradient_magnitude);
   % Close the figure to avoid displaying it
  close(gcf);
  % Compute and save the spin vector plot
  Phi{2} = Phi{1}; % Assuming the same wave function for both components for
spin calculation
```

```
  spin_vector_magnitudes(iter) = Draw_Spin2d(Phi, Method, Geometry2D, Figure,
iter);
  filename_spin_vector = fullfile(output_dir,
sprintf('frame_spin_vector_%04d.png', iter));
  saveas(gcf, filename_spin_vector);
  close(gcf);
  % Update the initial condition for the next iteration
  Phi_0{1} = Phi{1};
end
% Store the solutions in Outputs structure
Outputs.Solution = Solution_storage;
Outputs.Iterations = Evo;
% Create the GIFs from the saved images
create_gif(output_dir, gif_filename_magnitude, 'magnitude');
create_gif(output_dir, gif_filename_angle, 'angle');
create_gif(output_dir, gif_filename_velocity_x, 'velocity_x');
create_gif(output_dir, gif_filename_velocity_y, 'velocity_y');
create_gif(output_dir, gif_filename_gradient_magnitude, 'gradient_magnitude');
create_gif(output_dir, gif_filename_spin_vector, 'spin_vector');
% Use MakeVideo2d to create the video for the gradient magnitude
MakeVideo2d(Method, Geometry2D, Outputs, 'Function', @(phi, X, Y)
sqrt(abs(gradient(phi, Geometry2D.dx, Geometry2D.dy)).^2), 'VideoName',
video_filename_gradient_magnitude, 'Figure', setfield(Figure, 'title',
'Gradient Magnitude'));
% Use MakeVideo2d to create the video for the magnitude
MakeVideo2d(Method, Geometry2D, Outputs, 'Function', @(phi, X, Y) abs(phi).^2,
'VideoName', video_filename, 'Figure', setfield(Figure, 'title', 'Wave Function
|Phi|^2'));
% Create a video for the gradient magnitude as a surface plot
MakeVideo3d(Method, Geometry2D, Outputs, 'Function', @(phi, X, Y)
sqrt(abs(gradient(phi, Geometry2D.dx, Geometry2D.dy)).^2), 'VideoName',
video_filename_gradient_magnitude_surf, 'Figure', setfield(Figure, 'title',
'Gradient Magnitude (3D)'));
% Retrieve outputs
Energy = double(cell2mat(Outputs.Energy));
Angular_Momentum = double(cell2mat(Outputs.Angular_momentum));
xrms = double(cell2mat(Outputs.y_rms));
yrms = double(cell2mat(Outputs.x_rms));
grad = double(cell2mat(Outputs.User_defined_local(1)));
vort = double(cell2mat(Outputs.User_defined_local(2)));
% Get the number of iterations
num_iterations = length(Energy);
% Open the file for writing
% Save the results to a text file
fileID = fopen(results_filename, 'w');
% Write the headers
fprintf(fileID,
'Iteration\tEnergy\tAngular_Momentum\txrms\tyrms\tgrad\tvort\n');
% Write the data for each iteration in rows
```

```
for iter = 1:num_iterations
  fprintf(fileID, '%d\t%f\t%f\t%f\t%f\t%f\t%f\n', iter, Energy(iter),
Angular_Momentum(iter), xrms(iter), yrms(iter), grad(iter), vort(iter));
end
% Close the file
fclose(fileID);
%results_filename = fullfile(output_dir,
sprintf('longernewresults_width%.2f_decay%.2f_ellipticity%.2fnumCenters%.2fradi
us%.2f.txtellispeangle%.2f.txt', gaussian_width, decay_time, ellipticity,
numCenters, radius, ellipse_angle));
% Write the headers
% Plot the energy vs. iteration using scatter plot
figure;
% Plot Energy separately
subplot(4,2,1);
plot(1:num_iterations, Energy, 'b', 'LineWidth', 2); % Line plot for Energy
xlabel('Iteration');
ylabel('Energy');
title('Energy vs. Iteration');
grid on; % Add grid line
% Plot Angular Momentum
subplot(4,2,3);
plot(1:num_iterations, Angular_Momentum, 'r', 'LineWidth', 2); % Plot Angular
Momentum with lines
xlabel('Iteration');
ylabel('Angular Momentum');
title('Angular Momentum vs. Iteration');
grid on; % Add grid lines
% Plot X-rms, Y-rms, Gradient Norm, and Vorticity Norm combined
subplot(4,2,5);
hold on; % Enable hold to overlay plots
plot(1:num_iterations, xrms, 'g', 'LineWidth', 2); % Plot xrms with lines
plot(1:num_iterations, yrms, 'm', 'LineWidth', 2); % Plot yrms with lines
plot(1:num_iterations, grad, 'y', 'LineWidth', 2); % Plot grad norm with lines
plot(1*num_iterations, vort, 'c', 'LineWidth', 2); % Plot vorticity norm with
lines
xlabel('Iteration');
ylabel('Values');
title('X-rms, Y-rms, Grad Norm, and Vorticity Norm vs. Iteration');
legend('X-rms', 'Y-rms', 'Grad Norm', 'Vorticity Norm', 'Location', 'best'); %
Add legend
grid on; % Add grid lines
hold off; % Disable hold to prevent further overlays
% Plot X-rms vs Y-rms separately
subplot(4,2,7);
scatter(xrms, yrms, 50, 'm', 'filled', 'MarkerEdgeColor', 'k'); % Increase
marker size, set color, and edge color
xlabel('X-rms');
ylabel('Y-rms');
```

```
title('X-rms vs Y-rms');
grid on; % Add grid lines
% Plot the average magnitude of the spin vectors
subplot(4,2,8);
plot(1:Evo, spin_vector_magnitudes, 'LineWidth', 2);
xlabel('Iteration');
ylabel('Average Magnitude of Spin Vectors');
title('Average Magnitude of Spin Vectors vs. Iteration');
grid on;
% Plot the max x-directional velocity
subplot(4,2,2);
plot(1:Evo, max_velocity_x, 'LineWidth', 2);
xlabel('Iteration');
ylabel('Max X-Directional Velocity');
title('Max X-Directional Velocity vs. Iteration');
grid on;
% Plot the max y-directional velocity
subplot(4,2,4);
plot(1:Evo, max_velocity_y, 'LineWidth', 2);
xlabel('Iteration');
ylabel('Max Y-Directional Velocity');
title('Max Y-Directional Velocity vs. Iteration');
grid on;
% Plot the max gradient magnitude
subplot(4,2,6);
plot(1:Evo, max_gradient_magnitude, 'LineWidth', 2);
xlabel('Iteration');
ylabel('Max Gradient Magnitude');
title('Max Gradient Magnitude vs. Iteration');
grid on;
% Adjust subplot spacing
sgtitle('Simulation Outputs'); % Add a centered title for the entire figure
set(gcf, 'Position', [100, 100, 1000, 800]); % Adjust figure size
% Save the subplot figure as a PDF
%subplot_pdf_filename = fullfile(output_dir, 'simulation_outputs.pdf');
%saveas(gcf, subplot_pdf_filename);
% Create a simple text-based output
fileID = fopen(text_filename, 'w');
fprintf(fileID, '**Simulation Outputs**\n\n');
% Add references to the GIFs and videos
fprintf(fileID, 'Magnitude GIF: %s\n', gif_filename_magnitude);
fprintf(fileID, 'Angle GIF: %s\n', gif_filename_angle);
fprintf(fileID, 'X-Directional Velocity GIF: %s\n', gif_filename_velocity_x);
fprintf(fileID, 'Y-Directional Velocity GIF: %s\n', gif_filename_velocity_y);
fprintf(fileID, 'Gradient Magnitude GIF: %s\n',
gif_filename_gradient_magnitude);
fprintf(fileID, 'Spin Vector GIF: %s\n', gif_filename_spin_vector);
fprintf(fileID, 'Gradient Magnitude Video: %s\n',
video_filename_gradient_magnitude);
```

```
fprintf(fileID, 'Wave Function Magnitude Video: %s\n', video_filename);
fprintf(fileID, 'Gradient Magnitude (3D) Video: %s\n\n',
video_filename_gradient_magnitude_surf);
% Add a reference to the saved subplot figure
%fprintf(fileID, 'Simulation Outputs Figure: %s\n\n', subplot_pdf_filename);
% Add the simulation code to the last part of the text file
fprintf(fileID, '**Simulation Code:**\n\n');
code_text = fileread(code_filename);
fprintf(fileID, '%s', code_text);
fclose(fileID);
%% Functions
function draw_function_2d(phi, Geometry2D, Figure, iter, type, colormap_type)
  h = figure('Visible', 'off'); % Create figure in the background
  pcolor(Geometry2D.X, Geometry2D.Y, phi); % Drawing function
  axis equal
  axis tight
  shading interp; % Setting shading
  colormap(colormap_type); % Setting colormap
  colorbar; % Setting colorbar
  view(2); % Setting view
  xlabel(Figure.x); % Setting x-axis label
  ylabel(Figure.y); % Setting y-axis label
  if strcmp(type, 'Magnitude')
      title([Figure.title, num2str(iter)]); % Setting title with iteration
number for magnitude
  elseif strcmp(type, 'Angle')
      title(['Angle of Phi - Iteration ', num2str(iter)]); % Setting title with
iteration number for angle
  elseif strcmp(type, 'Velocity X')
      title(['X-Directional Velocity - Iteration ', num2str(iter)]); % Setting
title with iteration number for x-directional velocity
  elseif strcmp(type, 'Velocity Y')
      title(['Y-Directional Velocity - Iteration ', num2str(iter)]); % Setting
title with iteration number for y-directional velocity
  else
      title(['Gradient Magnitude - Iteration ', num2str(iter)]); % Setting
title with iteration number for gradient magnitude
  end
  if isvector(Figure.axis)
      caxis(Figure.axis);
  end
  drawnow; % Drawing
end
function create_gif(output_dir, gif_filename, type)
  % Get list of all PNG files in the directory
  if strcmp(type, 'magnitude')
      image_files = dir(fullfile(output_dir, 'frame_magnitude_*.png'));
  elseif strcmp(type, 'angle')
      image_files = dir(fullfile(output_dir, 'frame_angle_*.png'));
```

```
  elseif strcmp(type, 'velocity_x')
      image_files = dir(fullfile(output_dir, 'frame_velocity_x_*.png'));
  elseif strcmp(type, 'velocity_y')
      image_files = dir(fullfile(output_dir, 'frame_velocity_y_*.png'));
  elseif strcmp(type, 'spin_vector')
      image_files = dir(fullfile(output_dir, 'frame_spin_vector_*.png'));
  else
      image_files = dir(fullfile(output_dir,
'frame_gradient_magnitude_*.png'));
  end
  num_images = length(image_files);
  if num_images == 0
      error('No images found in the directory.');
  end
  % Read the first image to get dimensions
  [A, map] = rgb2ind(imread(fullfile(output_dir, image_files(1).name)), 256);
  % Initialize the GIF
  imwrite(A, map, gif_filename, 'gif', 'LoopCount', Inf, 'DelayTime', 0.05); %
Adjust DelayTime to make GIF shorter
  % Append the rest of the images to the GIF
  for k = 2:num_images
      [A, map] = rgb2ind(imread(fullfile(output_dir, image_files(k).name)),
256);
      imwrite(A, map, gif_filename, 'gif', 'WriteMode', 'append', 'DelayTime',
0.05); % Adjust DelayTime to make GIF shorter
  end
end
function MakeVideo2d(Method, Geometry2D, Outputs, varargin)
  %% Setting default Function input
  Default_Function = @(phi, X, Y) abs(phi).^2;
  %% Analysis of inputs
  Analyse_Var = inputParser; % Creating the parser
  Analyse_Var.addOptional('Function', Default_Function); % Optional input
'Function'
  Analyse_Var.addOptional('VideoName', 'MyVideo', @(x) (ischar(x) +
iscell(x))); % Optional input 'Name'
  Analyse_Var.addOptional('Figure', Figure_Var2d, @(x) isstruct(x)); % Optional
input 'Figure' which must be a structure
  %% Parsing inputs
  % Parsing inputs
  Analyse_Var.parse(varargin{:}); % Analysing the inputs
  % Setting inputs
  Function = Analyse_Var.Results.Function; %Storing the 'Function' input
  Name = Analyse_Var.Results.VideoName; %Storing the 'Name' input
  Figure = Analyse_Var.Results.Figure; %Storing the 'Name' input
  % Setting figure and properties
  figure('Visible', 'off'); % Create figure in the background
  surf(Geometry2D.X, Geometry2D.Y, Function(Outputs.Solution{1}, Geometry2D.X,
Geometry2D.Y), 'EdgeColor', 'none')
```

```
  shading interp;
  colormap(Figure.map); % Setting colormap to 'jet'
  view(2)
  title(Figure.title); % Setting title
  drawnow;
  set(gcf, 'nextplot', 'replacechildren', 'Visible', 'off');
  % Create a VideoObject
  if iscell(Name)
      vidObj = VideoWriter(strcat(Name{1}, '.avi'));
  else
      vidObj = VideoWriter(Name);
  end
  vidObj.Quality = 100;
  vidObj.FrameRate = 12;
  open(vidObj);
  for m = 1:Outputs.Iterations
      surf(Geometry2D.X, Geometry2D.Y, Function(Outputs.Solution{m},
Geometry2D.X, Geometry2D.Y), 'EdgeColor', 'none')
      shading interp;
      colormap(Figure.map); % Setting colormap to 'jet'
      view(2)
      title(Figure.title); % Setting title
      writeVideo(vidObj, getframe(gcf));
  end
  close(vidObj);
end
function MakeVideo3d(Method, Geometry2D, Outputs, varargin)
  %% Setting default Function input
  Default_Function = @(phi, X, Y) abs(phi).^2;
  %% Analysis of inputs
  Analyse_Var = inputParser; % Creating the parser
  Analyse_Var.addOptional('Function', Default_Function); % Optional input
'Function'
  Analyse_Var.addOptional('VideoName', 'MyVideo', @(x) (ischar(x) +
iscell(x))); % Optional input 'Name'
  Analyse_Var.addOptional('Figure', Figure_Var2d, @(x) isstruct(x)); % Optional
input 'Figure' which must be a structure
  %% Parsing inputs
  % Parsing inputs
  Analyse_Var.parse(varargin{:}); % Analysing the inputs
  % Setting inputs
  Function = Analyse_Var.Results.Function; %Storing the 'Function' input
  Name = Analyse_Var.Results.VideoName; %Storing the 'Name' input
  Figure = Analyse_Var.Results.Figure; %Storing the 'Name' input
  % Setting figure and properties
  figure('Visible', 'off'); % Create figure in the background
  surf(Geometry2D.X, Geometry2D.Y, Function(Outputs.Solution{1}, Geometry2D.X,
Geometry2D.Y), 'EdgeColor', 'none')
  shading interp;
```

```
  colormap(Figure.map); % Setting colormap to 'jet'
  view(3) % 3D view
  title(Figure.title); % Setting title
  drawnow;
  set(gcf, 'nextplot', 'replacechildren', 'Visible', 'off');
  % Create a VideoObject
  if iscell(Name)
      vidObj = VideoWriter(strcat(Name{1}, '.avi'));
  else
      vidObj = VideoWriter(Name);
  end
  vidObj.Quality = 100;
  vidObj.FrameRate = 12;
  open(vidObj);
  for m = 1:Outputs.Iterations
      surf(Geometry2D.X, Geometry2D.Y, Function(Outputs.Solution{m},
Geometry2D.X, Geometry2D.Y), 'EdgeColor', 'none')
      shading interp;
      colormap(Figure.map); % Setting colormap to 'jet'
      view(3) % 3D view
      title(Figure.title); % Setting title
      writeVideo(vidObj, getframe(gcf));
  end
  close(vidObj);
end
function Grad_norm = Example_outputs(Geometry2D, phi, x, y, fftx, ffty)
  Grad_x = ifft2(1i * fftx .* fft2(phi));
  Grad_y = ifft2(1i * ffty .* fft2(phi));
  Grad_x_norm = sqrt((Geometry2D.dx * Geometry2D.dy) *
sum(sum(abs(Grad_x).^2)));
  Grad_y_norm = sqrt((Geometry2D.dx * Geometry2D.dy) *
sum(sum(abs(Grad_y).^2)));
  Grad_norm = Grad_x_norm + Grad_y_norm;
end
function Vorticity_norm = computeVorticityAtOrigin(Geometry2D, phi, x, y, fftx,
ffty)
  % Placeholder function for computing vorticity norm
  % Calculate gradients
  Grad_x = ifft2(1i * fftx .* fft2(phi));
  Grad_y = ifft2(1i * ffty .* fft2(phi));
  % Calculate vorticity
  Vorticity = Grad_y - Grad_x;
  % Compute norm at origin
  Vorticity_norm = abs(Vorticity(ceil(end/2), ceil(end/2)));
end
%% Draw spin vector of the wave functions Phi
%% INPUTS:
%%          Phi: Wave functions (cell array)
```

```
%%          Method: Structure containing variables concerning the method
(structure) (see Method_Var2d.m)
%%          Geometry2D: Structure containing variables concerning the geometry
of the problem in 2D (structure) (see Geometry2D_Var2d.m)
%%          Figure: Structure containing variables concerning the figures
(structure) (see Figure_Var2d.m)
%% FUNCTIONS USED:
%%          draw_function_2d: To draw the wave function's square modulus and
angle (line 19 and 23)
function avg_magnitude = Draw_Spin2d(Phi, Method, Geometry2D, Figure, iter)
%% Computing the spin vector
Sx = real(conj(Phi{1}).*Phi{2})./(sqrt(abs(Phi{1}).^2+abs(Phi{2}).^2)); % spin
vector x-component
Sy = imag(conj(Phi{1}).*Phi{2})./(sqrt(abs(Phi{1}).^2+abs(Phi{2}).^2)); % spin
vector y-component
Density = sqrt(abs(Phi{1}).^2 + abs(Phi{2}).^2);
%% Printing the spin vector in the x-y plane
Figure.label = 1; % Number of the figure
Figure.title = ['Spin vector in the x-y plane (with density contour) -
Iteration ', num2str(iter)]; % Storing title of the figure
quiver_function_2d(Density, Sx, Sy, Geometry2D, Figure); % Drawing the spin
vector
% Calculate the magnitude of the spin vector
magnitude = sqrt(Sx.^2 + Sy.^2);
avg_magnitude = mean(magnitude(:)); % Calculate the average magnitude
end
function quiver_function_2d(Density, Sx, Sy, Geometry2D, Figure)
  h = figure('Visible', 'off'); % Create figure in the background
  quiver(Geometry2D.X, Geometry2D.Y, Sx, Sy); % Drawing function
  hold on;
  contour(Geometry2D.X, Geometry2D.Y, Density); % Adding density contour
  hold off;
  axis equal
  axis tight
  shading interp; % Setting shading
  colormap(jet); % Setting colormap
  colorbar; % Setting colorbar
  view(2); % Setting view
  xlabel(Figure.x); % Setting x-axis label
  ylabel(Figure.y); % Setting y-axis label
  title(Figure.title); % Setting title
  drawnow; % Drawing
end
```

Example Code for Data Analysis

```
% Initialize the parameters
num_C_values = 10; % Number of centers values
R_values = [3.00, 4.00, 5.00, 6.00]; % Radius values
ellipticity_value = 1.0; % Fixed ellipticity value
% Initialize the matrix to store minimum angular momentum values
min_angular_momentum_matrix = NaN(num_C_values, length(R_values));
% Indices for C values (centers 0 to 9)
C_values = 0:9;
% Loop through each radius value
for k = 1:length(R_values)
   R_value = R_values(k);

   % Loop through each C value
   for j = 1:num_C_values
       % Construct the filename
       C_value = C_values(j);
       filename =
sprintf('timeonlydependentpotentiallongernewresults_width0.30_decay1.00_ellipti
city%.2fnumCenters%.2fradius%.2f.txtellispeangle0.00.txt', ellipticity_value,
C_value, R_value);

       % Debugging: Display the constructed filename
       disp(['Trying to open file: ', filename]);

       % Open and read the file
       fileID = fopen(filename, 'r');
       if fileID == -1
           warning('File %s not found', filename);
           min_angular_momentum_matrix(j, k) = NaN; % Assign NaN to indicate
missing data
           continue;
       end

       % Read the data from the file
       data = textscan(fileID, '%f%f%f%f%f%f%f', 'Delimiter', '\t',
'HeaderLines', 1);
       fclose(fileID);

       % Extract the angular momentum values
       angular_momentum = data{3}; % The third column is Angular_Momentum

       % Find the minimum angular momentum in the current file
       min_angular_momentum_in_file = min(angular_momentum);

       % Store the minimum angular momentum in the matrix
       min_angular_momentum_matrix(j, k) = min_angular_momentum_in_file;
   end
```

```
end
% Create a meshgrid for the surf plot
[C_grid, R_grid] = meshgrid(C_values, R_values);
% Transpose the matrix for correct plotting
min_angular_momentum_matrix = min_angular_momentum_matrix';
% Create the surf plot
figure;
surf(C_grid, R_grid, min_angular_momentum_matrix, 'EdgeColor', 'none');
xlabel('Number of Centers (0-9)', 'Color', 'w');
ylabel('Radius', 'Color', 'w');
zlabel('Min Angular Momentum', 'Color', 'w');
title('Surf Plot of Min Angular Momentum Values with Fixed Ellipticity (1)',
'Color', 'w');
colorbar; % Add a colorbar for reference
colormap parula; % Use the normal colormap for better visualization
set(gca, 'Color', 'k', 'XColor', 'w', 'YColor', 'w', 'ZColor', 'w'); % Set axis
background to black
% Initialize a figure for the 2D plots
figure;
hold on;
set(gca, 'Color', 'k'); % Set background to black
% Define line styles and markers for distinction
line_styles = {'-', '--', '-.', ':', '-', '--', '-.', ':', '-', '--'};
markers = {'o', 's', 'd', '^', 'v', '<', '>', 'p', 'h', 'x'};
colors = jet(length(R_values)); % Generate distinct colors for each line using
the jet colormap
% Fixed C value for centers (4)
C_value_fixed = 4;
% Loop through each radius value
for k = 1:length(R_values)
   R_value = R_values(k);

   % Construct the filename
   filename =
sprintf('timeonlydependentpotentiallongernewresults_width0.30_decay1.00_ellipti
city%.2fnumCenters%.2fradius%.2f.txtellispeangle0.00.txt', ellipticity_value,
C_value_fixed, R_value);

   % Debugging: Display the constructed filename
   disp(['Trying to open file: ', filename]);

   % Open and read the file
   fileID = fopen(filename, 'r');
   if fileID == -1
       warning('File %s not found', filename);
       continue;
   end

   % Read the data from the file
```

```
    data = textscan(fileID, '%f%f%f%f%f%f%f', 'Delimiter', '\t', 'HeaderLines',
1);
    fclose(fileID);

    % Extract the iteration, energy, and angular momentum values
    iterations = data{1}; % The first column is Iteration
    energy = data{2}; % The second column is Energy
    angular_momentum = data{3}; % The third column is Angular_Momentum

    % Check for non-empty energy data
    if isempty(energy)
        warning('No energy data found in %s', filename);
        continue;
    end

    % Plot the angular momentum through each iteration
    yyaxis left;
    plot(iterations, angular_momentum, [line_styles{k}, markers{k}], 'Color',
colors(k,:), 'DisplayName', sprintf('AM Radius = %.2f', R_value));
    ylabel('Angular Momentum', 'Color', 'w');

    % Plot the energy through each iteration
    yyaxis right;
    plot(iterations, energy, 'LineStyle', '--', 'Color', colors(k,:),
'DisplayName', sprintf('Energy Radius = %.2f', R_value));
    ylabel('Energy', 'Color', 'w');
end
% Finalize the plot
xlabel('Iteration', 'Color', 'w');
title('Angular Momentum and Energy vs Iteration with Centers Fixed at 4',
'Color', 'w');
legend show;
grid on;
set(gca, 'XColor', 'w', 'YColor', 'w'); % Set axis colors to white
set(gcf, 'Color', 'k'); % Set figure background to black
% Update legend text color to white
legend_handle = legend;
set(legend_handle, 'TextColor', 'w', 'EdgeColor', 'w', 'Color', 'k'); % Ensure
the legend text and edge are white
hold off;
```