ES_APPM 446 Kuramoto-Sivashinsky Equation

Northwestern University

Nicolas Guerra, Solomon Valore-Caplan
Fall of 2022

1 Motivation

Kuramoto-Sivashinsky Equation:

$$u_t + \Delta u + \Delta^2 u + \frac{1}{2}|\nabla u|^2 = 0$$

In the late 1970s, Yoshiki Kuramoto and Gregory Sivashinsky wanted to find a way to model diffusive–thermal instabilities in flame propagation[1][2]. Their work brought about the derivation of the Kuramoto-Sivashinsky equation which has applications not only in the initial interest of laminar flame fronts but also in pipe flow, chemical-reaction dynamics, and in general plasma dynamics[3]. One of the properties that make this equation interesting is the chaotic behavior that can be observed as the solution of the equation evolves through time[4].

2 Methods

Our initial attempt at numerically solving the 2D Kuramoto-Sivashinsky equation on the annulus was to use Strang splitting with a combination of implicit and explicit timesteppers. We planned to use an implicit timestepper (Crank–Nicolson) to solve the Laplacian and biharmonic terms and an explicit timestepper (Runge–Kutta) to solve the squared advection terms. However, we could not work out how to properly treat the biharmonic component implicitly since it involved a mixed spatial derivative.

Our problem was that the simulated solution was blowing up exponentially no matter how we tried to represent the mixed spatial derivative. We tried a fully explicit timestepper and were able to recover the expected behavior of the model.

Our final procedure, then, was to use a fully explicit timestepper. We chose to use a four–step Adams–Bashforth method for timestepping and used the second order centered-difference method for the spatial derivatives. We used periodic boundary conditions in both x and y because we found several other simulations that had done so, so we're able to compare our results to theirs.

We still experienced exponential growth in our simulation, which we remedied by renormalizing our solution to a maximum magnitude of 2 after every timestep. We found that when we renormalized to a maximum magnitude of 1 we did not recover the expected chaotic behavior. We were not sure why this was, but thought it might have something to do with needing values greater than one to unsettle the highly structured solution.

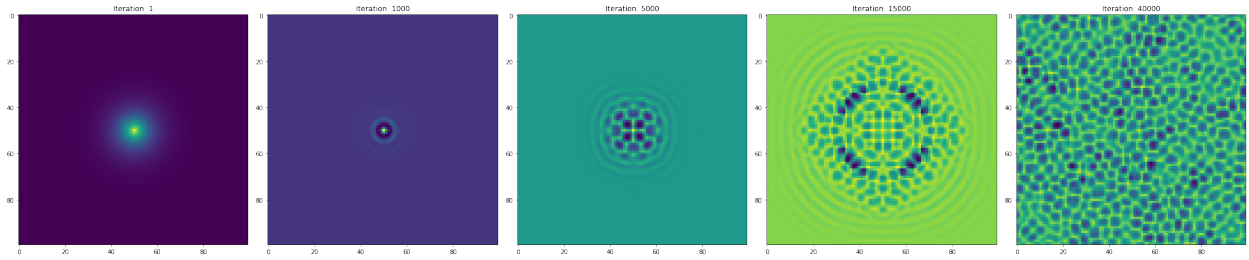
3 Verification of Algorithm

To confirm that our algorithm behaved as expected, we compared it with the solution provided by Mikael Mortensen[5]. To do so, we started with the same initial condition of $u(\mathbf{x}, 0) = exp(-0.1\mathbf{x} \cdot \mathbf{x})$. We also made sure to use the same domain size of $[-30\pi, 30\pi]^2$. Qualitatively, we can see that our solution matches Mortensen's solution. In the next section, there are some figures to aid the following descriptions. At first, the solution has only a circle which shrinks inward. Then, our solution has a rose-like pattern that grows more and more elaborate, until finally, the solution seems to have reached bubbly randomness. This behavior is expected since we knew this equation would lead us to chaotic activity to represent concepts such as flame propagation.

[1] https://www.sciencedirect.com/science/article/abs/pii/0094576577900960?via%3Dihub
[2] https://academic.oup.com/ptps/article/doi/10.1143/PTPS.64.346/1871466?login=false
[3] https://royalsocietypublishing.org/doi/10.1098/rspa.2014.0932
[4] https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.120.024102
[5] https://shenfun.readthedocs.io/en/latest/kuramatosivashinsky.html

4 Example Simulations

Using the initial condition and domain of the previous section, along with a resolution of $N = 100$ grid points and timestep of $dt = alpha * grid\_x.dx$ where $alpha = 0.001$, one gets the following solution.

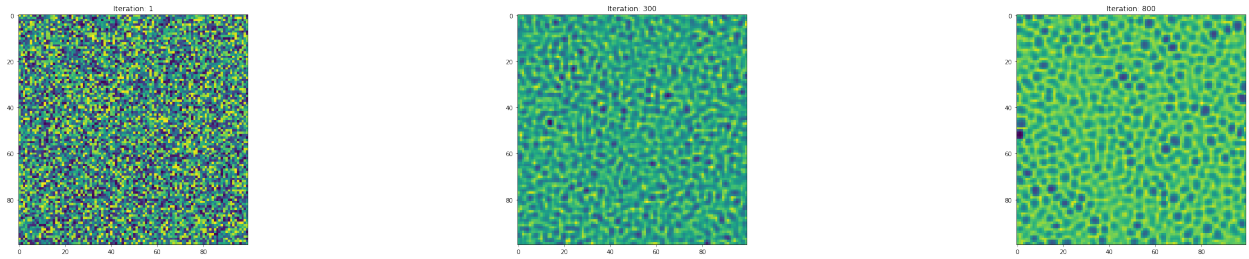


(a) Initial Condition (b) Circle (c) Rose (d) Nicer Rose (e) Random Bubbles

Figure 1: Evolution of Kuramoto-Sivashinsky Equation with Spot Initial Condition

If you would like to see a complete animation of this first solution, check out the YouTube video of this footnote[6].

In this next simulation, all parameters were kept the same except for the initial condition which was instead taken randomly from a uniform distribution.



(a) Initial Condition (b) Blurred (c) Converged

Figure 2: Evolution of Kuramoto-Sivashinsky Equation with Random Initial Condition

Although the initial conditions in both simulations were vastly different, the solution in both instances seems to converge to the image of bubbly randomness. The solution to which the simulations converge to depicts the phenomenon of flame propagation on a 2D field. To help visualize such a phenomenon, visualize a surface with flames all across it. The movement of these flames is modeled nicely with these simulations.

5 Code Documentation

To solve the Kuramoto-Sivashinsky Equation, we used an explicit scheme. This means we have to put the equation in the form: $\delta_t X = F(X)$. Below is the Python class to represent the equation in this form.

```
from timesteppers import StateVector

class ExplicitKS:

        def __init__(self, u, dx, dy, d2x, d2y):

            self.X = StateVector([u])
```

[6] https://youtube.com/shorts/ZPZGsiDnTNw?feature=share

```
8             def F(X):
9                 # Laplacian of u
10                lap_u = (d2x @ X.data) + (d2y @ X.data)
11                # Laplacian^2 of u
12                lapsq_u = (d2x @ lap_u) + (d2y @ lap_u)
13                # Full equation
14                return - lap_u - lapsq_u - 0.5*(dx @ X.data)**2 - 0.5*(dy @ X.data)**2
15            self.F = F
```

To actually go about solving the equation, use the following block of code. To change the resolution of the grid, increase N to have a higher resolution or decrease N to have a lower resolution. One can see in lines 8 and 9, the length of the X and Y grid is 60π. To change this, simply change the second argument of the functions. Notice that IC is the initial condition used in the first simulation while the commented IC is the initial condition used in the second simulation of the previous section. To switch initial conditions, simply comment the first IC and uncomment the second IC, or make up your own initial condition using x and y. The current desired spatial order of accuracy is 2 (for faster plotting), but this can simply be increased in line 34. The current timestep is $dt = alpha * grid\_x.dx$ where $alpha = 1e-2$. If you would like to increase the timestep, increase $alpha$ in line 38. If you would like to decrease the timestep, decrease $alpha$ in line 38. Currently, we are using the 4-step Adams-Bashforth method, however, if you would like to edit the number of steps, you can do so in line 40. In line 43, we can see that the number of iterations that will be performed is 50,000. We will plot the solution every 1,000 iterations as can be seen in line 46. Changing the plot every 1,000 iterations, showing it in line 50, and then clearing it in line 51 gives a nice animation of the solution.

```
1  import finite
2  from timesteppers import AdamsBashforth
3  import matplotlib.pyplot as plt
4  import numpy as np
5  from IPython.display import clear_output
6
7  N = 100 # Resolution
8  grid_x = finite.UniformPeriodicGrid(N, 60*np.pi)
9  grid_y = finite.UniformPeriodicGrid(N, 60*np.pi)
10 domain = finite.Domain((grid_x, grid_y))
11 x, y = domain.values()
12 xm, ym = domain.plotting_arrays()
13
14 # Initial Condition
15 r = np.sqrt((x-30*np.pi)**2 + (y-30*np.pi)**2)
16 IC = np.exp(-0.1*r)
17
18 #IC = np.random.uniform(size=(N,N))
19
20 u = np.zeros(domain.shape)
21 u[:] = IC
22
23 # Derivative Operators
24 spatial_order = 2
25
26 gridx = domain.grids[0]
27 gridy = domain.grids[1]
28
29 dx  = finite.DifferenceUniformGrid(1, spatial_order, gridx, 0)
30 dy  = finite.DifferenceUniformGrid(1, spatial_order, gridy, 1)
31 d2x = finite.DifferenceUniformGrid(2, spatial_order, gridx, 0)
32 d2y = finite.DifferenceUniformGrid(2, spatial_order, gridy, 1)
33
34 # Equation Set
35 kursiv = ExplicitKS(u, dx, dy, d2x, d2y)
36
37 # Timestepper
38 alpha = 1e-2
39 dt = alpha*grid_x.dx
```

```
ts = AdamsBashforth(kursiv, 4, dt)

# Plotting
for i in range(50000):
    ts.step(dt)
    u /= np.max(u)/2
    if ts.iter % 1000 == 0:
        plt.figure(figsize=(7,7))
        plt.imshow(u)
        plt.title('Iteration: '+str(ts.iter))
        plt.show()
        clear_output(wait=True)
```

To run and visualize the solution of the Kuramoto-Sivashinsky Equation, copy and paste the code in the previous coding blocks into one cell of a Jupyter Notebook making sure one has all the needed dependencies such as timesteppers, finite, matplotlib, numpy, and IPython. Finally, run the cell and observe the evolution of the solution.