Exercise_10

July 10, 2025

[2]:
```
# Install PhyFlow and Import libraries
%%capture
!pip install --quiet phiflow;
!pip install nbconvert;
!apt-get install texlive texlive-xetex texlive-latex-extra pandoc;
from google.colab import drive
drive.mount("/content/drive");
from phi.torch.flow import *
```

1 Advanced Deep Learning for Physics (IN2298)

2 Exercise 10

3 Kuramoto-Sivashinsky Simulator

3.1 (1) Solver implementation

[3]:
```
#function for the solver
def function(X: int,sign: int,alpha: float,x: tensor):
  return math.cos(3*math.pi*x / X) + sign*0.1*math.cos(2*math.pi*x/
↪X)*(1-alpha*math.sin(2*math.pi*x/X))

# implement solver
class KuramotoSivashinsky:

  def __init__(self,X: int, nx: int, dt: float, batch_size: int):
    '''
    Initialize solver Kuramoto Sivashinsky Equation

    parameters:
    X  : float
        periodic domain length [0,X]

    nx : int
        number of spatial grid points

    dt : float
```

```
          time step
        '''
        # Define spatial prameters
        self.X = X
        self.nx = nx
        self.dt = dt

        # Define PhiFlow shape
        self.shape = spatial(x=nx)
        self.batch_size = batch(b=batch_size)
        #create spatial grid
        self.x = math.linspace(0, X, self.shape)
        self.dx = self.x[1] - self.x[0]
        # Precompute wavenumbers for FFT
        k = math.fftfreq(self.shape, self.dx) * 2 * math.pi
        self.k = k
        self.k2 = k**2
        self.k4 = k**4
        self.L = self.k2 - self.k4
        # Initialize solution array:
        self.u = math.zeros_like(self.x)
        self.u_hat = math.zeros_like(self.x)

    def set_initial_condition(self,u: tensor):
        """
        Set the initial condition using a function of x.
        Parameters:
        func : callable
            Function u(x, 0)
        """
        self.u = u
        self.u_hat = math.fft(self.u)

    def NotLin(self,c_hat):
        # direct space for computing u2
        c = math.ifft(c_hat).real
        c2 = c**2
        # Fourier Space
        c2_hat = math.fft(c2)
        # Compute derivative in Fourier space:
        dudx_hat = -.5j * self.k * c2_hat
        return dudx_hat

    def step(self):
        '''
        Return the spatial grid and the solution with
        The exponential
```

```
        time-stepping Runge-Kutta of second order.
        '''
        L = self.L
        Nu = self.NotLin(self.u_hat)
        eldt = math.exp(L*self.dt)
        dt = self.dt
        tol = 1e-14
        phi1 = math.where(math.abs(L) < tol, dt, math.divide_no_nan((eldt - 1),L))
        phi2 = math.where(math.abs(L) < tol, .5*dt, math.divide_no_nan((eldt - 1␣
 ↪- L * dt),(L**2 * dt)))
        a = self.u_hat*eldt + Nu*phi1
        self.u_hat = a + (self.NotLin(a)-Nu)*phi2
        self.u = math.ifft(self.u_hat)

        return self.x,self.u.real
```

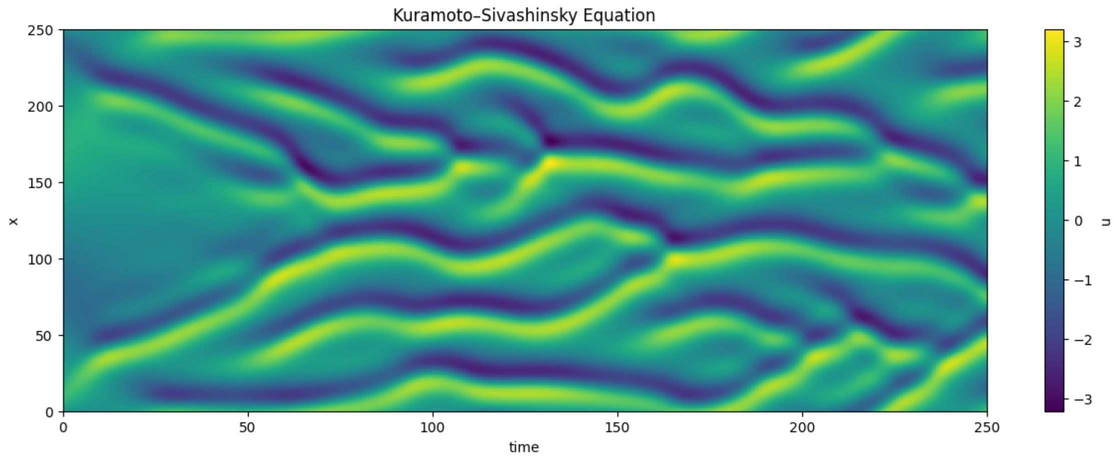
[4]:
```
import matplotlib.pyplot as plt
# setting variables
X = 50
nx = 250
dt = .5
alpha = 4
batch_size = 1
x = math.linspace(0, X,spatial(x=nx))
u = function(X,1,alpha,x)

# defining initial tensor
math.precision(64)
total_u = []
ks = KuramotoSivashinsky(X=X, nx=nx, dt=dt,batch_size=batch_size)
ks.set_initial_condition(u)
for i in range(250):
    _, u = ks.step()
    total_u.append(math.expand(u,spatial(time=1)))

u_final = math.concat(total_u,'time')
u_final = u_final.vector['x']
print(u_final)
import matplotlib.pyplot as plt
print(u_final)
u_numpy = u_final.numpy('x', 'time')  # shape (250, 250)
plt.figure(figsize=(15, 5))
plt.imshow(u_numpy, aspect='auto', origin='lower', extent=[0, 250, 0, 250])
plt.xlabel('time')
plt.ylabel('x')
plt.colorbar(label='u')
plt.title('Kuramoto-Sivashinsky Equation')
```

```
plt.show()
```

```
(x =250, time =250) 4.00e-04 ± 1.2e+00
(-3e+00...3e+00)
(x =250, time =250) 4.00e-04 ± 1.2e+00
(-3e+00...3e+00)
```



3.2 (2) Dataset generation

[4]:
```
# setting variables
X = 50
nx = 250
dt = .5
x = math.linspace(0, X,spatial(x=nx))

u = []
batch_size=6
for i in range(batch_size):
  sign = np.random.choice([-1,1])
  alpha = np.random.uniform(-8,8)
  y = function(X,sign,alpha,x)
  u.append(math.expand(y,batch(batch=1)))

u = math.concat(u,'batch')
print(u)

# defining initial tensor
math.precision(64)
total_u = []
ks = KuramotoSivashinsky(X=X, nx=nx, dt=dt,batch_size=batch_size)
ks.set_initial_condition(u)
```

```
for i in range(250):
    _, u = ks.step()
    total_u.append(math.expand(u,spatial(time=1)))

u_final = math.concat(total_u,'time')
u_final = u_final.vector['x']
print(u_final)
u_numpy = u_final.numpy('batch,x,time')
np.save('ex10_output_data.npy', u_numpy)
```

```
(batch =6, x =250) 1.33e-04 ± 7.2e-01
(-1e+00…1e+00)
(batch =6, x =250, time =250) 1.33e-04 ± 1.2e+00
(-3e+00…3e+00)
```

[1]:
```
#%%capture
!jupyter nbconvert --to pdf --output /content/drive/MyDrive/Fisica/ADL4P/
↪Exercise_10.pdf /content/drive/MyDrive/Fisica/ADL4P/Exercise_10.ipynb
```