Exercise 6

June 5, 2023

[ ]:
```
from phi.torch.flow import*
import matplotlib.pyplot as plt
```

- Numerical solver for Burger's equation

[ ]:
```
# implement the scheme
def burgers_forward(u, dx, dt):
    u_new = np.zeros_like(u)
    N = len(u_new)
    coeff = .5 * dt / dx #constant

    #first point periodic domain
    if u[-1] + [1] < 0:
        u_new[0] = u[0] - coeff * (u[1] ** 2 - u[0] ** 2)
    else:
        u_new[0] = u[0] - coeff * (u[0] ** 2 - u[-1] ** 2)

    #last points periodic domain
    if u[-2] + u[0] < 0:
        u_new[-1] = u[-1] - coeff * (u[0] ** 2 - u[-1] ** 2)
    else:
        u_new[-1] = u[-1] - coeff * (u[-1] ** 2 - u[-2] ** 2)

    for i in range(1, N - 1):
        nei_avg = u[i - 1] + u[i + 1] # local avg of nei's
        if nei_avg < 0 :
            u_new[i] = u[i] - coeff * ( u[i + 1] ** 2 - u[i] ** 2)
        else:
            u_new[i] = u[i] - coeff * ( u[i] ** 2 - u[i - 1] ** 2)
    return u_new
```

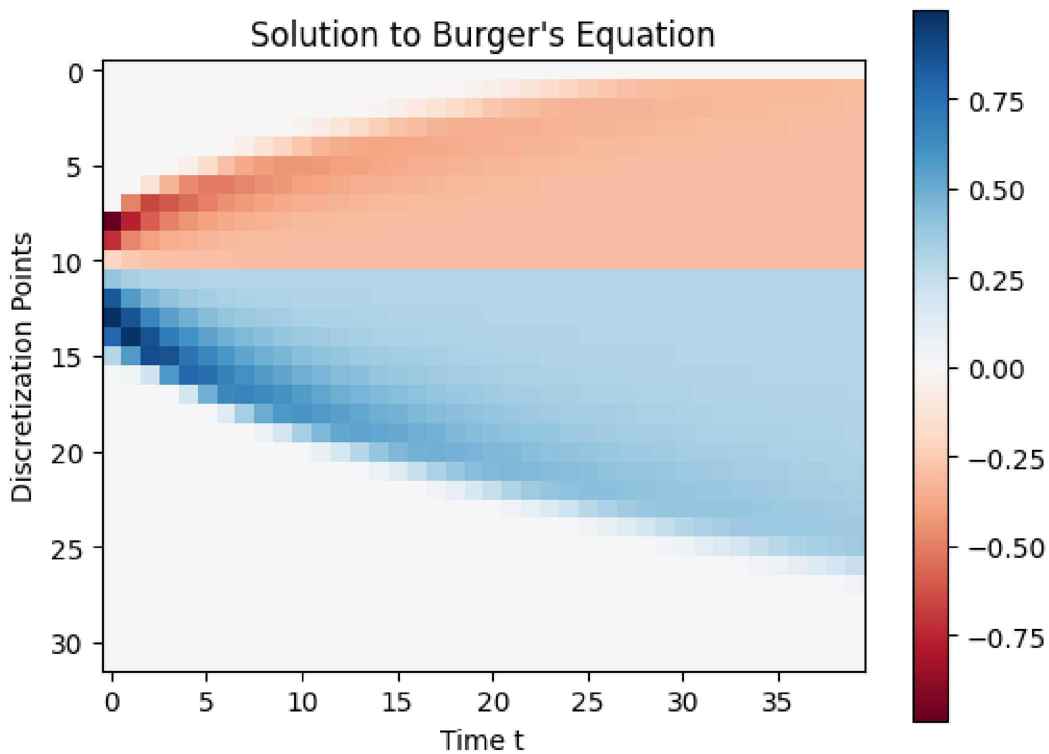
[ ]:
```
# domain lenght
Lx = 2 * PI
# discretisation points
N = 32
# total number of steps
num_steps = 40
# domain params
```

```
dx = Lx / (N -1)
dt = dx
sol = np.zeros(shape = (num_steps, N))
```

[ ]:
```
# locations
locs = np.linspace(start = 0, stop = 2 * PI, num = N)
# initial conditions
sol[0,:] = [math.sin(3 * loc) if loc < PI and loc > 0.5 * PI else 0 for loc in␣
 ↪locs]
```

[ ]:
```
for i in range(1, num_steps):
    sol[i] = burgers_forward(sol[i - 1], dx ,dt)
```

[ ]:
```
fig, ax = plt.subplots()
plt.title("Solution to Burger's Equation")
plt.xlabel("Time t")
plt.ylabel("Discretization Points")
plt.imshow(np.transpose(sol), cmap='RdBu')
plt.colorbar()
plt.show()
```



```
[ ]: def burgers_simulate(initial_state, dx, dt, num_steps = 15):
         internal_states = list()
         state = initial_state
         for i in range(num_steps - 1):
             internal_states.append(state)
             state = burgers_forward(state, dx, dt)
         return state, internal_states
```

- Backpropagation

```
[ ]: #we need  u_t/ u_(t-1)
     def burgers_backward(u, dx, dt):
         N = len(u)

         grad = np.zeros_like(u)
         coeff = .5 * dt / dx

         if u[-1] + [1] < 0:
             grad[0] = 1 + 2 * coeff * u[0]
         else:
             grad[0] = 1 - 2 * coeff * u[0]

         if u[-2] + u[0] < 0:
             grad[-1] = 1 + 2 * coeff * u[-1]
         else:
             grad[-1] = 1 - 2 * coeff * u[-1]


         for i in range(1, N-1):
             nei_avg = u[i - 1] + u[i + 1]
             if nei_avg < 0:
                 grad[i] = 1 + 2 * coeff *  u[i]

             else:
                 grad[i] = 1 - 2 * coeff * u[i]
         return grad
```

- Reconstructing inital conditions

```
[ ]: def loss_func(prediction, target):
         #MSE
         return np.square(target - prediction).mean()
     def grad_loss_func(preddiction, target):
         return 2 * (preddiction - target) / len(preddiction)
```

```
[ ]: def backward_iter(internal_states, dx, dt):
         num_steps, N = np.shape(internal_states)
         grad_total = np.ones(N)
```

```
    for i in reversed(range(num_steps)):
        u = internal_states[i]
        grad = burgers_backward(u,dx, dt) # u_t / u_{t - 1}
        grad_total *= grad # u_t / u_{t - 1} * u_{t - 1} / u_{t - 2} * ... *
 ↪u_1 / u_0
    return grad_total
```

[ ]:
```
Lx = 2 * PI
# discretisation points
N = 32
# total number of steps
num_steps = 40
# domain params
dx = Lx / (N -1)
dt = .1 * dx
num_steps = 15
```

[ ]:
```
time_15 = np.load("burgers_target_state.npy") # target
time_0 = np.sin(locs) # initial guess for t_0
init_vals = time_0
```

[ ]:
```
def backpropagation(u0, target, dx, dt, lr ,num_iters):
    for iter in range(num_iters):
        u_15, internal_states = burgers_simulate(u0, dx, dt)

        loss = loss_func(u_15, target)
        if iter % 100 == 0:
            print(f"Iteration: {iter}, Loss: {loss}")
        if iter == 250:
            lr *= .1
        grad_output = grad_loss_func(u_15, target) # L / u_15
        grad_internal_states = backward_iter(internal_states, dx, dt)
        grad = grad_internal_states * grad_output # L / u_0
        u0 -= lr * grad
    print(f"Final Loss: {loss}")
    return u0
```

[ ]:
```
lr = .1
num_iters = 1000
u0_optimized = backpropagation(time_0, time_15, dx, dt, lr, num_iters)
```

```
Iteration: 0, Loss: 2.192313929819946
Iteration: 100, Loss: 1.400337888317142
Iteration: 200, Loss: 0.7291237857538302
Iteration: 300, Loss: 0.4267813018077336
Iteration: 400, Loss: 0.36314646887201907
Iteration: 500, Loss: 0.3354598035218966
```

```
Iteration: 600, Loss: 0.3178775908349867
Iteration: 700, Loss: 0.30568081672955116
Iteration: 800, Loss: 0.2958684410499573
Iteration: 900, Loss: 0.28710665504696165
Final Loss: 0.2793248004493647
```

[ ]:
```python
time_15_optimized = burgers_forward(u0_optimized, dx, dt)
```

[ ]:
```python
plt.plot(locs, time_15, label = "u_15 ref")
plt.plot(locs, time_15_optimized, label = "u_15 optimized")
plt.grid()
plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x1665bfbd0>
```