Exercise 11 - KS Learning

In [1]:
```
import scipy as scp
from phi.torch.flow import *
import matplotlib.pyplot as plt
device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
device = torch.device(device)
```

Differentiable Solver

In [2]:
```
class DifferentiableKS():
    def __init__(self, resolution, domain_size, dt):
        self.resolution = resolution
        self.domain_size = domain_size
        self.dt = dt
        self.dx = domain_size/resolution

        # Matrices for exp. timestepping
        self.wavenumbers = math.fftfreq(math.spatial(x=resolution), self.dx).vector[0] * 1j
        self.L_mat = -self.wavenumbers**2-self.wavenumbers**4
        self.exp_lin = math.exp(self.L_mat * dt)
        self.nonlinear_coef_1 = math.divide_no_nan((self.exp_lin - 1) , self.L_mat)
        self.nonlinear_coef_1 = math.where(self.nonlinear_coef_1==0, self.dt, self.nonlinear_coef_1)
        self.nonlinear_coef_2 = math.divide_no_nan((self.exp_lin - 1 - self.L_mat*self.dt), (self.dt * self.L_mat**2))
        self.nonlinear_coef_2 = math.where(self.nonlinear_coef_2==0, self.dt/2, self.nonlinear_coef_2)

    def etrk2(self, u):
        nonlin_current = self.calc_nonlinear(u)
        u_interm = self.exp_lin * math.fft(u) + nonlin_current*self.nonlinear_coef_1
        u_new  = u_interm + (self.calc_nonlinear(math.real(math.ifft(tensor(u_interm,u.shape)))) +
                 - nonlin_current) * self.nonlinear_coef_2
        return math.real(math.ifft(tensor(u_new,u.shape)))

    def calc_nonlinear(self,u):
        return -0.5*self.wavenumbers*math.fft(u**2)

with math.precision(64):
    diff_ks = DifferentiableKS(resolution=50, domain_size=10, dt=1/2)
```

In [3]:
```
with math.precision(64):
    x = diff_ks.domain_size*math.tensor(np.arange(0,diff_ks.resolution),spatial('x'))/diff_ks.resolution
    u_dataset_init = [math.cos(2*x) +0.1*math.cos(2*math.PI*x/diff_ks.domain_size)*(1-2*math.sin(2*math.PI*x/diff_ks.domain_size)),
                      math.cos(2*x) +0.1*math.cos(2*math.PI*x/diff_ks.domain_size)*(1-1*math.sin(2*math.PI*x/diff_ks.domain_size)),
                      math.cos(2*x) -0.1*math.cos(2*math.PI*x/diff_ks.domain_size)*(1+2*math.sin(2*math.PI*x/diff_ks.domain_size)),
                      math.cos(2*x) +0.1*math.cos(2*math.PI*x/diff_ks.domain_size)*(1-3*math.sin(2*math.PI*x/diff_ks.domain_size)),
                      math.cos(2*x) +0.1*math.cos(2*math.PI*x/diff_ks.domain_size)*(1-8*math.sin(2*math.PI*x/diff_ks.domain_size)),
                      math.cos(2*x) -0.1*math.cos(2*math.PI*x/diff_ks.domain_size)*(1+4*math.sin(2*math.PI*x/diff_ks.domain_size)),]

    u_dataset_init = [math.expand(u, batch(b=1)) for u in u_dataset_init]
    u_dataset_init = math.concat(u_dataset_init, batch('b'))

    # Exponential timestepping with RK2
    u_traj = [u_dataset_init]
    u_iter = u_dataset_init
```

```python
    nonlin_iter = diff_ks.calc_nonlinear(u_dataset_init)
    for i in range(8000):
        u_iter = diff_ks.etrk2(u_iter)
        u_traj.append(u_iter)

u_traj = tensor(u_traj,instance('time') , u_iter.shape).numpy(['b','time', 'x']).astype(np.single)
np.savez('./dataset',u_traj)
```

Model

In [4]:
```python
class ConvResNet1D(torch.nn.Module):
    def __init__(self, flow_size, res_net_channels, res_net_depth, device):
        super(ConvResNet1D, self).__init__()

        self.upsample_conv_layer = torch.nn.Conv1d(1, res_net_channels, kernel_size=3, padding=1,
                                                   padding_mode='circular', bias=False).to(device)
        self.conv_layers = []
        self.relu_layers = []
        self.res_net_depth = res_net_depth
        for i in range(res_net_depth):
            self.conv_layers.append(torch.nn.Conv1d(res_net_channels, res_net_channels, kernel_size=3, padding=1,
                                                    padding_mode='circular', bias=False).to(device))
            self.relu_layers.append(torch.nn.LeakyReLU(inplace=True).to(device))
            self.conv_layers.append(torch.nn.Conv1d(res_net_channels, res_net_channels, kernel_size=3, padding=1,
                                                    padding_mode='circular', bias=False).to(device))

            self.relu_layers.append(torch.nn.LeakyReLU(inplace=True).to(device))

        self.downsample_conv_layer = torch.nn.Conv1d(res_net_channels, 1, kernel_size=3, padding=1,
                                                   padding_mode='circular', bias=False).to(device)
        self.net = torch.nn.Sequential(self.upsample_conv_layer, *self.conv_layers,
                                       *self.relu_layers, self.downsample_conv_layer)

    def forward(self,x):
        x = self.upsample_conv_layer(x)
        for i in range(self.res_net_depth):
            x_skip = x
            x = self.conv_layers[i*2](x)
            x = self.relu_layers[i*2](x)
            x = self.conv_layers[i*2+1](x)+x_skip
            x = self.relu_layers[i*2+1](x)
        x = self.downsample_conv_layer(x)
        return x
```

Dataset preparation

In [5]:
```python
# TRAINING DATASET - uses torch.utils.data to create iterable
# functions to arange dataset into training trajectories
def stack_prediction_inputs(data, prediction_horizon, input_slices):
    indices = np.arange(data.shape[0]-prediction_horizon - input_slices)[:,None] + np.arange(0,input_slices)
    return data[indices]

def stack_prediction_targets(data, prediction_horizon, input_slices):
    indices = np.arange(data.shape[0]-prediction_horizon - input_slices)[:,None] + np.arange(input_slices,input_slices+prediction_horizon)
    return data[indices]

dataset = np.load('./dataset.npz')['arr_0']
```

```
prediction_horizon = 1
dataset_entries = dataset.shape[0]
entry_size = dataset.shape[1]-prediction_horizon-1

torch_inputs = np.array([stack_prediction_inputs(d, prediction_horizon,1) for d in dataset])
print(torch_inputs.shape)
torch_inputs = torch.Tensor(torch_inputs.reshape(-1,1,torch_inputs.shape[-1]))

torch_outputs = np.array([stack_prediction_targets(d, prediction_horizon,1) for d in dataset])
print(torch_outputs.shape)
torch_outputs = torch.Tensor(torch_outputs.reshape(-1,prediction_horizon, torch_outputs.shape[-1]))

torch_dataset = torch.utils.data.TensorDataset(torch_inputs, torch_outputs)
torch_dataloader = torch.utils.data.DataLoader(torch_dataset, shuffle=True, batch_size=128)
```

```
(6, 7999, 1, 50)
(6, 7999, 1, 50)
```

Supervised Network Training (1-step)

In [6]:

```
# create network, check parameter count
cr_1step = ConvResNet1D(diff_ks.resolution, 16, 10, device)
print('Number of parameters: ',sum(p.numel() for p in cr_1step.parameters()))
```

```
Number of parameters:  15456
```

In [7]:

```
# PART 1: SUPERVISED TRAINING
optimizer = torch.optim.Adam(cr_1step.parameters(), lr=5e-5)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma = 0.9)
loss = torch.nn.MSELoss()

test_horizon = 100
for epoch in range(50):
    running_loss =0.0
    for i, data in enumerate(torch_dataloader, 0):
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)
            optimizer.zero_grad()

            nn_outputs = cr_1step(inputs)

            loss_value = loss(nn_outputs, labels)
            loss_value.backward()
            optimizer.step()

    # print loss every epoch
    running_loss += loss_value.item()
    print(f'[{epoch + 1}] loss: {running_loss :.6f}')
    running_loss = 0.0
    scheduler.step()
```

```
[1] loss: 0.007197
[2] loss: 0.003308
[3] loss: 0.002362
[4] loss: 0.002026
[5] loss: 0.001816
[6] loss: 0.001510
[7] loss: 0.001543
[8] loss: 0.001487
[9] loss: 0.001228
[10] loss: 0.001237
[11] loss: 0.001115
[12] loss: 0.001120
[13] loss: 0.001068
[14] loss: 0.001036
[15] loss: 0.001106
[16] loss: 0.001040
[17] loss: 0.000969
[18] loss: 0.001052
[19] loss: 0.000948
[20] loss: 0.000876
[21] loss: 0.000957
[22] loss: 0.000943
[23] loss: 0.000831
[24] loss: 0.000900
[25] loss: 0.000865
[26] loss: 0.000868
[27] loss: 0.000848
[28] loss: 0.000831
[29] loss: 0.000763
[30] loss: 0.000770
[31] loss: 0.000778
[32] loss: 0.000772
[33] loss: 0.000801
[34] loss: 0.000848
[35] loss: 0.000677
[36] loss: 0.000795
[37] loss: 0.000783
[38] loss: 0.000744
[39] loss: 0.000747
[40] loss: 0.000740
[41] loss: 0.000766
[42] loss: 0.000734
[43] loss: 0.000724
[44] loss: 0.000716
[45] loss: 0.000754
[46] loss: 0.000729
[47] loss: 0.000776
[48] loss: 0.000723
[49] loss: 0.000725
[50] loss: 0.000759
```

Supervised Network Training (3-step)

In [8]:

```
# create network, check parameter count
cr_3step = ConvResNet1D(diff_ks.resolution, 16, 10, device)
print('Number of parameters: ',sum(p.numel() for p in cr_3step.parameters()))
```

```
Number of parameters:  15456
```

In [9]:
```python
# PART 2: longer prediction horizon
dataset = np.load('./dataset.npz')['arr_0']
prediction_horizon = 3
dataset_entries = dataset.shape[0]
entry_size = dataset.shape[1]-prediction_horizon-1

torch_inputs = np.array([stack_prediction_inputs(d, prediction_horizon,1) for d in dataset])
print(torch_inputs.shape)
torch_inputs = torch.Tensor(torch_inputs.reshape(-1,1,torch_inputs.shape[-1]))

torch_outputs = np.array([stack_prediction_targets(d, prediction_horizon,1) for d in dataset])
print(torch_outputs.shape)
torch_outputs = torch.Tensor(torch_outputs.reshape(-1,prediction_horizon, torch_outputs.shape[-1]))

torch_dataset = torch.utils.data.TensorDataset(torch_inputs, torch_outputs)
torch_dataloader = torch.utils.data.DataLoader(torch_dataset, shuffle=True, batch_size=128)


# PART 2: SUPERVISED TRAINING
optimizer = torch.optim.Adam(cr_3step.parameters(), lr=5e-5)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma = 0.9)
loss = torch.nn.MSELoss()

test_horizon = 100
for epoch in range(50):
    running_loss =0.0
    for i, data in enumerate(torch_dataloader, 0):
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)
            optimizer.zero_grad()

            inputs = [inputs]
            nn_outputs = []
            for _ in range(prediction_horizon):
                nn_outputs.append(cr_3step(inputs[-1]))
                inputs.append(nn_outputs[-1].detach())        # added detach for supervised data-matching -> no differentiation through time

            nn_outputs = torch.concat(nn_outputs,axis=1)
            loss_value = loss(nn_outputs, labels)
            loss_value.backward()
            optimizer.step()

    # print loss every epoch
    running_loss += loss_value.item()
    print(f'[{epoch + 1}] loss: {running_loss :.6f}')
    running_loss = 0.0
    scheduler.step()
```

```
(6, 7997, 1, 50)
(6, 7997, 3, 50)
[1] loss: 0.020420
[2] loss: 0.010044
[3] loss: 0.007130
[4] loss: 0.005636
[5] loss: 0.005235
[6] loss: 0.004766
[7] loss: 0.004816
[8] loss: 0.004742
[9] loss: 0.004343
[10] loss: 0.004036
[11] loss: 0.003806
[12] loss: 0.003709
[13] loss: 0.004052
[14] loss: 0.003823
[15] loss: 0.003626
[16] loss: 0.003779
[17] loss: 0.003613
[18] loss: 0.003663
[19] loss: 0.003331
[20] loss: 0.003533
[21] loss: 0.003789
[22] loss: 0.003043
[23] loss: 0.003213
[24] loss: 0.003120
[25] loss: 0.003006
[26] loss: 0.003338
[27] loss: 0.002979
[28] loss: 0.003128
[29] loss: 0.003160
[30] loss: 0.003297
[31] loss: 0.002922
[32] loss: 0.002904
[33] loss: 0.003049
[34] loss: 0.003283
[35] loss: 0.002919
[36] loss: 0.003105
[37] loss: 0.002844
[38] loss: 0.003127
[39] loss: 0.003100
[40] loss: 0.003077
[41] loss: 0.002935
[42] loss: 0.003190
[43] loss: 0.003096
[44] loss: 0.002841
[45] loss: 0.003219
[46] loss: 0.002770
[47] loss: 0.002848
[48] loss: 0.003017
[49] loss: 0.002950
[50] loss: 0.002840
```

Differentiable Physics Network Training (3-step)

In [10]:

```
# create network, check parameter count
cr_phys = ConvResNet1D(diff_ks.resolution, 16, 10, device)
print('Number of parameters: ',sum(p.numel() for p in cr_3step.parameters()))
```

```
Number of parameters:  15456
```

In [11]:

```python
# PART 3: differentiable physics training
dataset = np.load('./dataset.npz')['arr_0']
prediction_horizon = 3
dataset_entries = dataset.shape[0]
entry_size = dataset.shape[1]-prediction_horizon-1

torch_inputs = np.array([stack_prediction_inputs(d, prediction_horizon,1) for d in dataset])
print(torch_inputs.shape)
torch_inputs = torch.Tensor(torch_inputs.reshape(-1,1,torch_inputs.shape[-1]))

torch_outputs = np.array([stack_prediction_targets(d, prediction_horizon,1) for d in dataset])
print(torch_outputs.shape)
torch_outputs = torch.Tensor(torch_outputs.reshape(-1,prediction_horizon, torch_outputs.shape[-1]))

torch_dataset = torch.utils.data.TensorDataset(torch_inputs, torch_outputs)
torch_dataloader = torch.utils.data.DataLoader(torch_dataset, shuffle=True, batch_size=128)


# PART 3: PHYSICS LOSS TRAINING
optimizer = torch.optim.Adam(cr_phys.parameters(), lr=5e-5)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma = 0.9)

def unsupervised_loss(prediction):
    prediction_input = prediction[:,:-1,:].reshape((-1,prediction.shape[-1]))
    prediction_input = tensor(prediction_input, instance('i'), spatial(x=prediction_input.shape[-1]))
    stepped_prediction = diff_ks.etrk2(prediction_input)
    loss = torch.mean((prediction[:,1:,:].reshape((-1,prediction.shape[-1]))-
                       stepped_prediction.native(['i','x']))**2/
                       (torch.mean(torch.abs(prediction[:,1:,:]))))
    return loss

test_horizon = 100
for epoch in range(50):
    running_loss =0.0
    for i, data in enumerate(torch_dataloader, 0):
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)
            optimizer.zero_grad()

            inputs = [inputs]
            nn_outputs = []
            for _ in range(prediction_horizon):
                nn_outputs.append(cr_phys(inputs[-1]))
                inputs.append(nn_outputs[-1].detach())        # added detach for supervised data-matching -> no differentiation through time

            nn_outputs = torch.concat(nn_outputs,axis=1)
            loss_value = unsupervised_loss(torch.concat([inputs[0],nn_outputs],axis=1))
            loss_value.backward()
            optimizer.step()

    # print loss every epoch
    running_loss += loss_value.item()
    print(f'[{epoch + 1}] loss: {running_loss :.6f}')
    running_loss = 0.0
    scheduler.step()
```

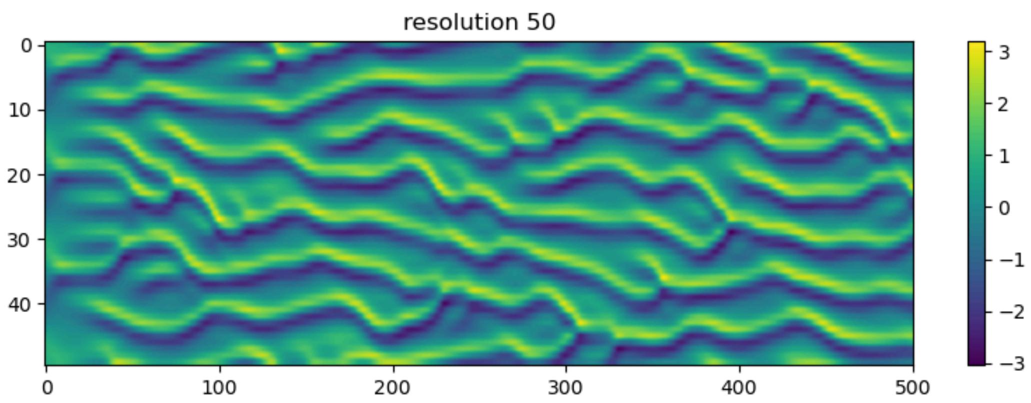
```
(6, 7997, 1, 50)
(6, 7997, 3, 50)
[1] loss: 0.026447
[2] loss: 0.017606
[3] loss: 0.012147
[4] loss: 0.009086
[5] loss: 0.006416
[6] loss: 0.005301
[7] loss: 0.004582
[8] loss: 0.003960
[9] loss: 0.003599
[10] loss: 0.003506
[11] loss: 0.003358
[12] loss: 0.003063
[13] loss: 0.002786
[14] loss: 0.002802
[15] loss: 0.002613
[16] loss: 0.002313
[17] loss: 0.002455
[18] loss: 0.002221
[19] loss: 0.002279
[20] loss: 0.002146
[21] loss: 0.002058
[22] loss: 0.001984
[23] loss: 0.001969
[24] loss: 0.001928
[25] loss: 0.001870
[26] loss: 0.001753
[27] loss: 0.001899
[28] loss: 0.001726
[29] loss: 0.001862
[30] loss: 0.001718
[31] loss: 0.001667
[32] loss: 0.001571
[33] loss: 0.001679
[34] loss: 0.001636
[35] loss: 0.001583
[36] loss: 0.001481
[37] loss: 0.001560
[38] loss: 0.001459
[39] loss: 0.001467
[40] loss: 0.001455
[41] loss: 0.001398
[42] loss: 0.001461
[43] loss: 0.001550
[44] loss: 0.001438
[45] loss: 0.001507
[46] loss: 0.001470
[47] loss: 0.001400
[48] loss: 0.001392
[49] loss: 0.001408
[50] loss: 0.001382
```

Evaluation

In [12]:

```
with math.precision(64):
    x = diff_ks.domain_size*math.tensor(np.arange(0,diff_ks.resolution),spatial('x'))/diff_ks.resolution
    u_test = [math.expand(math.cos(2*x), batch('b'))]
    u_iter = u_test[0]
```

```python
    nonlin_iter = diff_ks.calc_nonlinear(u_test[0])
    for i in range(500):
        u_iter = diff_ks.etrk2(u_iter)
        u_test.append(u_iter)

u_test = tensor(u_test,instance('time') , u_iter.shape).numpy(['b','time', 'x']).astype(np.single)
plt.figure(figsize=(10,3))
#ax = plt.axes([0, 1, 1, 1])
plt_data = np.real(u_test.transpose()[:,::1])
plt.imshow(plt_data, aspect='auto')
plt.colorbar()
plt.title("resolution "+str(diff_ks.resolution))
plt.show()
```



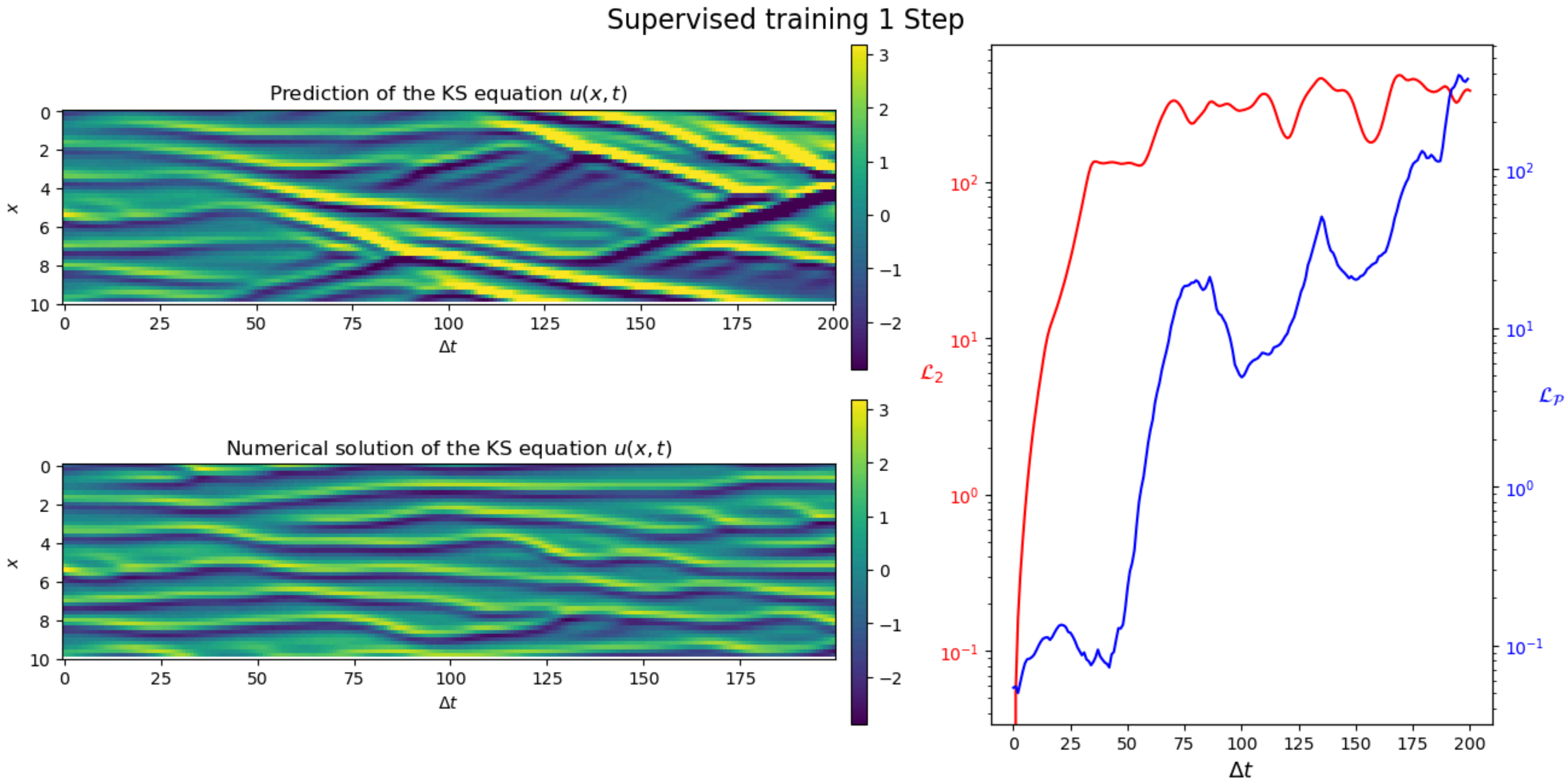
In [13]:

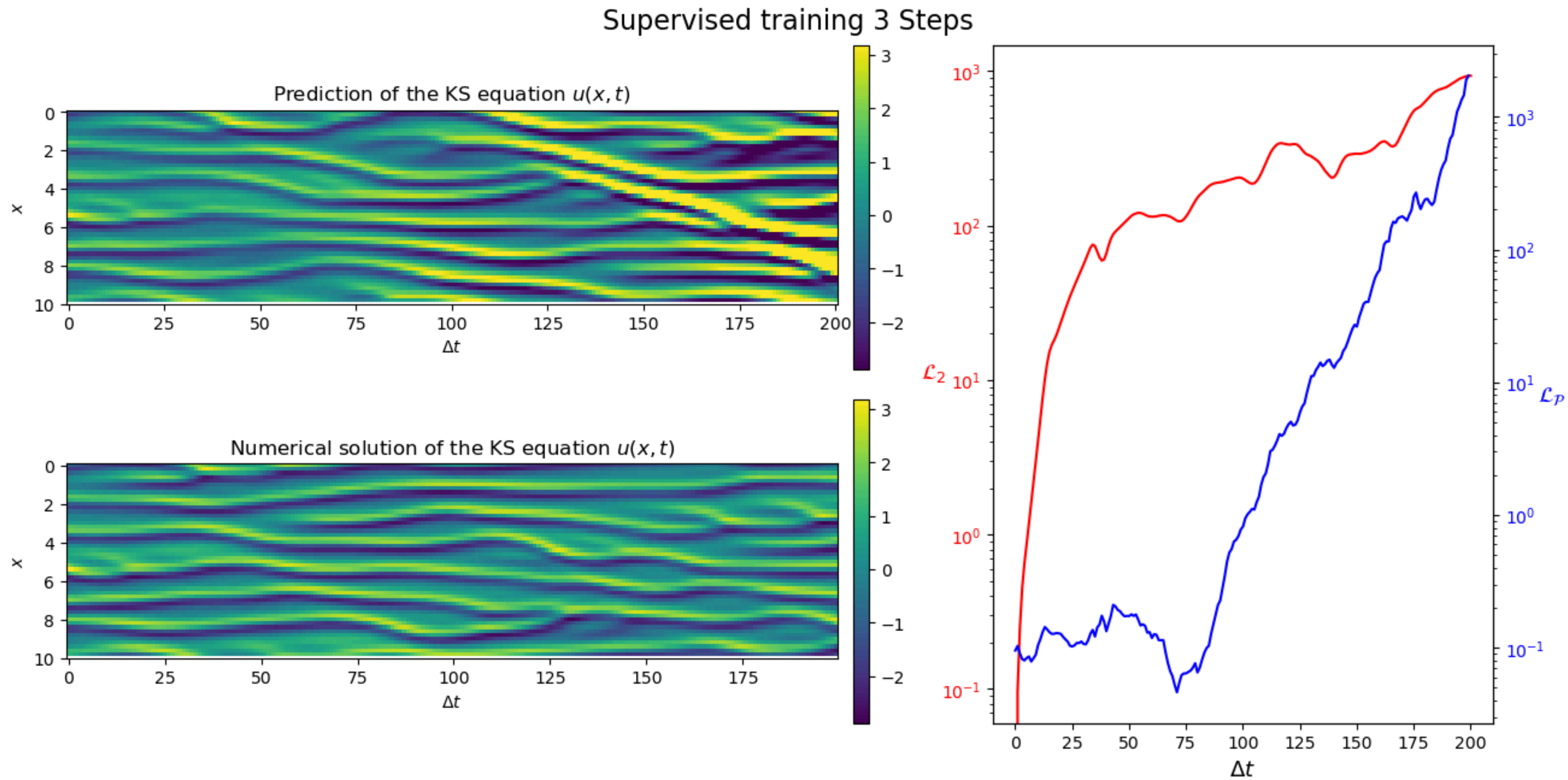
```python
import matplotlib.gridspec as gridspec

horizon = 200
plot_increment = 1

## 1 STEP SUPERVISED
test_frame_in = torch.tensor(u_test[0:1,100:101,:])
test_frame_out = [test_frame_in]
device_iter = test_frame_out[0].to(device)
for j in range(horizon):
    device_iter = cr_1step(device_iter)
    test_frame_out.append(device_iter.detach().cpu())

test_frame_out = torch.concat(test_frame_out, axis=0).numpy()

num_solution = u_test[0,100:100+horizon:plot_increment,:]
vmin, vmax = num_solution.min(), num_solution.max()

gs_kw = dict(width_ratios= [2, 1.3])
f, axes = plt.subplot_mosaic([['upper left', 'right'],
                              ['lower left', 'right']],
                              gridspec_kw=gs_kw, figsize=(13,6.5),
                              constrained_layout=True)
ax0 = axes['upper left']
im0 = ax0.imshow(np.transpose(test_frame_out[::plot_increment,0,:]), vmin=vmin, vmax=vmax)
ax0.set_yticks(np.arange(0,diff_ks.resolution+1,diff_ks.resolution/(diff_ks.domain_size/2)),np.arange(0,diff_ks.domain_size+1,2))
ax0.set_ylabel(r'$x$')
ax0.set_xlabel(r'$\Delta t$')
```

```python
ax0.set_title(r'Prediction of the KS equation $u(x,t)$')
plt.colorbar(im0, ax=ax0, pad=0.002)

#f.subplots(2,2,3,gridspec_kw={'width_ratios': [2, 1]})
ax1 = axes['lower left']
im1 = ax1.imshow(np.transpose(num_solution))
ax1.set_yticks(np.arange(0,diff_ks.resolution+1,diff_ks.resolution/(diff_ks.domain_size/2)),np.arange(0,diff_ks.domain_size+1,2))
ax1.set_ylabel(r'$x$')
ax1.set_xlabel(r'$\Delta t$')
ax1.set_title(r'Numerical solution of the KS equation $u(x,t)$')
plt.colorbar(im1, ax= ax1, pad =0.002)

prediction_input = test_frame_out[:-1,:]
prediction_input = tensor(prediction_input[:,0,:], instance('i'), spatial(x=prediction_input.shape[-1]))
stepped_prediction = diff_ks.etrk2(prediction_input).native(['i','x']).detach().cpu().numpy()

ax2 = axes['right']
ax2_2 = ax2.twinx()
ax2.plot(np.sum((test_frame_out[:,0,:]-u_test[0,100:101+horizon,:])**2, axis=-1), color='r')
ax2.tick_params(axis='y', labelcolor='r')
ax2.set_yscale('log')
ax2.set_ylabel(r'$\mathcal{L}_2$', rotation=0, color='r', fontsize = 13)
ax2.set_xlabel(r'$\Delta t$', fontsize = 13)
ax2_2.plot(np.sum((test_frame_out[1:,:][:,0,:]-stepped_prediction)**2, axis=-1), color='b')
ax2_2.tick_params(axis='y', labelcolor='b')
ax2_2.set_ylabel(r'$\mathcal{L}_\mathcal{P}$', rotation=0, color='b', fontsize = 13)
ax2_2.set_yscale('log')
f.suptitle('Supervised training 1 Step',fontsize=16)
plt.show()


## 3 STEP SUPERVISED
test_frame_in = torch.tensor(u_test[0:1,100:101,:])
test_frame_out = [test_frame_in]
device_iter = test_frame_out[0].to(device)
for j in range(horizon):
    device_iter = cr_3step(device_iter)
    test_frame_out.append(device_iter.detach().cpu())

test_frame_out = torch.concat(test_frame_out, axis=0).numpy()

gs_kw = dict(width_ratios= [2, 1.3])
f, axes = plt.subplot_mosaic([['upper left', 'right'],
                              ['lower left', 'right']],
                              gridspec_kw=gs_kw, figsize=(13,6.5),
                              constrained_layout=True)
ax0 = axes['upper left']
im0 = ax0.imshow(np.transpose(test_frame_out[::plot_increment,0,:]), vmin=vmin, vmax=vmax)
ax0.set_yticks(np.arange(0,diff_ks.resolution+1,diff_ks.resolution/(diff_ks.domain_size/2)),np.arange(0,diff_ks.domain_size+1,2))
ax0.set_ylabel(r'$x$')
ax0.set_xlabel(r'$\Delta t$')
ax0.set_title(r'Prediction of the KS equation $u(x,t)$')
plt.colorbar(im0, ax=ax0, pad=0.002)

#f.subplots(2,2,3,gridspec_kw={'width_ratios': [2, 1]})
ax1 = axes['lower left']
im1 = ax1.imshow(np.transpose(num_solution))
ax1.set_yticks(np.arange(0,diff_ks.resolution+1,diff_ks.resolution/(diff_ks.domain_size/2)),np.arange(0,diff_ks.domain_size+1,2))
ax1.set_ylabel(r'$x$')
ax1.set_xlabel(r'$\Delta t$')
```

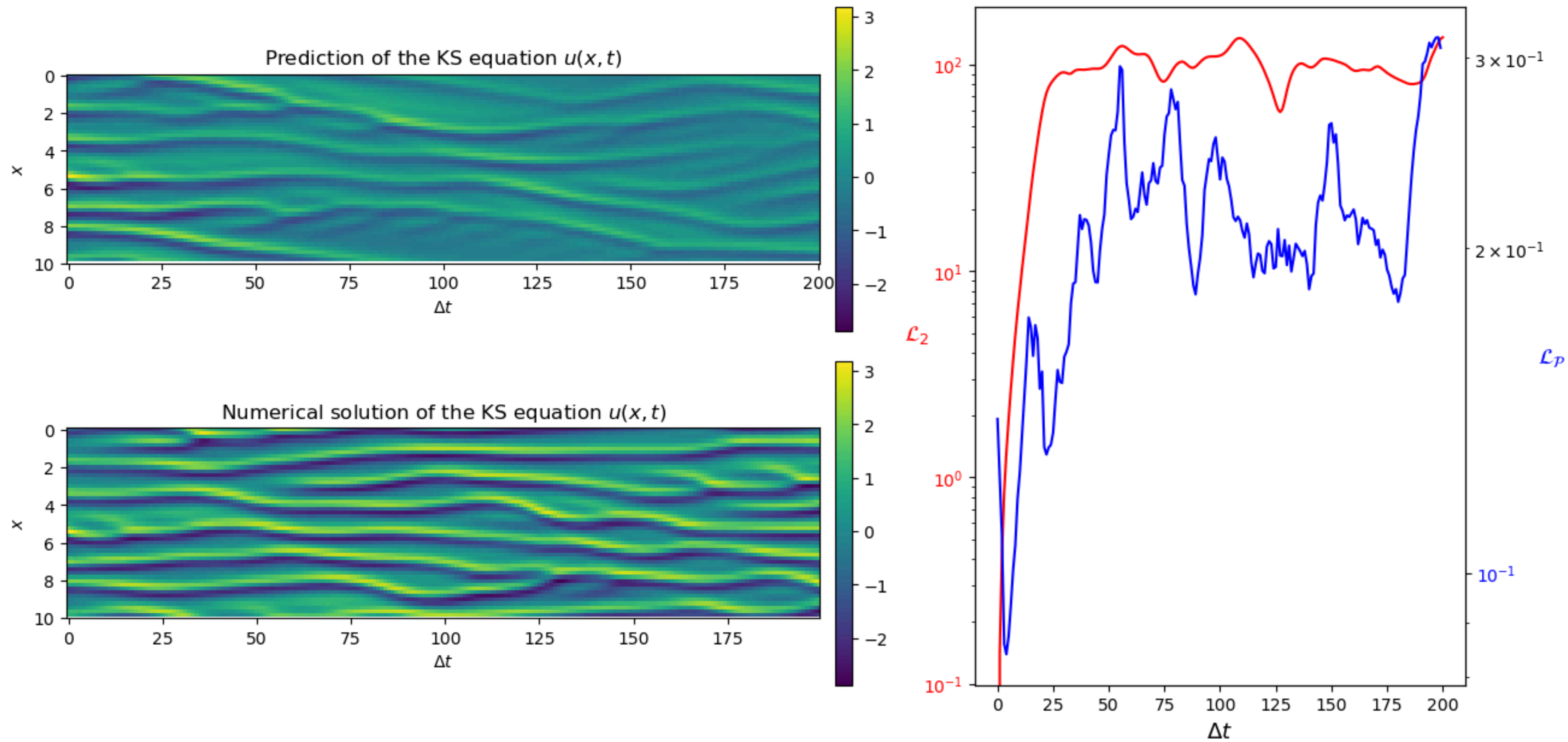
```python
ax1.set_title(r'Numerical solution of the KS equation $u(x,t)$')
plt.colorbar(im1, ax= ax1, pad =0.002)

prediction_input = test_frame_out[:-1,:]
prediction_input = tensor(prediction_input[:,0,:], instance('i'), spatial(x=prediction_input.shape[-1]))
stepped_prediction = diff_ks.etrk2(prediction_input).native(['i','x']).detach().cpu().numpy()

ax2 = axes['right']
ax2_2 = ax2.twinx()
ax2.plot(np.sum((test_frame_out[:,0,:]-u_test[0,100:101+horizon,:])**2, axis=-1), color='r')
ax2.tick_params(axis='y', labelcolor='r')
ax2.set_yscale('log')
ax2.set_ylabel(r'$\mathcal{L}_2$', rotation=0, color='r', fontsize = 13)
ax2.set_xlabel(r'$\Delta t$', fontsize = 13)
ax2_2.plot(np.sum((test_frame_out[1:,:][:,0,:]-stepped_prediction)**2, axis=-1), color='b')
ax2_2.tick_params(axis='y', labelcolor='b')
ax2_2.set_ylabel(r'$\mathcal{L}_\mathcal{P}$', rotation=0, color='b', fontsize = 13)
ax2_2.set_yscale('log')
f.suptitle('Supervised training 3 Steps',fontsize=16)
plt.show()


## 3 STEP PHYSICS LOSS
test_frame_in = torch.tensor(u_test[0:1,100:101,:])
test_frame_out = [test_frame_in]
device_iter = test_frame_out[0].to(device)
for j in range(horizon):
    device_iter = cr_phys(device_iter)
    test_frame_out.append(device_iter.detach().cpu())

test_frame_out = torch.concat(test_frame_out, axis=0).numpy()

gs_kw = dict(width_ratios= [2, 1.3])
f, axes = plt.subplot_mosaic([['upper left', 'right'],
                              ['lower left', 'right']],
                              gridspec_kw=gs_kw, figsize=(13, 6.5),
                              constrained_layout=True)
ax0 = axes['upper left']
im0 = ax0.imshow(np.transpose(test_frame_out[::plot_increment,0,:]), vmin=vmin, vmax=vmax)
ax0.set_yticks(np.arange(0,diff_ks.resolution+1,diff_ks.resolution/(diff_ks.domain_size/2)),np.arange(0,diff_ks.domain_size+1,2))
ax0.set_ylabel(r'$x$')
ax0.set_xlabel(r'$\Delta t$')
ax0.set_title(r'Prediction of the KS equation $u(x,t)$')
plt.colorbar(im0, ax=ax0, pad=0.002)

#f.subplots(2,2,3,gridspec_kw={'width_ratios': [2, 1]})
ax1 = axes['lower left']
im1 = ax1.imshow(np.transpose(num_solution))
ax1.set_yticks(np.arange(0,diff_ks.resolution+1,diff_ks.resolution/(diff_ks.domain_size/2)),np.arange(0,diff_ks.domain_size+1,2))
ax1.set_ylabel(r'$x$')
ax1.set_xlabel(r'$\Delta t$')
ax1.set_title(r'Numerical solution of the KS equation $u(x,t)$')
plt.colorbar(im1, ax= ax1, pad =0.002)

prediction_input = test_frame_out[:-1,:]
prediction_input = tensor(prediction_input[:,0,:], instance('i'), spatial(x=prediction_input.shape[-1]))
stepped_prediction = diff_ks.etrk2(prediction_input).native(['i','x']).detach().cpu().numpy()

ax2 = axes['right']
ax2_2 = ax2.twinx()
```

```python
ax2.plot(np.sum((test_frame_out[:,0,:]-u_test[0,100:101+horizon,:])**2, axis=-1), color='r')
ax2.tick_params(axis='y', labelcolor='r')
ax2.set_yscale('log')
ax2.set_ylabel(r'$\mathcal{L}_2$', rotation=0, color='r', fontsize = 13)
ax2.set_xlabel(r'$\Delta t$', fontsize = 13)
ax2_2.plot(np.sum((test_frame_out[1:,:][:,0,:]-stepped_prediction)**2, axis=-1), color='b')
ax2_2.tick_params(axis='y', labelcolor='b')
ax2_2.set_ylabel(r'$\mathcal{L}_\mathcal{P}$', rotation=0, color='b', fontsize = 13)
ax2_2.set_yscale('log')
f.suptitle('Physics loss training', fontsize=16)
plt.show()
```




Supervised training 3 Steps
Prediction of the KS equation $u(x, t)$
Numerical solution of the KS equation $u(x, t)$
x
Δt
$\mathcal{L}_2$
$\mathcal{L}_\mathcal{P}$

Physics loss training



In [ ]: