

Chapter 31-16: PDEs Solvers Using Neural Network Methods

PDE study considered from the viewpoint of neural networks. Many PDEs describe the evolution of a spatially distributed system over time. The state of such a system is defined by a value  $v(x, t)$  that depends on a spatial variable  $x$  that is usually a vector and on time  $t$ . The value itself can be a vector as well.

"Learning samples" can consist of discrete values of the initial condition  $v_{i,0}(x_{i,0})$  and values  $v_{ij}(x_i, t_j)$  at later times. The initial values are presented to the network as inputs. The outputs of the (recurrent) network at later times  $t_j$  are then compared with the sample values to calculate the error.

Since PDEs are often spatially homogeneous, it makes sense to use recurrent convolutional networks here. This is the same type that is often used in image processing.

The size of the convolutional kernel depends on the degree of spatial derivatives in the PDE:

The network needs to have at least one layer for each component of the value  $v$ . Additional hidden layers might be required, especially if the PDE has higher temporal derivatives.

The example in the cell below was taken from the Github repository, <MatthewFilipovich / neural-network-pde-solver> . Besides this 1D heat equation example, examples for 2D heat and 1D wave are also contained there. Neural network PDE solving is a very active area, with lots of participants, and new solvers are coming out at a fast pace.

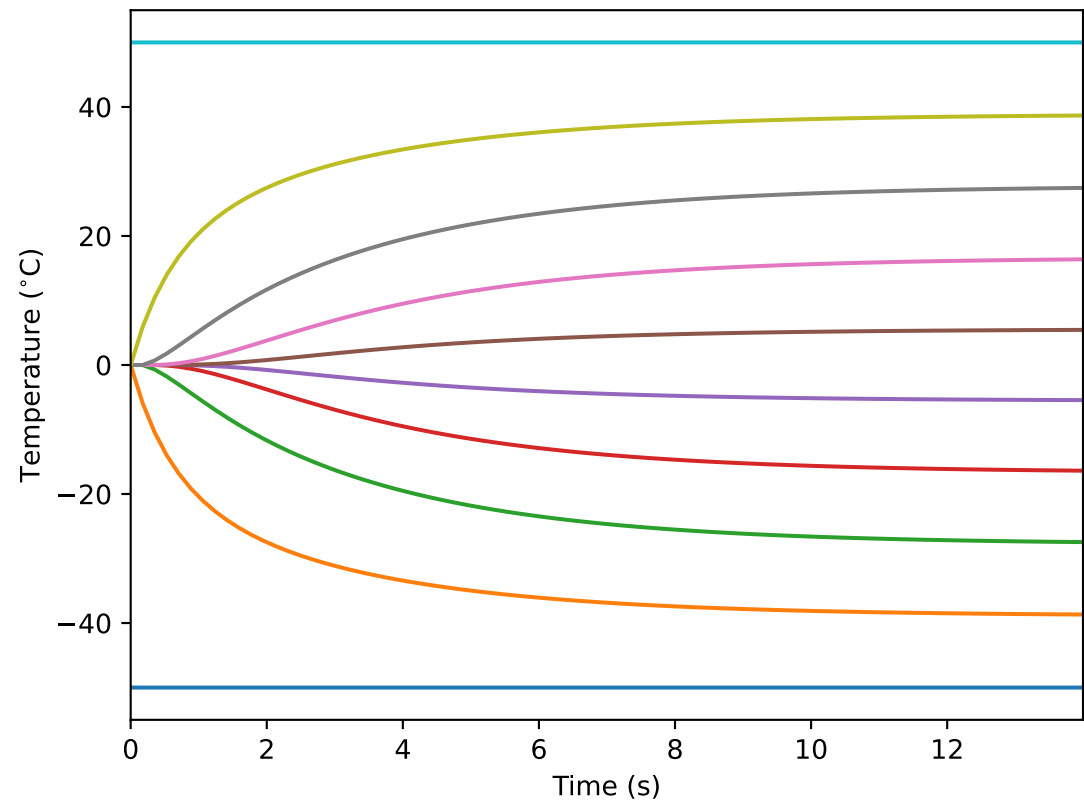
In this particular example, Mr. Filipovich has elected to show plots from new and older methods for comparison. (The 2nd plot shows a straight line, but if you look closely at the 5th plot, which is its companion, you will see that its line is not quite straight.)

The nengo\_pde module is not hosted on Pypi, and cannot be installed by pip. But due to the presence of the `_init_.py` file in the module folder, placing the folder in the Jupyter home directory allows the nengo\_pde module to be imported without difficulty into a Jupyter cell. (It is, however, necessary to install the nengo module from Pypi.)

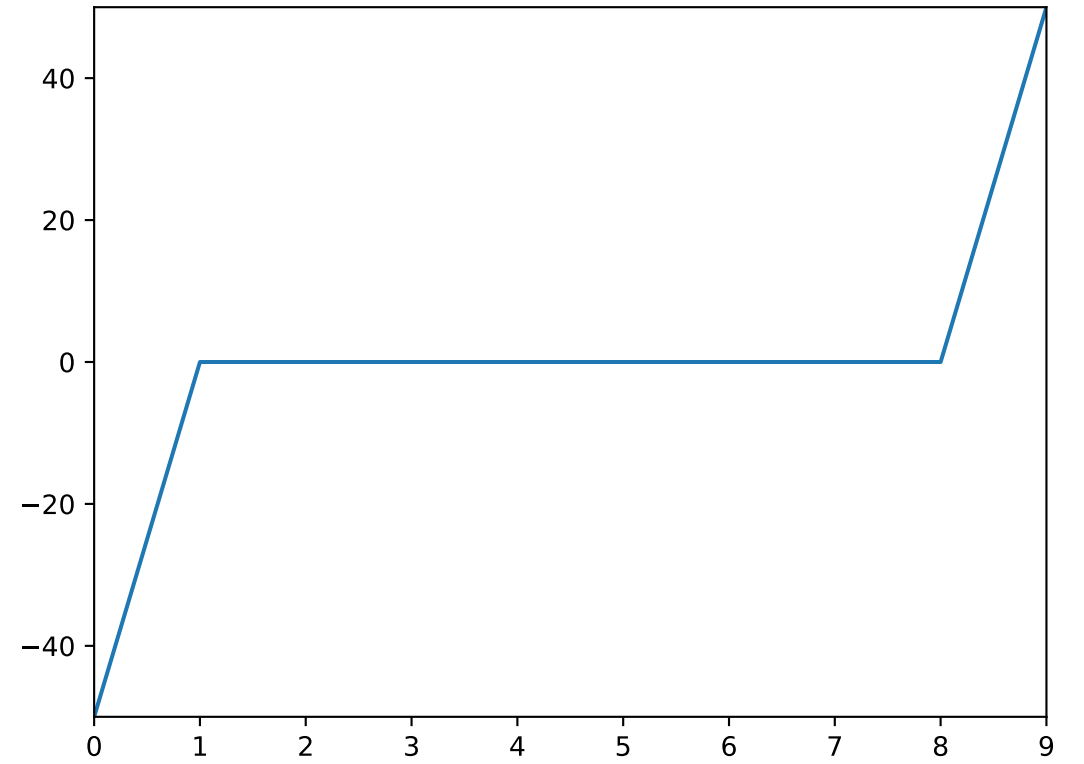
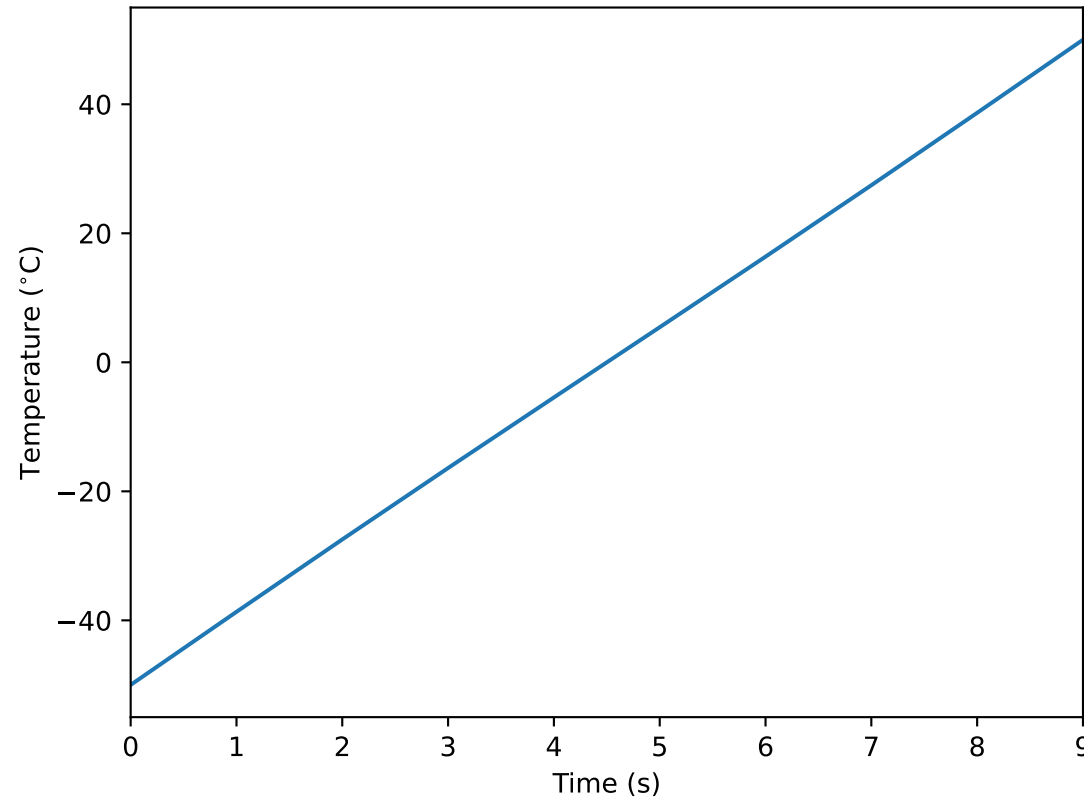
In [14]:

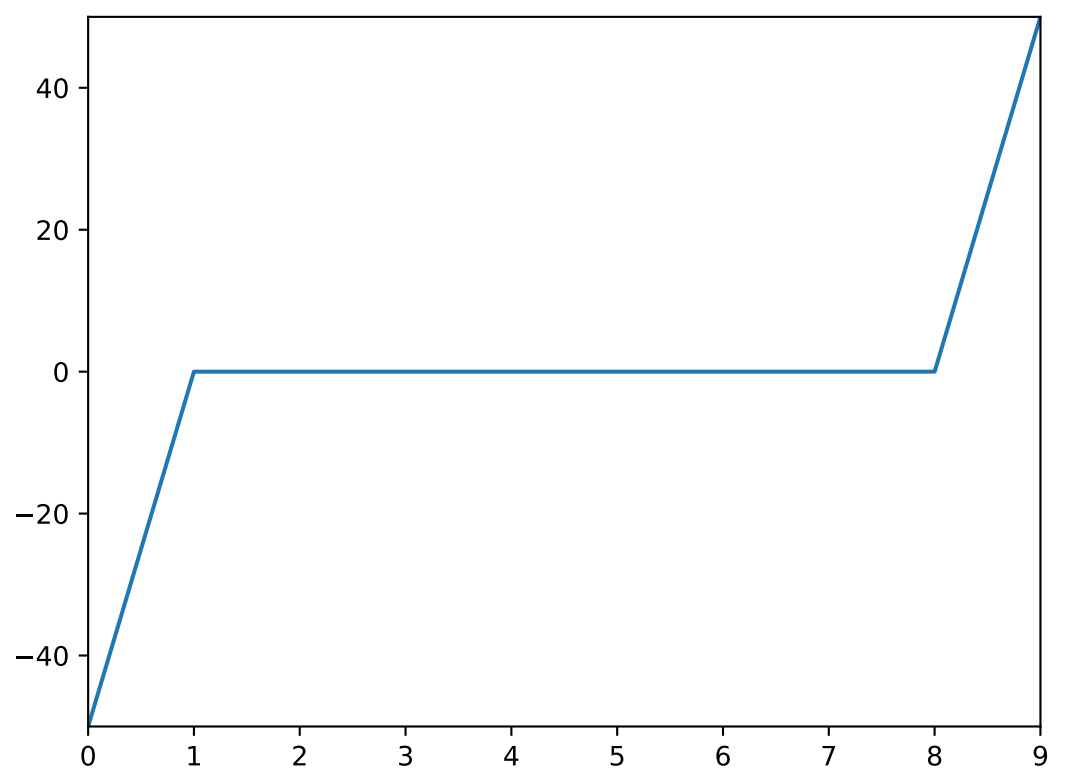
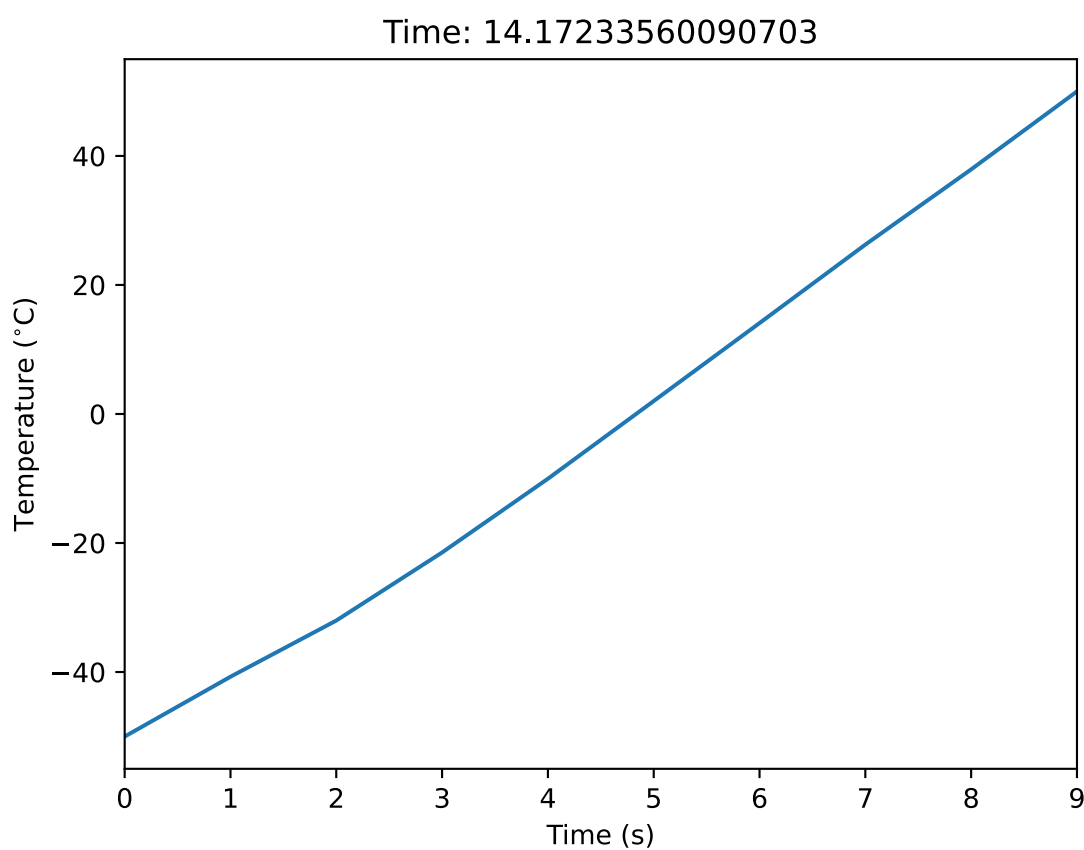
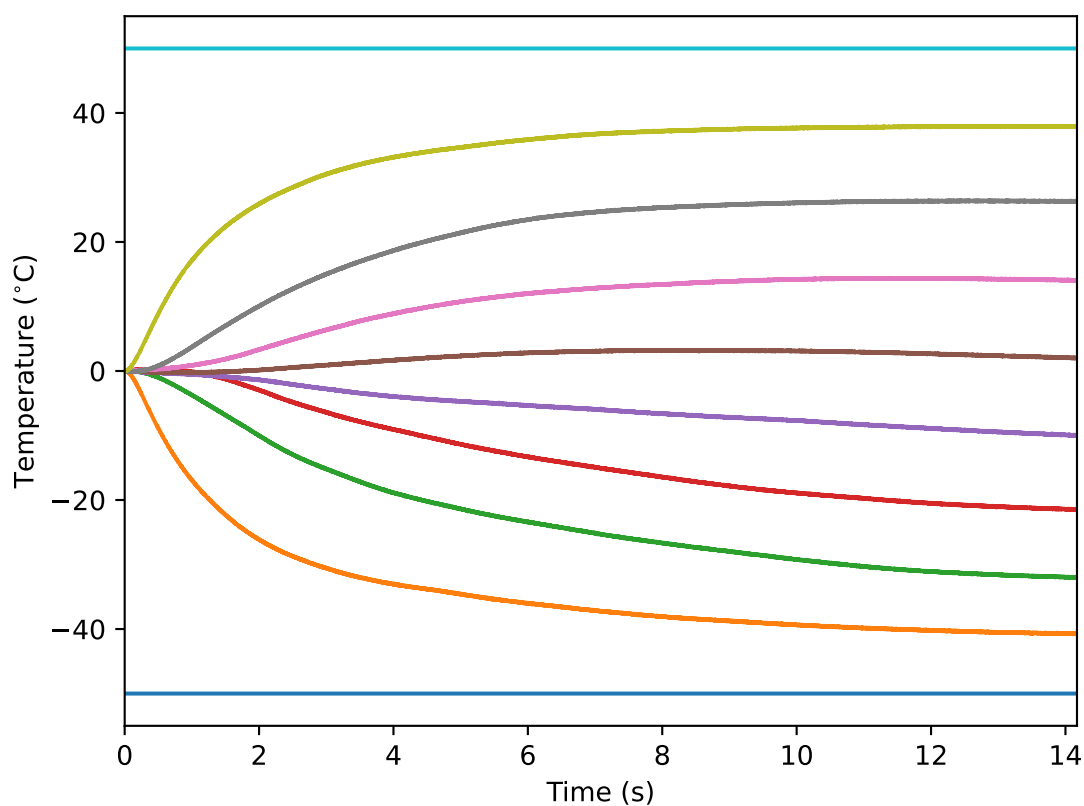
```
1  """Module solves heat equation in 1D using finite difference method and nengo_pde."""
2  from nengo_pde import Solver1D
3  import matplotlib.pyplot as plt
4  %config InlineBackend.figure_formats = ['svg']
5
6
7  def feedback_connection(u):
8      return - (K/dx**2) * 2*u
9
10
11  def lateral_connection(u):
12      return K/dx**2 * u
13
14
15  # Nengo simulation
16  t_steps = 80 # Number of time steps
17  x_steps = 8 # Number of x steps
18  neurons = 500 # Number of neurons
19  radius = 100 # Radius of neurons
20  boundaries = [-50, 50] # Constant boundary conditions
21  solver = Solver1D(feedback_connection, lateral_connection)
22
23  # Grid properties
24  K = 4.2
25  x_len = 20 # mm
26  dx = x_len/x_steps
27  dt = dx**2/(2*K**2) # dt chosen for stability
28
29  # Run finite difference method simulation
30  solver.run_FDM_order1(dt, t_steps, x_steps, boundaries)
31  fig, ax = solver.plot_population(dt, False)
32  ax.set_xlabel('Time (s)')
33  ax.set_ylabel('Temperature ( $\varnothing$ C)')
34  plt.show()
35  fig, ax = solver.plot_grid(t_steps, False)
36  ax.set_xlabel('Time (s)')
37  ax.set_ylabel('Temperature ( $\varnothing$ C)')
38  plt.show()
39  solver.animate(nframes=t_steps)
40
41  # Run nengo_pde simulation
42  solver.run_nengo_order1(dt, t_steps, x_steps, boundaries, neurons, radius)
43  fig, ax = solver.plot_population(0.001, False)
44  ax.set_xlabel('Time (s)')
45  ax.set_ylabel('Temperature ( $\varnothing$ C)')
46  plt.show()
47  fig, ax = solver.plot_grid(t_steps, False)
48  ax.set_xlabel('Time (s)')
49  ax.set_ylabel('Temperature ( $\varnothing$ C)')
```

```
50 plt.show()
51 solver.animate(nframes=t_steps)
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
```



Time: 14.17233560090703





Out[14]: <matplotlib.animation.FuncAnimation at 0x25558739d60>

There is an interesting paper with the title: *Variational Monte Carlo Approach to Partial Differential Equations with Neural Networks*, available on the arXiv eprint site under the name arXiv:2206.01927v2. (Because it deals with both Monte Carlo as well as Neural Networks, the work could be referred to either of two of these pde notebooks.) The Github repository for the project is RehMoritz/vmc\_pde.

(Although the arXiv site copy of the paper has good integrity when viewed online, two of the figures may be mangled on PDF download. Preserving all rendering details is possible if first transferred to a technically capable reader such as Sumatra or Atril.)

Following are instructions for getting a Jupyter notebook working to show the example study from the repository, for two OS platforms:

For Windows:

1. From the site: <https://whls.blob.core.windows.net/unstable/index.html> download one of the following versions of Jaxlib,   
cpu/jaxlib-0.1.75-cp37-none-win\_amd64.whl  
cpu/jaxlib-0.1.75-cp38-none-win\_amd64.whl  
cpu/jaxlib-0.1.75-cp39-none-win\_amd64.whl  
the choice depending on whether Python 3.7, 3.8, or 3.9 is available for use. Place the downloaded file in the Jupyter working directory.
2. Install the jaxlib file, from within Jupyter, with the command `!pip3 install jaxlib-0.1.75-cp39-none-win_amd64.whl`.
3. Install the module flax, from within Jupyter, with the command `!pip3 install flax==0.3.6`.
4. If a version of jax is installed, uninstall it or delete the site-package.
5. If action was taken in 4. above, a kernel restart is probably called for.
6. Install the module jax, from within Jupyter, with the command `!pip3 install jax==0.2.18`. Pip should do the install in spite of a slight disagreement concerning dependencies.
7. Download the zipped version of the repository and extract it, resulting in vmc\_pde-main. From the subdirectory 'vmc\_fluids', find 13 local modules (excluding 'main.py'). Copy these modules to the working directory of Jupyter.
8. Open the file `main.py` in Idle or wherever, then copy it to a Jupyter cell. The file should run, producing extensive printed data and seven time-sequence plots. The main program is shown below, but not run here. Instead, three of the plots are shown below it.

For Linux:

1. Install jaxlib with: `!pip3 install jaxlib==0.1.75`  
Pip should automatically choose a sub-version which is compatible with the installed Python environment.
2. Install the module flax, from within Jupyter, with the command `!pip3 install flax==0.3.6`.
3. If a version of jax is installed, uninstall it or (easier) delete two site-package folders: jax and jax.<..>-dist-info.
4. If action was taken in 3. above, a kernel restart is probably called for.
5. Install the module jax, from within Jupyter, with the command `!pip3 install jax==0.2.18`. Pip should do the install in spite of a slight disagreement concerning dependencies.
6. If they are not installed already, use pip to install mpi4py and h5py.
7. Download the zipped version of the repository and extract it, resulting in vmc\_pde-main. From the subdirectory 'vmc\_fluids', find 13 local modules (excluding 'main.py'). Copy these modules to the working directory of Jupyter.
8. Open the file `main.py` in Idle or wherever, then copy it to a Jupyter cell. The file should run, producing extensive printed data and seven time-sequence plots. The main program is shown below, but not run here. Instead, three of the plots are shown below it.

```
In [ ]: 1 import jax
2 jax.config.update("jax_enable_x64", True)
3 import jax.numpy as jnp
4 import flax.linen as nn
5 import numpy as np
6 import matplotlib.pyplot as plt
7 %config InlineBackend.figure_formats = ['svg']
8 import os
9 import time
10
11 import global_defs
12 import var_state
13 import sampler
14 import net
15 import grid
16 import train
17 import evolutionEq
18 import tdvp
19 import stepper
20 import visualization
21 import mpi_wrapper
22 import util
23
24
25 def norm_fun(v, S):
26     # norm_fun for the timesteps
27     return v @ S @ v
28
29
30 # Initializing the net
31 initKey = 1
32 sampleKey = 1
33
34 mode_dict = {"fluidpaper": {"offset": jnp.ones(2) * 0.25, "dim": 2, "latent_space_name": "cos_dist", "mcmcbound": 0.25, "grid": 1},
35              "harmonicOsc": {"offset": jnp.ones(2) * 1, "dim": 2, "latent_space_name": "Gauss", "mcmcbound": 0.25, "grid": 1},
36              "harmonicOsc_diff": {"offset": jnp.array([1, 0, 0, 1, 0, 0]) * 1, "dim": 6, "latent_space_name": "Gauss", "mcmcbound": 0.25, "grid": 1},
37              "diffusion": {"offset": jnp.zeros(8), "dim": 8, "latent_space_name": "Student_t", "mcmcbound": 0.25, "grid": 1},
38              "diffusion_anisotropic": {"offset": jnp.zeros(12), "dim": 12, "latent_space_name": "Gauss", "mcmcbound": 0.25, "grid": 1},
39              "mwe": {"offset": jnp.zeros(2), "dim": 2, "latent_space_name": "Gauss", "mcmcbound": 0.25, "grid": 1}}
40 mode = "harmonicOsc_diff"
41 mode = "diffusion"
42 mode = "mwe"
43
44 """
45 List of things that have to be set manually before starting a run:
46 - parameter nu of the student - t in BOTH (!!) sampler.py and net.py - starts with nu=2 atm.
47 - network specifications, whether to use both s and t, etc.
48     - Diffusion: noAdd
49     - harmonicOsc: DifferentAdd
```

```

50 - timestep:
51     - Diffusion: dt = 1e-7, fixed, with increasing step size, factor:, maxStep:
52     - harmonicOsc: dt=1e-4, fixed, with increasing step size, factor: 1.3, maxStep: 1e-2
53 - blocks:
54     - Diffusion:: 4, intermediate (dim//2)
55     - harmonicOsc: 4, intermediate (dim // 2)
56 - latent space covariance matrix:
57     - Diffusion: np.eye(..) + A @ A.T
58     - harmonicOsc: L @ L.T
59 """"
60
61 dim = mode_dict[mode]["dim"]
62 offset = mode_dict[mode]["offset"]
63 mcmcbound = mode_dict[mode]["mcmcbound"]
64 gridbound = mode_dict[mode]["gridbound"]
65 symgrid = mode_dict[mode]["symgrid"]
66 latent_space_name = mode_dict[mode]["latent_space_name"]
67 evolution_type = mode_dict[mode]["evolution_type"]
68
69 # set up sampler
70 sampler = sampler.Sampler(dim=dim, numChains=30, name=latent_space_name, mcmc_info={"offset": offset, "bound"
71
72 # set up variational state
73 print("Identifier -3")
74 vState = var_state.VarState(sampler, dim, initKey, 4, network_args={"intermediate": (dim // 2,) * 1, "offset":
75 print(f"Number of Model parameters: {vState.numParameters}")
76
77
78 # Some (old) sanity checks - can be removed
79 mynet = {"net": vState.net, "params": vState.params}
80 x_real = jnp.ones(dim)
81 print(mynet["params"])
82 z_latent, _ = mynet["net"].apply(mynet["params"], x_real, evaluate=False, inv=False)
83 x_real, _ = mynet["net"].apply(mynet["params"], z_latent, evaluate=False, inv=True)
84 print(z_latent)
85 print(x_real)
86
87 x_real = - jnp.ones(dim)
88 z_latent, jac = mynet["net"].apply(mynet["params"], x_real, evaluate=False, inv=False)
89 x_real, jac_inv = mynet["net"].apply(mynet["params"], z_latent, evaluate=False, inv=True)
90 print(z_latent)
91 print(x_real)
92
93 x_real = jnp.zeros(dim)
94 z_latent, jac = mynet["net"].apply(mynet["params"], x_real, evaluate=False, inv=False)
95 x_real, jac_inv = mynet["net"].apply(mynet["params"], z_latent, evaluate=False, inv=True)
96 print(z_latent)
97 print(x_real)
98
99
100 # Initializing the grid
101 if dim == 2:
102     bounds = np.ones((dim,)) * gridbound
103     n_gridpoints = 200
104     grid = grid.Grid(bounds, n_gridpoints, sym=symgrid)
105     integral = vState.integrate(grid)
106     print("Integral value:", integral)
107
108 # time evolution
109 dt = 1e-7
110 tol = 1e-2
111 maxStep = 1e-2
112 comp_integrals = False
113 # myStepper = stepper.AdaptiveHeun(timeStep=dt, tol=tol, maxStep=maxStep)
114 myStepper = stepper.FixedStepper(timeStep=dt, mode='Heun', maxStep=maxStep, increase_fac=1.3)
115 tdvpEq = tdvp.TDVP()
116 timings = util.Timings()
117 evolutionEq = evolutionEq.EvolutionEquation(dim=dim, name=evolution_type)
118 nSamplesTDVP = 10000
119 nSamplesObs = 10000
120
121 # data to learn a specific state
122 # std_dev = 1
123 # size = (1, 1000, dim)
124 # mode = "standard_normal"
125 # data, target_fun = train.gen_data(size, mode=mode, std=std_dev)
126 # net = train.train(vState, data, grid, lr=1e-3, batchsize=100, target_fun=target_fun, epoches=200)
127
128 wdir = "output/" + mode + f"/NsamplesTDVP{nSamplesTDVP}_NsamplesObs{nSamplesObs}_T10/"
129 wdir = "output/" + mode + f"/NsamplesTDVP{nSamplesTDVP}_NsamplesObs{nSamplesObs}/"
130 wdir = "output/" + mode + f"/NsamplesTDVP{nSamplesTDVP}_NsamplesObs{nSamplesObs}_Tdifferent/"
131 wdir = "output/" + mode + f"/test_NsamplesTDVP{nSamplesTDVP}_NsamplesObs{nSamplesObs}_maxStep{maxStep}/"
132 if mpi_wrapper.rank == 0:
133     try:
134         os.makedirs(wdir)
135     except OSError:
136         print("Creation of the directory %s failed" % wdir)
137     else:
138         print("Successfully created the directory %s " % wdir)
139
140 t = 0
141 t_end = 5
142 plot_every = 1e0
143
144 if dim == 2:
145     # visualization.plot_vectorfield(grid, evolutionEq)

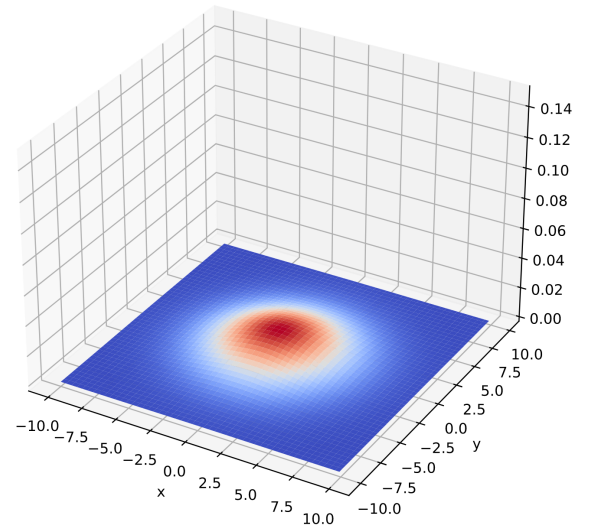
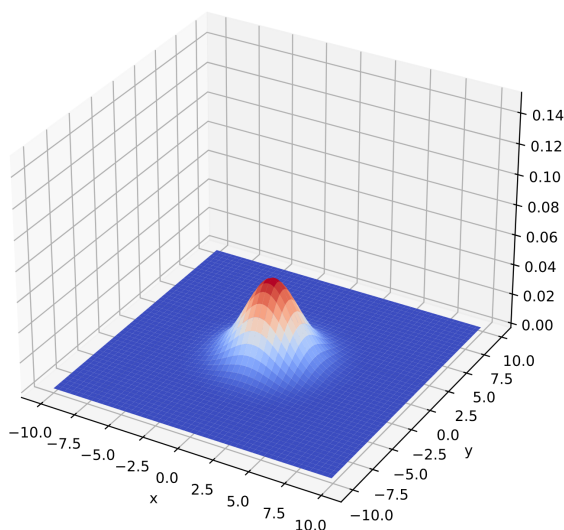
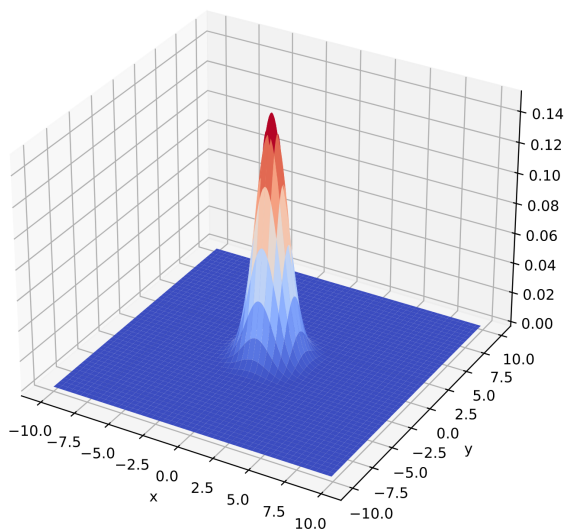
```



```

146 # plt.savefig(wdir + 'vectorfield.pdf')
147 # plt.show()
148
149 visualization.plot(vState, grid, proj=False)
150 plt.savefig(wdir + f't_{t:.3f}.pdf')
151 plt.show()
152
153 # states = vState.sample(2000000)
154 # visualization.plot_data(states, grid, title='Samples')
155 # plt.show()
156
157
158 infos = {"times": [], "ev": [], "snr": [], "solver_res": [], "tdvp_error": [], "dist_params": []}
159 n_list = []
160 while t < t_end + dt:
161     t1 = time.perf_counter()
162     dp, dt, info = myStepper.step(0, tdvpEq, vState.get_parameters(), evolutionEq=evolutionEq, psi=vState, n=
163     vState.set_parameters(dp)
164     infos["times"].append(t)
165
166     print(f"t = {t:.3f}, dt = {dt:e}")
167     print("\t Timings:")
168     timings.print_timings()
169     print(f"\t Total (in main.py): {time.perf_counter() - t1}")
170
171     print("\t Data:")
172     print(f"\t > Solver Residual = {tdvpEq.solverResidual}")
173     print(f"\t > TDVP Error = {tdvpEq.tdvp_error}")
174     if comp_integrals:
175         print(f"\t > Integral 1sigma = {info['integral_1sigma']}")
176         print(f"\t > Integral 0.5sigma = {info['integral_0.5sigma']}")
177         print(f"\t > Integral 0.1sigma = {info['integral_0.1sigma']}")
178     print(f"\t > Entropy = {info['entropy']}")
179     print(f"\t > dist params = {vState.params['params']['dist_params']}")
180     print(f"\t > Means = {info['x1']}")
181     print(f"\t > Covar = {info['covar']}")
182
183     for key in info.keys():
184         if key not in infos.keys():
185             infos[key] = []
186             infos[key].append(info[key])
187     infos["ev"].append(tdvpEq.ev)
188     infos["snr"].append(tdvpEq.snr)
189     infos["solver_res"].append(tdvpEq.solverResidual)
190     infos["tdvp_error"].append(tdvpEq.tdvp_error)
191     infos["dist_params"].append(vState.params['params']['dist_params'])
192
193     n = round(t / plot_every)
194     if np.abs(t - n * plot_every) < dt and dim == 2 and n not in n_list:
195         n_list.append(n)
196         integral = vState.integrate(grid)
197         print("Integral value:", integral)
198
199         visualization.plot(vState, grid, proj=False)
200         plt.savefig(wdir + f't_{t:.3f}.pdf')
201         plt.show()
202
203     print(vState.net.apply(vState.params, jnp.zeros(dim,), evaluate=False, inv=True)[0])
204
205     # visualization.plot_line(vState, scale=10, fit=True, offset=offset)
206     # plt.show()
207
208     t = t + dt
209
210 util.store_infos(wdir, infos)
211 visualization.make_final_plots(wdir, infos)
212 plt.show()
213
214

```



The paper describes the method as applicable to diffusion, i.e. parabolic, problem environments. Because elliptic equations are also amenable to solution by Monte Carlo techniques, it might be possible to extend the treatment to them also.

