Autosave disabled

### Chapter 15-16: 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]:
```python
"""Module solves heat equation in 1D using finite difference method and nengo_pde."""
from nengo_pde import Solver1D
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ['svg']


def feedback_connection(u):
    return - (K/dx**2) * 2*u


def lateral_connection(u):
    return K/dx**2 * u


# Nengo simulation
t_steps = 80  # Number of time steps
x_steps = 8   # Number of x steps
neurons = 500   # Number of neurons
radius = 100   # Radius of neurons
boundaries = [-50, 50]   # Constant boundary conditions
solver = Solver1D(feedback_connection, lateral_connection)

# Grid properties
K = 4.2
x_len = 20   # mm
dx = x_len/x_steps
dt = dx**2/(2*K**2)   # dt chosen for stability

# Run finite difference method simulation
solver.run_FDM_order1(dt, t_steps, x_steps, boundaries)
fig, ax = solver.plot_population(dt, False)
ax.set_xlabel('Time (s)')
ax.set_ylabel('Temperature ($^{\circ}$C)')
plt.show()
fig, ax = solver.plot_grid(t_steps, False)
ax.set_xlabel('Time (s)')
ax.set_ylabel('Temperature ($^{\circ}$C)')
plt.show()
solver.animate(nframes=t_steps)

# Run nengo_pde simulation
solver.run_nengo_order1(dt, t_steps, x_steps, boundaries, neurons, radius)
fig, ax = solver.plot_population(0.001, False)
ax.set_xlabel('Time (s)')
ax.set_ylabel('Temperature ($^{\circ}$C)')
plt.show()
fig, ax = solver.plot_grid(t_steps, False)
ax.set_xlabel('Time (s)')
ax.set_ylabel('Temperature ($^{\circ}$C)')
```

```
50  plt.show()
51  solver.animate(nframes=t_steps)
52
```
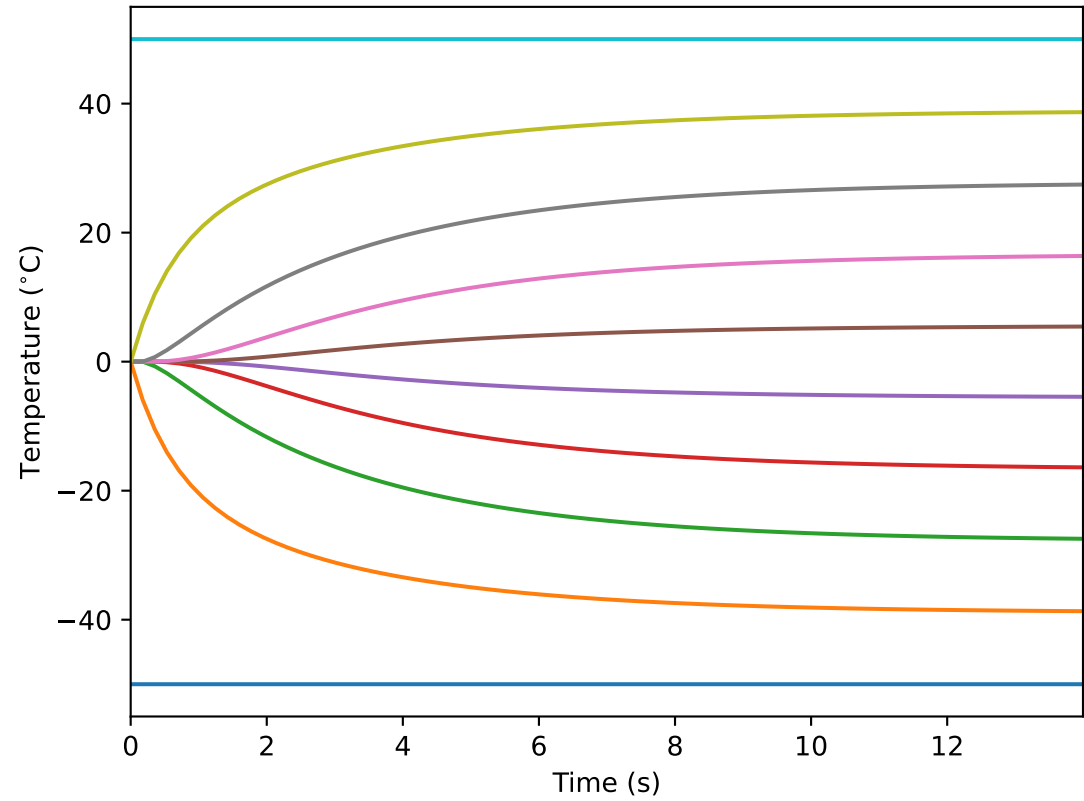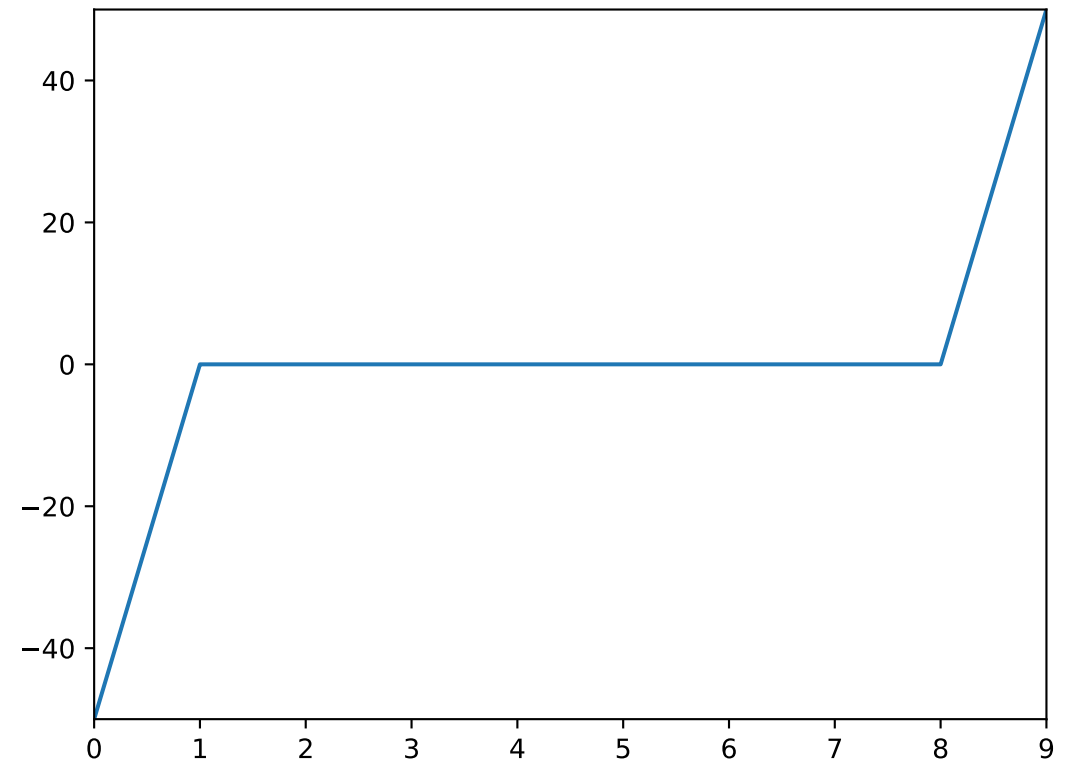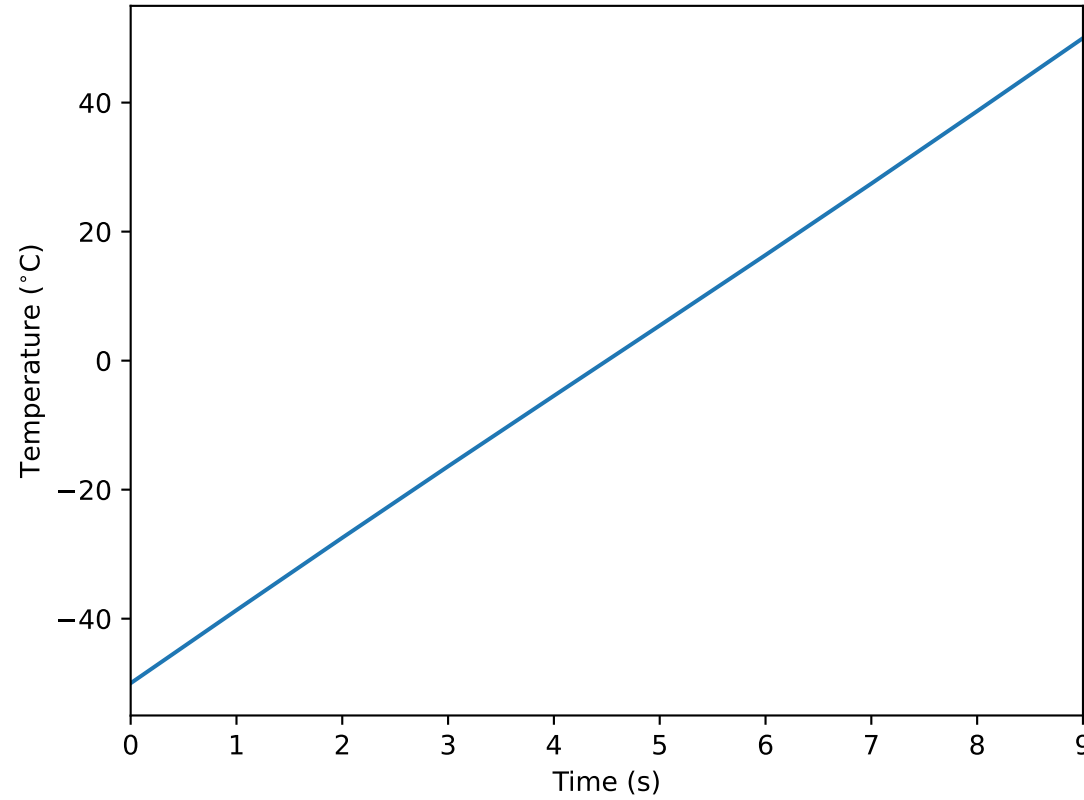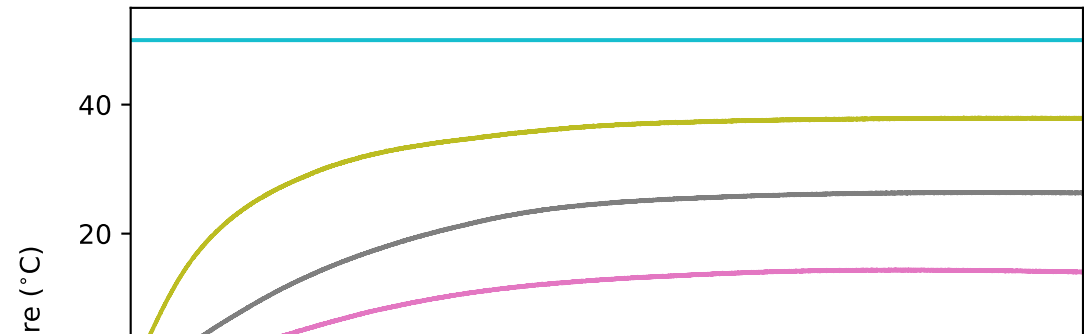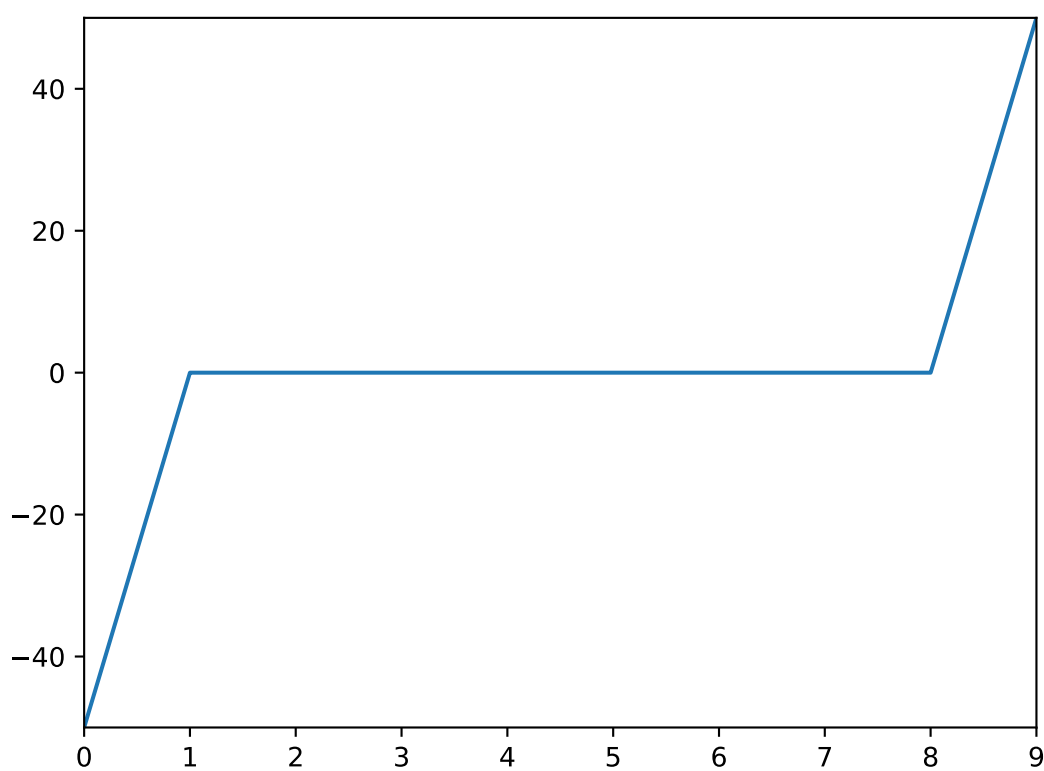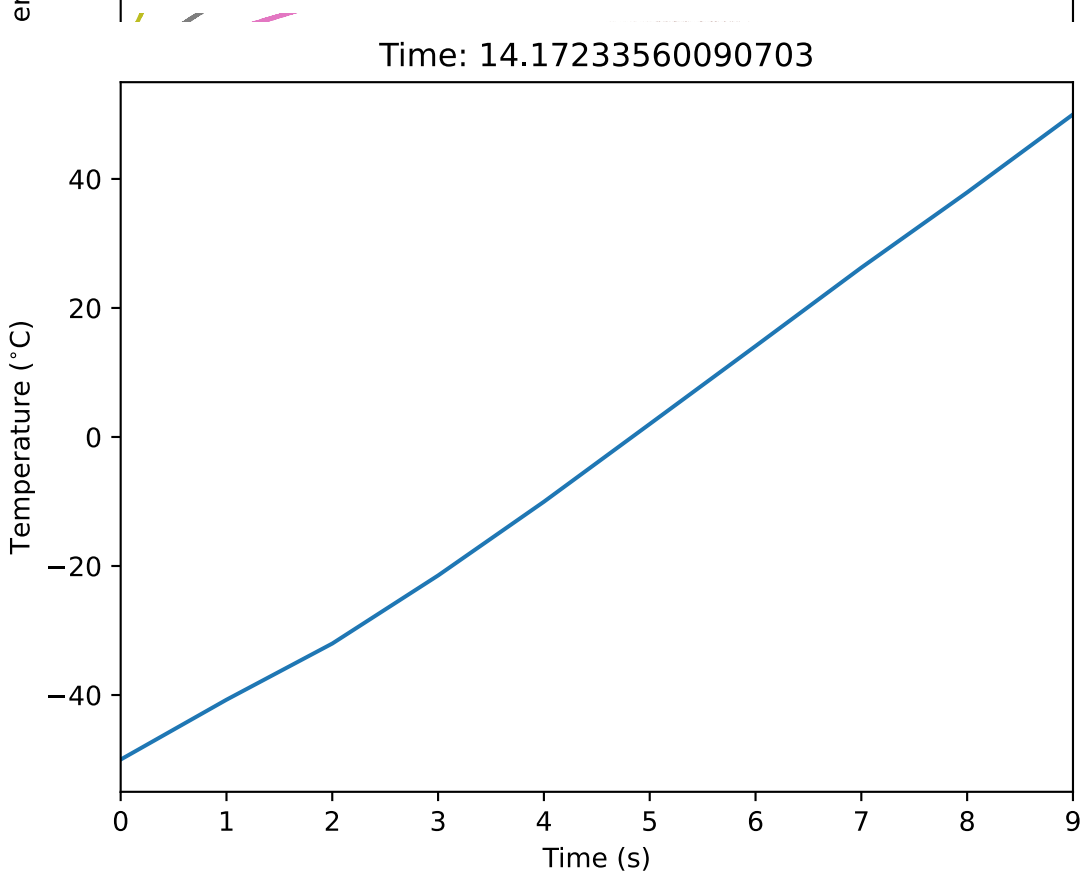


Time: 14.17233560090703





HMTL progress bar requires Jupyter Notebook >= 5.0 or Jupyter Lab. Alternatively, you can use TerminalProgressBar().

Build finished in 0:00:02.

HMTL progress bar requires Jupyter Notebook >= 5.0 or Jupyter Lab. Alternatively, you can use TerminalProgressBar().

Simulation finished in 0:00:06.

Time: 14.17233560090703

Temperature (°C)

40

20

0

−20

−40

0   1   2   3   4   5   6   7   8   9
Time (s)

40

20

0

−20

−40

0   1   2   3   4   5   6   7   8   9

`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.)

Following are instructions for getting a Jupyter notebook working to show the example study from the repository, if the platform is 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.

```python
import jax
jax.config.update("jax_enable_x64", True)
import jax.numpy as jnp
import flax.linen as nn
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ['svg']
import os
import time

import global_defs
import var_state
import sampler
import net
import grid
import train
import evolutionEq
import tdvp
import stepper
import visualization
import mpi_wrapper
import util


def norm_fun(v, S):
    # norm_fun for the timesteps
    return v @ S @ v


# Initializing the net
initKey = 1
sampleKey = 1

mode_dict = {"fluidpaper": {"offset": jnp.ones(2) * 0.25, "dim": 2, "latent_space_name": "cos_dist", "mcmcbou
             "harmonicOsc": {"offset": jnp.ones(2) * 1, "dim": 2, "latent_space_name": "Gauss", "mcmcbound":
             "harmonicOsc_diff": {"offset": jnp.array([1, 0, 0, 1, 0, 0]) * 1, "dim": 6, "latent_space_name":
             "diffusion": {"offset": jnp.zeros(8), "dim": 8, "latent_space_name": "Student_t", "mcmcbound": 0
             "diffusion_anisotropic": {"offset": jnp.zeros(12), "dim": 12, "latent_space_name": "Gauss", "mcm
             "mwe": {"offset": jnp.zeros(2), "dim": 2, "latent_space_name": "Gauss", "mcmcbound": 0.25, "grid
mode = "harmonicOsc_diff"
mode = "diffusion"
mode = "mwe"

"""
List of things that have to be set manually before starting a run:
- parameter nu of the student - t in BOTH (!!) sampler.py and net.py - starts with nu=2 atm.
- network specifications, whether to use both s and t, etc.
    - Diffusion: noAdd
    - harmonicOsc: DifferentAdd
- timestep:
    - Diffusion: dt = 1e-7, fixed, with increasing step size, factor:, maxStep:
    - harmonicOsc: dt=1e-4, fixed, with increasing step size, factor: 1.3, maxStep: 1e-2
- blocks:
    - Diffusion:: 4, intmediate (dim//2)
    - harmonicOsc: 4, intmediate (dim // 2)
- latent space covariance matrix:
    - Diffusion: np.eye(..) + A @ A.T
    - harmonicOsc: L @ L.T
"""

dim = mode_dict[mode]["dim"]
offset = mode_dict[mode]["offset"]
mcmcbound = mode_dict[mode]["mcmcbound"]
gridbound = mode_dict[mode]["gridbound"]
symgrid = mode_dict[mode]["symgrid"]
latent_space_name = mode_dict[mode]["latent_space_name"]
evolution_type = mode_dict[mode]["evolution_type"]

# set up sampler
sampler = sampler.Sampler(dim=dim, numChains=30, name=latent_space_name, mcmc_info={"offset": offset, "bound"

# set up variational state
print("Identifier -3")
vState = var_state.VarState(sampler, dim, initKey, 4, network_args={"intmediate": (dim // 2,) * 1, "offset":
print(f"Number of Model parameters: {vState.numParameters}")


# Some (old) sanity checks - can be removed
mynet = {"net": vState.net, "params": vState.params}
x_real = jnp.ones(dim)
print(mynet["params"])
z_latent, _ = mynet["net"].apply(mynet["params"], x_real, evaluate=False, inv=False)
x_real, _ = mynet["net"].apply(mynet["params"], z_latent, evaluate=False, inv=True)
print(z_latent)
print(x_real)

x_real = - jnp.ones(dim)
z_latent, jac = mynet["net"].apply(mynet["params"], x_real, evaluate=False, inv=False)
x_real, jac_inv = mynet["net"].apply(mynet["params"], z_latent, evaluate=False, inv=True)
print(z_latent)
print(x_real)

x_real = jnp.zeros(dim)
z_latent, jac = mynet["net"].apply(mynet["params"], x_real, evaluate=False, inv=False)
x_real, jac_inv = mynet["net"].apply(mynet["params"], z_latent, evaluate=False, inv=True)
```

```python
 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, nS
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
```
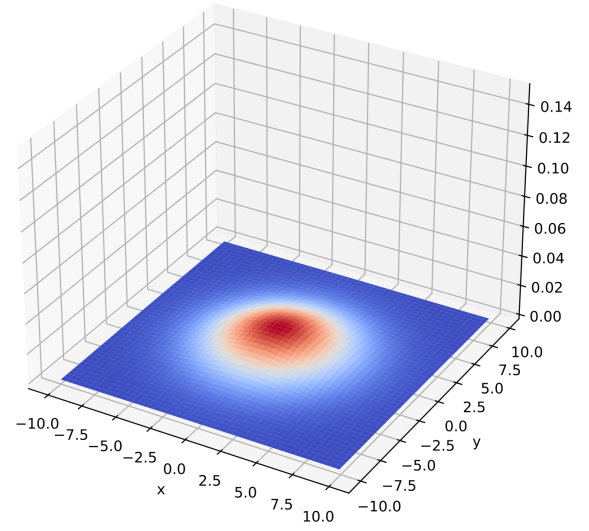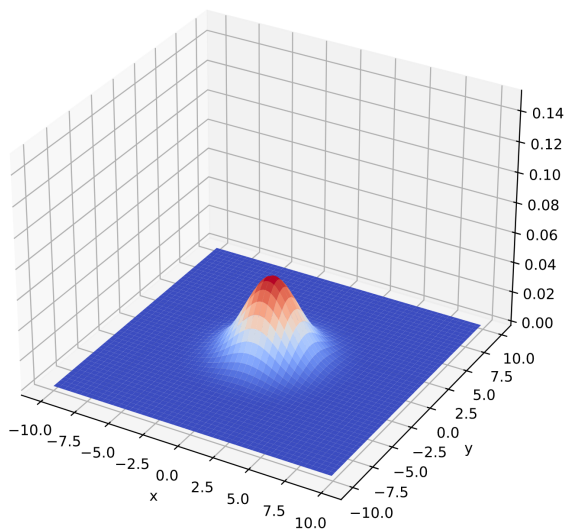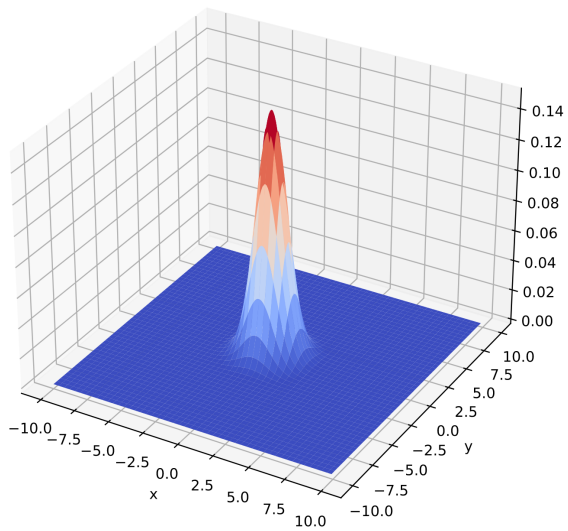


The paper describes the method as applicable to diffusion, i.e. parabolic, problem environments. Because elliptic equations are also amenable to solution by Monte Carlo techniques, it might be possible to extend the treatment to them also.