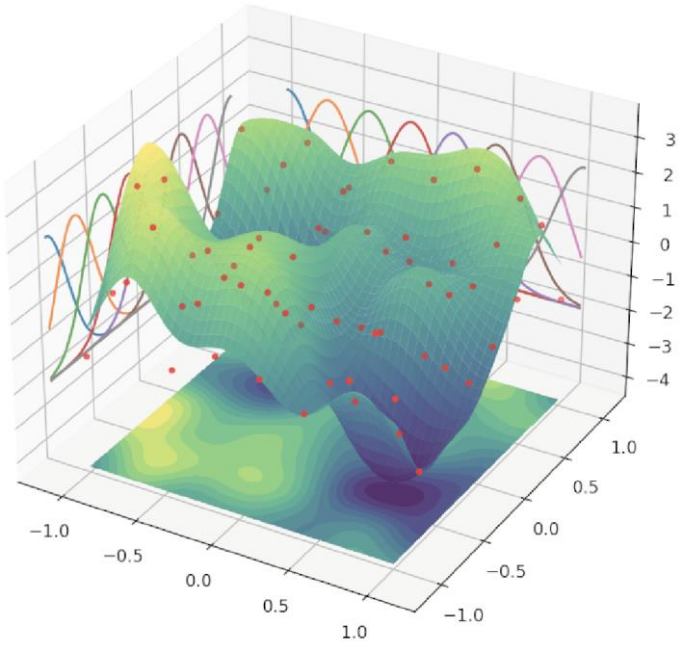
A Hybrid Framework for Fluid Flow Simulations: Combining SPH with Machine Learning

Rene Winchenbach, Nils Thuerey

Technical University Munich

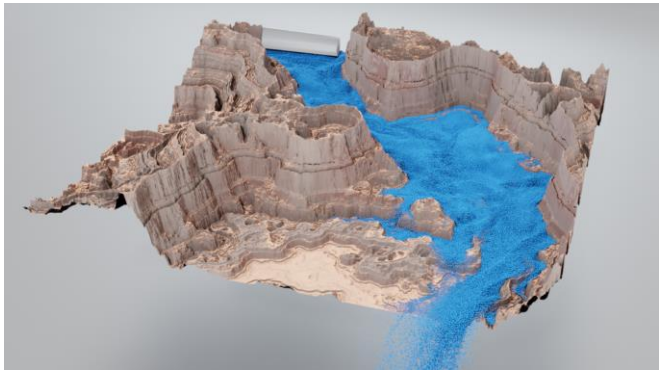


SPHERIC 2023

Rhodes, 28. June 2023

https://tinyurl.com/sphericDemo

Why Machine Learning?

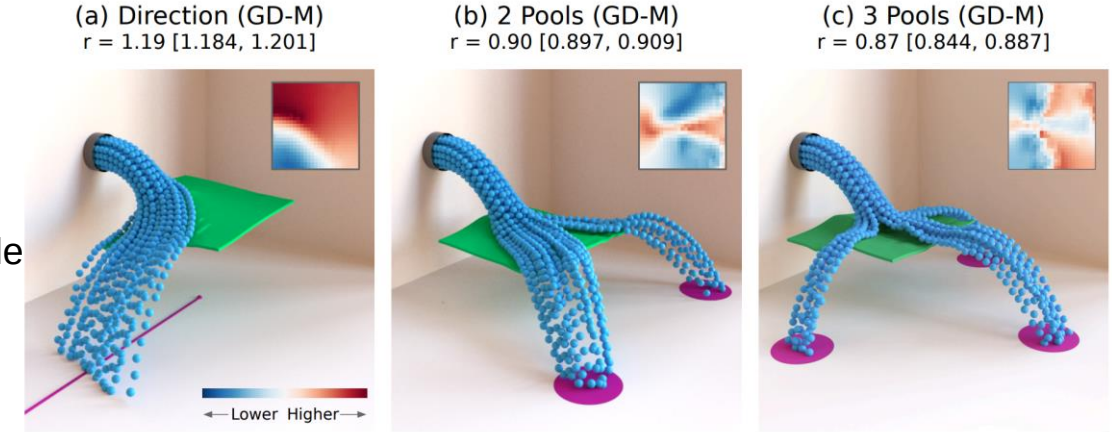


We distinguish two motivational approaches:

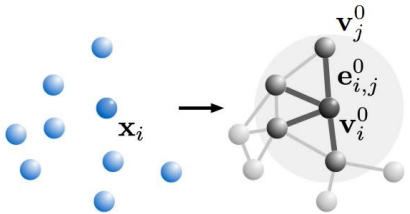
1. Computational Performance
2. Methodological limitations

The former is always helpful as faster simulations are never bad

The latter is much more useful in practice
Design optimization common in CFD
Goal: Minimize cost function w.r.t. shape
SPH is not easily differentiable
ML surrogates are automatically differentiable



Graph Convolutions and SPH



SPH is a graph convolution
A particle is a vertex of an implicit graph
Neighborhoods form the connectivity of the graph
SPH interpolations are message passing steps:

$$\langle A(\boldsymbol{x}_i)\rangle = \sum_j \frac{m_j}{\rho_j} A_j W(\boldsymbol{x}_j - \boldsymbol{x}_i)$$

A (continuous) graph convolution of feature f with g:

$$(f \star g)(\boldsymbol{x}_i) = \sum_j f_j g(\boldsymbol{x}_j - \boldsymbol{x}_i)$$

Where $f_j = \frac{m_j}{\rho_j} A_j$ and $g = W$ leads to an SPH interpolation step

GNNs *simply* learn g

What **we** will do now

The simplest SPH interpolation is a density summation step:

$$\rho_i = \sum_j m_j W_{ij}$$

We are going to learn this in 1D and we are going to do it **live**
For this case we know what the ideal graph convolution looks like
Ideally g should be equal to W (for our case a Wendland 2 kernel function)

PyTorch

PyTorch is a library for python developed by Meta
It is focused on tensors, which are ubiquitous in neural networks
Interface very similar to numpy for mathematical operations
Builtin neural network layer modules
Ecosystem of libraries, e.g., for GNNs
Supports automatic differentiation
Supports automatic parallelization

```python
import torch

# Create two tensors with requires_grad=True
x = torch.tensor(2.0, requires_grad=True)
y = torch.tensor(3.0, requires_grad=True)

# Perform some operations on the tensors z = x^2 + y^2
z = x * x + y * y

# Call backward() to compute gradients
z.backward()

# Check the gradients
print(f"Gradient of z with respect to x: {x.grad}")
print(f"Gradient of z with respect to y: {y.grad}")
```

$\frac{\partial}{\partial x} z\Big|_{x=2,y=3}$

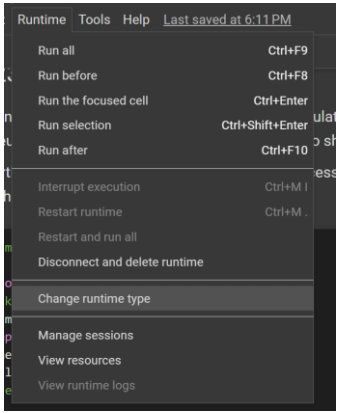
$\frac{\partial}{\partial y} z\Big|_{x=2,y=3}$

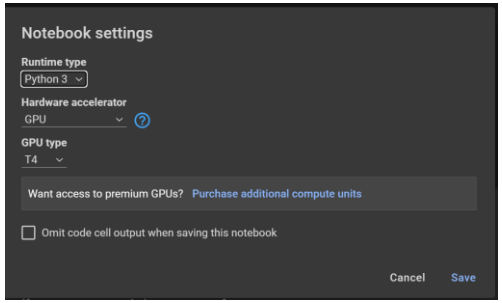
Google Colab

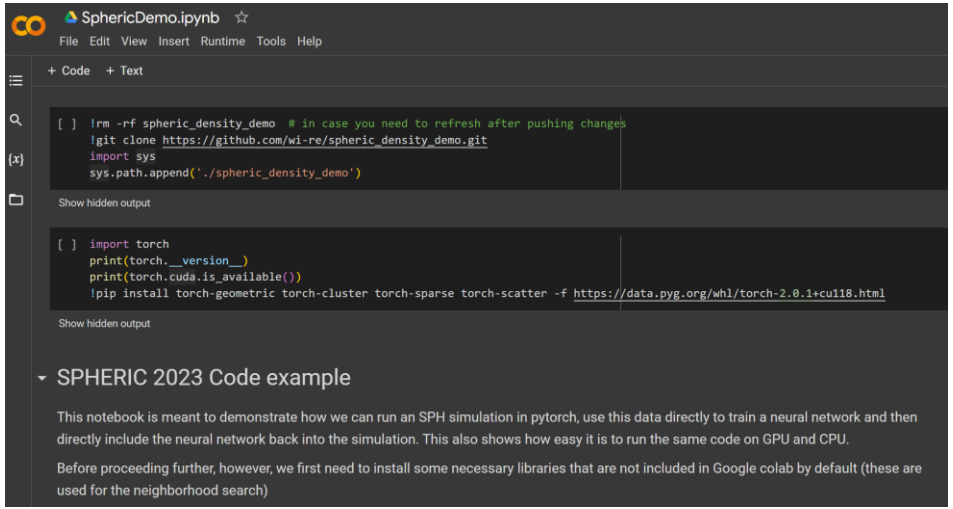
An important aspect of research and education is accessibility
Not everyone has powerful hardware to run simulations
There are many cloud based platforms to run computations on
Google Colab allows for code to be ran on GPUs by anyone with a Google account for free
Interface very similar to Jupyter Notebooks
Colab notebooks can be shared via simple links and require no local installations
But there are limitations (certainly to the free version)

Google Colab

This is what the demo should look like
First we make sure that we have GPU support







Note that the notebook can also be run without any GPU acceleration and instead using CPU parallelization

Demo I – Installing Prerequisites

To get going we need to install some python modules and libraries:
We can directly clone a Github repository with our own modules:

```
[ ]  !rm -rf spheric_density_demo  # in case you need to refresh after pushing changes
     !git clone https://github.com/wi-re/spheric_density_demo.git
     import sys
     sys.path.append('./spheric_density_demo')

     Cloning into 'spheric_density_demo'...
     remote: Enumerating objects: 29, done.
     remote: Counting objects: 100% (29/29), done.
     remote: Compressing objects: 100% (23/23), done.
     remote: Total 29 (delta 11), reused 20 (delta 5), pack-reused 0
     Unpacking objects: 100% (29/29), 4.80 MiB | 8.48 MiB/s, done.
```

And then install PyTorch Geometric for GNN features using pip (make sure the versions match)

```
[ ]  import torch
     print(torch.__version__)
     print(torch.cuda.is_available())
     !pip install torch-geometric torch-cluster torch-sparse torch-scatter -f https://data.pyg.org/whl/torch-2.0.1+cu118.html
```

Demo II – Loading Modules

Next we import the python modules we need for this demo and are ready to go

```
[ ]  # sph related imports
     from sph import *
     from perlin import *
     # neural network rlated imports
     from torch.optim import Adam
     from rbfConv import *
     from torch_geometric.loader import DataLoader
     from trainingHelper import *
     # plotting/UI related imports

     from plotting import *
     import matplotlib as mpl
     plt.style.use('dark_background')
     cmap = mpl.colormaps['viridis']
     from tqdm.notebook import trange, tqdm
     from IPython.display import display, Latex
```

Governing Equations

Compressible 1D SPH simulation $\frac{d\boldsymbol{u}}{dt} = \frac{\nabla p}{\rho} + \left(\frac{du}{dt}\right)_{diss}$

Pressure based on ideal gas EOS $p_i = \kappa\,(\rho_i - \rho_0); \frac{\nabla p_i}{\rho_i} = \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right) \nabla_i W_{ij}$

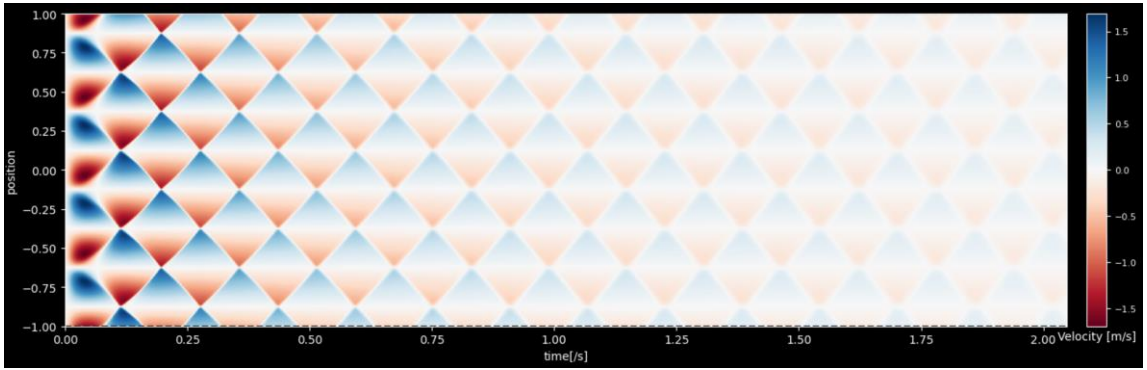
Kinematic viscosity based on standard SPH formulations $\left(\frac{d\boldsymbol{u}_i}{dt}\right)_{diss} = -\sum_j m_j \frac{-\alpha c_s \mu_{ij} + \beta \mu_{ij}^2}{(\rho_i + \rho_j)/2} \nabla_i W_{ij}; \mu_{ij} = h \frac{u_{ij} \cdot r_{ij}}{r_{ij} + \epsilon h^2}$

Density summation based $\rho_i = \sum_j m_j W_{ij}$

RK4 time integration

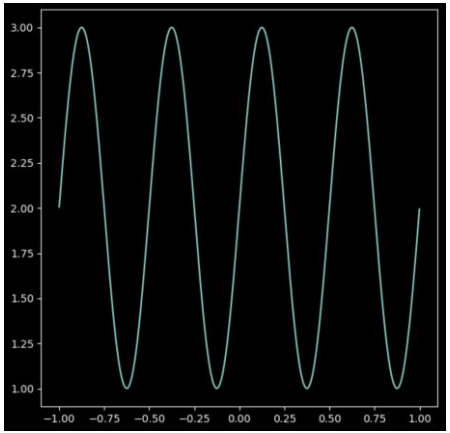
Explicit fixed time stepping

Periodic Boundary Conditions



Simulation Initialization

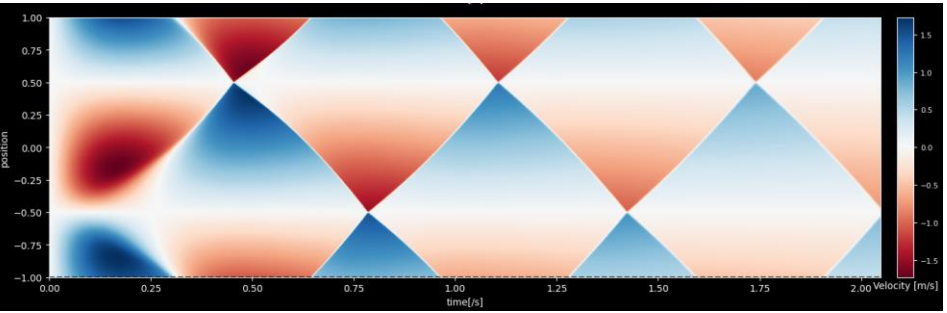
The simulation is initialized based on a desired density profile defined using a PDF
The default setup is a sinusoidal density profile
This profile creates a very regular simulation outcome that is easy to understand
The demo also comes with many other initial PDFs, including random ones
Simulation parameters:



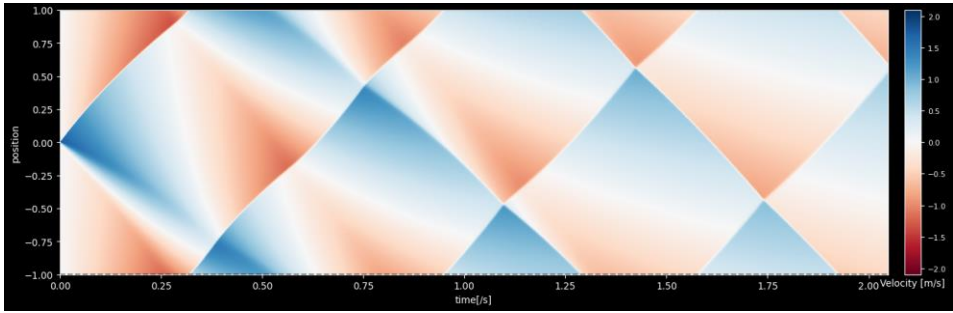

```
# simulation parameters
minDomain = -1 # minimum domain, leave at -1 for the most part
maxDomain = 1 # maximum domain, leave at 1 for the most part
# change base area to change initial starting density
baseArea = 2 / numParticles * 2
particleRadius = baseArea / 2.0
# change particle support to make simulation more/less smooth
particleSupport = particleRadius * 8.
# SPH parameters
xsphConstant = 0.0
diffusionAlpha = 1. # kinematic viscosity coefficient
diffusionBeta = 2.
kappa = 10 # EOS kappa term
restDensity = 1000 # EOS rest density term
dt = 1e-3 # fixed global timestep
c0 = 10 # speed of sound used in kinematic viscosity
```

Example Simulations

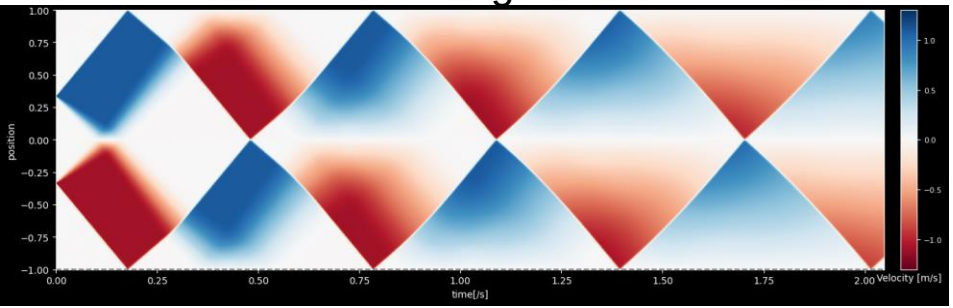
Sinusoidal



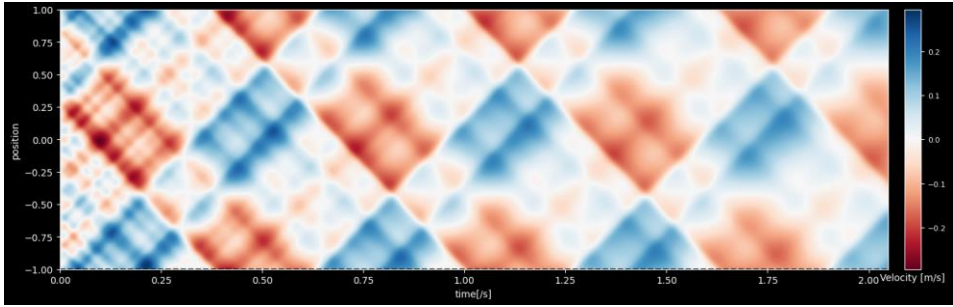
Sawtooth



Rectangular



Random



Demo III – Running the Simulation

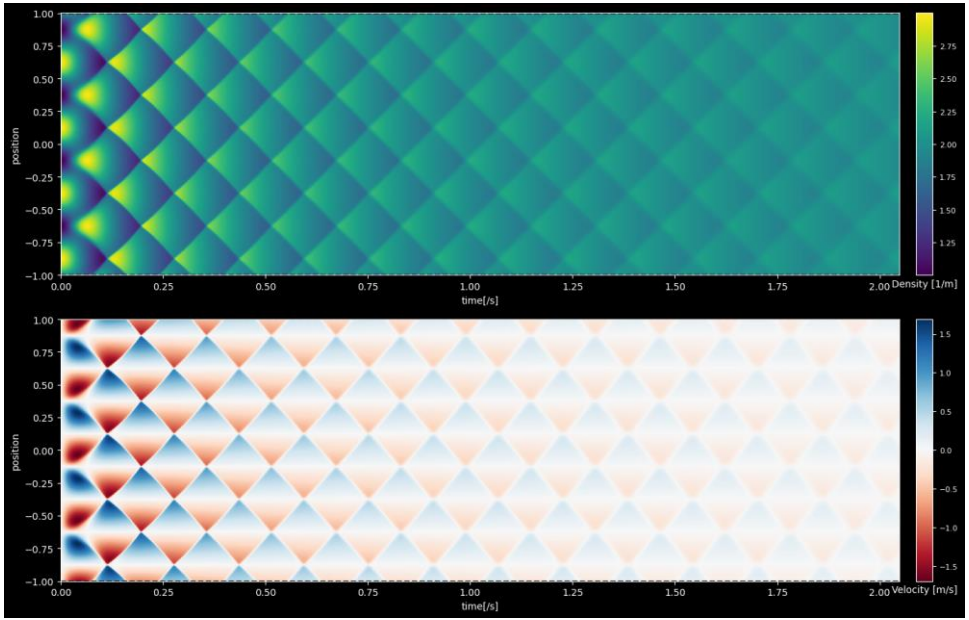
Next we run the simulation with 4 substeps

We also write out a lot of data to process it later

```
[ ] # run the simulation using RK4
    for i in tqdm(range(timesteps)):
        # Compute state for substep 1
        v1 = torch.clone(fluidVelocities)
        # RK4 substep 1
        dudt_k1, dxdt_k1, fluidDensity, fluidPressure = computeUpdate(fluidPositions, fluidVelocities, fluidAreas, minDomain, maxDomain, kappa, restDensity, diffusionAlpha, diffusionBeta, c0, xsphConstant, particleSupport, dt)
        # Compute state for substep 2
        x_k1 = fluidPositions + 0.5 * dt * dxdt_k1
        x_k1[x_k1 < minDomain] += maxDomain - minDomain
        x_k1[x_k1 > maxDomain] -= maxDomain - minDomain
        u_k1 = fluidVelocities + 0.5 * dt * dudt_k1
        # RK4 substep 2
        dudt_k2, dxdt_k2, _, _ = computeUpdate(x_k1, u_k1, fluidAreas, minDomain, maxDomain, kappa, restDensity, diffusionAlpha, diffusionBeta, c0, xsphConstant, particleSupport, 0.5 * dt)
        # Compute state for substep 2
        x_k2 = fluidPositions + 0.5 * dt * dxdt_k2
        x_k2[x_k2 < minDomain] += maxDomain - minDomain
        x_k2[x_k2 > maxDomain] -= maxDomain - minDomain
        u_k2 = fluidVelocities + 0.5 * dt * dudt_k2
        # RK4 substep 3
        dudt_k3, dxdt_k3, _, _ = computeUpdate(x_k2, u_k2, fluidAreas, minDomain, maxDomain, kappa, restDensity, diffusionAlpha, diffusionBeta, c0, xsphConstant, particleSupport,  0.5 * dt)
        # Compute state for substep 4
        x_k3 = fluidPositions + dt * dxdt_k3
        x_k3[x_k3 < minDomain] += maxDomain - minDomain
        x_k3[x_k3 > maxDomain] -= maxDomain - minDomain
        u_k3 = fluidVelocities + dt * dudt_k3
        # RK4 substep 4
        dudt_k4, dxdt_k4, _, _ = computeUpdate(x_k3, u_k3, fluidAreas, minDomain, maxDomain, kappa, restDensity, diffusionAlpha, diffusionBeta, c0, xsphConstant, particleSupport, dt)
        # RK substeps done, store current simulation state for later processing/learning. density and pressure are based on substep 1 (i.e., the starting point for this timestep)
        simulationStates.append(torch.stack([fluidPositions, fluidVelocities, fluidDensity, fluidPressure, dt/6 * (dudt_k1 + 2* dudt_k2 + 2 * dudt_k3 + dudt_k4), dudt_k1, dudt_k2, dudt_k3, dudt_k4, fluidAreas]))
        # time integration using RK4 for velocity
    #     fluidVelocities = fluidVelocities + dt * dudt_k1 # semi implicit euler mode
        fluidVelocities = fluidVelocities + dt/6 * (dudt_k1 + 2* dudt_k2 + 2 * dudt_k3 + dudt_k4)
        fluidPositions = fluidPositions + dt * fluidVelocities
        # enforce periodic boundary conditions
        fluidPositions[fluidPositions < minDomain] += maxDomain - minDomain
        fluidPositions[fluidPositions > maxDomain] -= maxDomain - minDomain
    # After the simulation has run we stack all the states into one large array for easier slicing and analysis
    simulationStates = torch.stack(simulationStates)
```

Demo IV – Simulation Results

The results should look like this:



Datasets and You

An important aspect of neural networks is the data generation/acquisition
For this demo we only have one simulation setup for training and testing
This is not ideal but suffices for this demonstration
Ideally the dataset should be as varied as possible
This includes using data augmentation to include simulation states that aren't possible in SPH
Batching the training can improve training stability but increases memory pressure

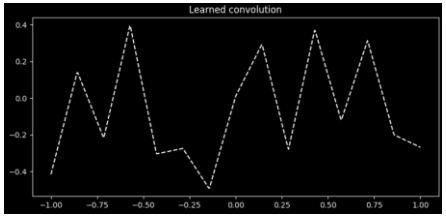
```
[ ]  ignoredTimesteps = 256
     batchSize = 4 # training batch size
     # Training done on all timesteps except the last ignoredTimesteps
     timestamps = np.arange(0,simulationStates.shape[0] - ignoredTimesteps)
     testBatch = np.arange(len(simulationStates) - ignoredTimesteps, len(simulationStates))

     # create pytorch dataloader (via pytorch geometric for convenience)
     dataLoader = DataLoader(timestamps, shuffle=True, batch_size = batchSize).batch_sampler
     dataIter = iter(dataLoader)
```

Demo V – Neural Network Setup

Next we setup a single layer neural network with empirically chosen hyperparameters:

```
[ ]  # Hyperparameters for the NN
     lr = 1e-1 # Learning rate
     iterations = 1000 # update iterations per epoch
     epochs = 5 # total number of epochs, LR is halved every epoch
     n = 15 # number of weights per continuous convolution
     basis = 'linear' # basis for the convolution, set to linear for CConv
     computeBatchSize = 128 # higher number = faster processing but more memory consumption (not relevant for small simulations)
     windowFn = None # getWindowFunction('Wendland2_1D')
     normalized = False # rbf normalization
```

We can then look at the initialized neural network and what the convolution looks like:



Which is certainly not a compact gaussian.

Demo VI – Neural Network Training

We now train the neural network:

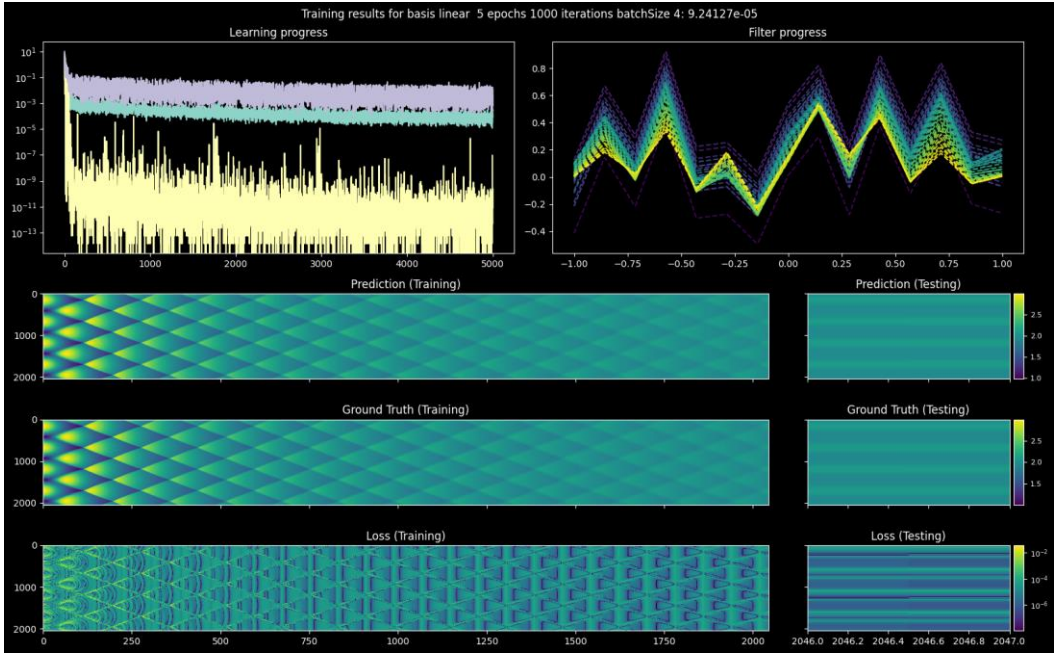
```
[ ]  # create progress bar and arrays to hold the learning progress
     pb = tqdm(range(epochs * iterations))
     lossArray = []
     batches = []
     weights = []
     testing = []
     # the actual learning process
     for epoch in range(epochs):
         losses = []
         b, l, w = processDataLoaderIter(pb, iterations, epoch, lr, \
                                         dataLoader, dataIter, batchSize, model, optimizer, \
                                         simulationStates, minDomain, maxDomain, particleSupport, \
                                         lossFunction, getFeatures, getGroundTruth, None,\
                                         train = True, prefix = '', augmentAngle = False, augmentJitter = False, jitterAmount = 0.01)
         lossArray.append(l)
         batches.append(b)
         weights.append(w)
         if True: #epoch % 1 == 0 and epoch > 0:
             lr = lr * 0.5
             for param_group in optimizer.param_groups:
                 param_group['lr'] = 0.5 * param_group['lr']
```

The learning rate is halved every 1000 weight updates to improve training stability

We also write out a lot of information to visualize the process

Demo VII – Neural Network Evaluation

The training results should look like this:
Prediction and ground truth are very similar
The learned convolution is still not gaussian
The training loss is small compared to the numerical precision we get from 32 bit floats
So, is this good enough?



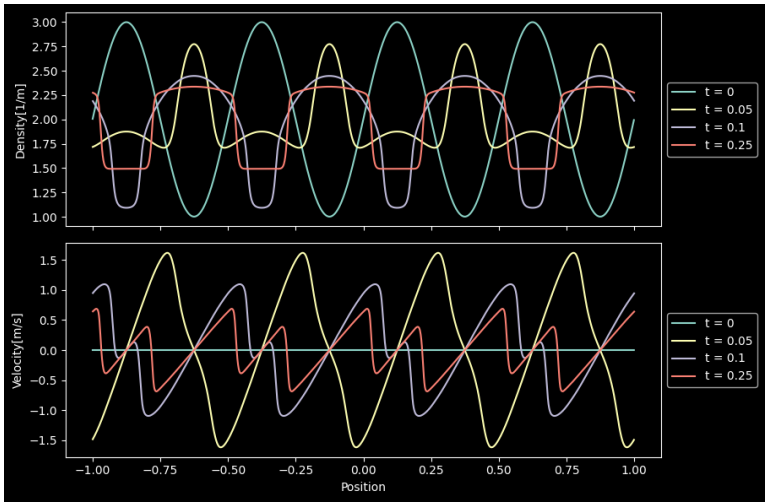
Building a hybrid simulation

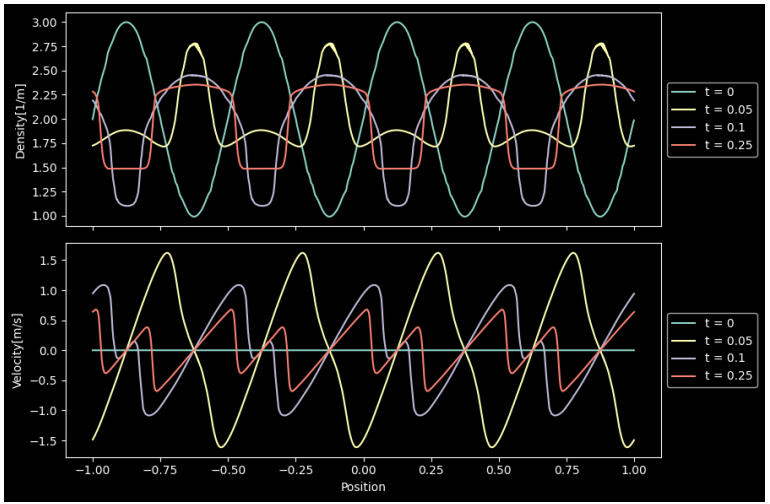
As the trained network is a graph convolution, including this into the simulation is straight forward:

```python
# SPH simulation step, returns dudt, dxdt as well as current density and pressure
def computeUpdateML(model, fluidPositions, fluidVelocities, fluidAreas, minDomain, maxDomain, kappa, restDensity, diffusionAlpha, diffusionBeta, c0, xsphCoefficient, particleSupport, dt):
    #  1. Create ghost particles for our boundary conditions
    ghostPositions = createGhostParticles(fluidPositions, minDomain, maxDomain)
    #  2. Find neighborhoods of all particles:
    fluidNeighbors, fluidRadialDistances, fluidDistances = findNeighborhoods(fluidPositions, ghostPositions, particleSupport)

    # ------------------------------------------------------------------------------------------------------------------------------------------------------ #
    #  3. Compute \rho using the trained neural network
    features = getFeatures(fluidPositions, fluidAreas, fluidVelocities, None)
    with torch.no_grad():
        fluidDensity = model((features[:,None], features[:,None]), fluidNeighbors, fluidRadialDistances[:,None] * torch.sign(fluidDistances[:,None]))[:,0]
    # ------------------------------------------------------------------------------------------------------------------------------------------------------ #

    #  4. Compute the pressure of each particle using an ideal gas EOS
    fluidPressure = (fluidDensity - 1.0) * kappa * restDensity
    #  5. Compute the XSPH term and apply it to the particle velocities:
    xsphUpdate = computeXSPH(fluidPositions, fluidVelocities, fluidDensity, fluidAreas, particleSupport, xsphCoefficient, fluidNeighbors, fluidRadialDistances)
    #  6. Compute pressure forces and resulting acceleration
    fluidPressureForces = computePressureForces(fluidPositions, fluidDensity, fluidPressure, fluidAreas, particleSupport, restDensity, fluidNeighbors, fluidRadialDistances, fluidDistances)
    fluidAccel = fluidPressureForces
    # 7. Compute kinematic viscosity
    laminarViscosity = computeDiffusion(fluidPositions, fluidVelocities, fluidAreas, fluidDensity, particleSupport, restDensity, diffusionAlpha, diffusionBeta, c0, fluidNeighbors, fluidRadialDistances, fluidDistances)
    fluidAccel += xsphUpdate / dt + laminarViscosity
    return fluidAccel, fluidVelocities, fluidDensity, fluidPressure
```

Demo VII – Hybrid Simulation Results

After doing this we get the hybrid simulation on the right compared to the ground truth on the left





Conclusions

Machine Learning is probably inevitable even in CFD
There are many opportunities for research in this direction
Making research more accessible is important
Trusting neural networks is a major challenge in many fields
Conducting open research that can be verified and repeated even more important than in classical research

https://tinyurl.com/sphericDemo

Scan the QR Code on the right to go to the Google Colab notebook and try it out yourself!
Or go to https://github.com/wi-re/spheric_density_demo.git also has the presentation slides