Automated NN generation: from the symbolic computation to the design of NN architectures for numerical predictions

O. Pannekoucke[1,2,3,*] and R. Fablet[4]

[1] INPT-ENM, Toulouse, France. [2] CNRM, Université de Toulouse, Météo-France, CNRS, Toulouse, France.
[3] CERFACS, Toulouse, France. [4] IMT-Atlantic, UMR CNRS Lab-STICC, Brest, France.

[*] contact: olivier.pannekoucke@meteo.fr

1. Context

Bridging physics and deep learning is a topical challenge. While deep learning frameworks open avenues in physical science, the design of physically-consistent deep neural network architectures is an open issue. In the spirit of physics-informed NNs, *PDE-NetGen* package provides new means to automatically translate physical equations, given as PDEs, into neural network architectures [Pannekoucke and Fablet, 2020]. *PDE-NetGen* combines symbolic calculus and a neural network generator. The later exploits NN-based implementations of PDE solvers using Keras. With some knowledge of a problem, *PDE-NetGen* is a plug-and-play tool to generate physics-informed NN architectures. They provide computationally-efficient yet compact representations to address a variety of issues, including among others adjoint derivation, model calibration, forecasting, data assimilation as well as uncertainty quantification. As an illustration, the workflow is first presented for the 2D diffusion equation, then applied to the data-driven and physics-informed identification of uncertainty dynamics for the Burgers equation.
PDE-NetGen is available on github `https://github.com/opannekoucke/pdenetgen`.
This work is part of the project KAPA (KAlman PAramétrique), founded by the LEFE INSU.

2. Neural Network Generation from symbolic PDEs

The neural network code generator presented here focuses on physical processes given as evolution equations which writes

$$\partial_t u = \mathcal{M}(u, \partial^\alpha u), \tag{1}$$

where u denotes either a scalar field or multivariate fields, $\partial^\alpha u$ denotes partial derivatives with respect to spatial coordinates, and $\mathcal{M}$ is the generator of the dynamics

NN implementation of spatial derivatives as Convolutional layers (CNN)

The core of the generator is the automatic translation of partial derivatives with respect to spatial coordinates into convolutional neural networks, this leverages on the finite-difference discretization of spatial derivatives. For instance, the finite-difference of $\partial_x^2 u$ leads to approximate the derivative by

$$\mathcal{F}_x^2 u(t,x) = \frac{u(t,x+\delta x) + u(t,x-\delta x) - 2u(t,x)}{\delta x^2} \approx \partial_x^2 u + O(\delta x^2) \tag{2}$$

where δx stands for the discretization space step. This makes appear a kernel stencil $k = [1/\delta x^2, -2/\delta x^2, 1/\delta x^2]$ that can be used in a 1D convolution layer with a linear activation function and without bias. A similar routine applies for 2D and 3D geometries. PDE-NetGen relies on the computer algebra system sympy [Meurer et al., 2017] to compute the stencil as well as to handle symbolic expressions. The computation of the spatial derivative are consistent at the second order and applies for partial derivative with respect to multiple coordinates *e.g.* ∂^3_{xxy}.

Implementation of time-scheme by using Residual Networks

Then, the time integration can be implemented either by a solver or by a ResNet architecture of a given time scheme e.g. an Euler scheme or a fourth order Runge-Kutta (RK4) scheme [Fablet et al., 2017].

Example: NN implementation of a 2D heterogeneous diffusion equation

For instance, the code used to generate the NN implementation of the heterogeneous 2D diffusion equation

$$\partial_t u = \nabla \cdot (\boldsymbol{\kappa} \nabla u)\,, \tag{3}$$

where $\boldsymbol{\kappa}(x,y) = [\kappa_{ij}(x,y)]_{(i,j)\in[1,2]\times[1,2]}$ is a field of 2×2 tensors ((x,y) are the spatial coordinates) and whose python implementation is detailed in Fig. 1-(a).

(a)

```python
from sympy import Function, symbols, Derivative
from pdenetgen import Eq, NNModelBuilder

# Defines the diffusion equation using sympy
t, x, y = symbols('t x y')
u = Function('u')(t,x,y)
kappa11 = Function('\\kappa_{11}')(x,y)
kappa12 = Function('\\kappa_{12}')(x,y)
kappa22 = Function('\\kappa_{22}')(x,y)

diffusion_2D = Eq(Derivative(u,t),
  Derivative(kappa11*Derivative(u,x)+
             kappa12*Derivative(u,y),x)+
  Derivative(kappa12*Derivative(u,x)+
             kappa22*Derivative(u,y),y)).doit()

# Defines the neural network code generator
diffusion_nn_builder = NNModelBuilder(diffusion_2D,
       class_name="NNDiffusion2DHeterogeneous")

# Renders the neural network code
exec(diffusion_nn_builder.code)

# Create a 2D Diffusion model
diffusion_model = NNDiffusion2DHeterogeneous()
```

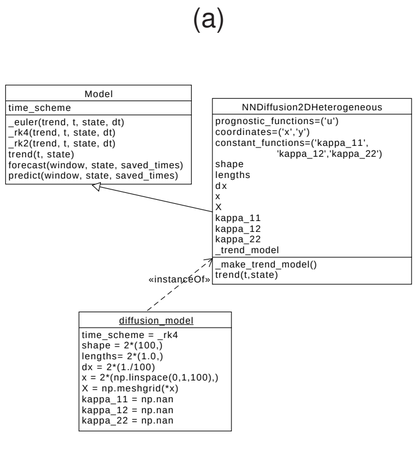
(b)

```python
# Example of computation of a derivative
kernel_Du_x_o1 = np.asarray([[0.0,-1/(2*self.dx[self.coordinates.index('x')]),0.0],
 [0.0,0.0,0.0],
 [0.0,1/(2*self.dx[self.coordinates.index('x')]),0.0]]).reshape((3, 3)+(1,1))
Du_x_o1 = DerivativeFactory((3, 3),kernel=kernel_Du_x_o1,name='Du_x_o1')(u)

# Computation of trend_u
mul_0 = keras.layers.multiply([Dkappa_11_x_o1,Du_x_o1],name='MulLayer_0')
mul_1 = keras.layers.multiply([Dkappa_12_x_o1,Du_y_o1],name='MulLayer_1')
mul_2 = keras.layers.multiply([Dkappa_12_y_o1,Du_x_o1],name='MulLayer_2')
mul_3 = keras.layers.multiply([Dkappa_22_y_o1,Du_y_o1],name='MulLayer_3')
mul_4 = keras.layers.multiply([Du_x_o2,kappa_11],name='MulLayer_4')
mul_5 = keras.layers.multiply([Du_y_o2,kappa_22],name='MulLayer_5')
mul_6 = keras.layers.multiply([Du_x_o1_y_o1,kappa_12],name='MulLayer_6')
sc_mul_0 = keras.layers.Lambda(lambda x: 2.0*x,name='ScalarMulLayer_0')(mul_6)
trend_u = keras.layers.add([mul_0,mul_1,mul_2,mul_3,mul_4,mul_5,sc_mul_0],name='AddLayer_0')
```

Figure: 1: (a) Neural Network generator for a heterogeneous 2D diffusion equation, and (b) sample of python code for the class which implements the diffusion equation Eq. 3 as a neural-network by using Keras;

(a)



(b)

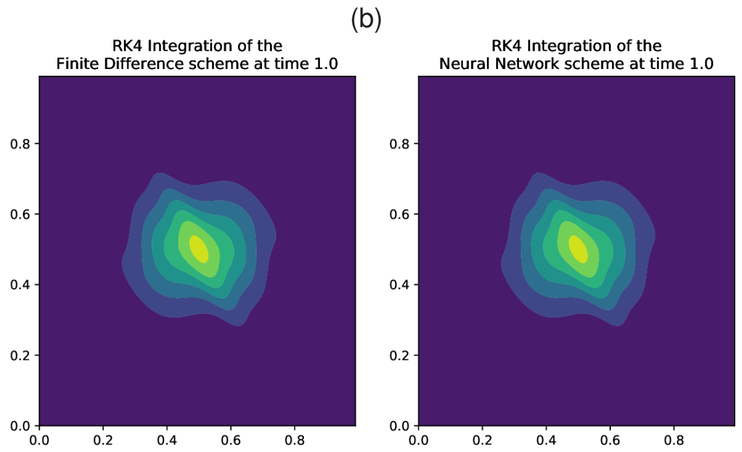


Figure: 2: (a) Part of the python code of the *NNDiffusion2DHeterogeneous* class which implements the diffusion equation Eq. (3) as a neural-network by using Keras (only one derivative is explicitly given, for the sake of simplicity); (b) Starting from a Dirac at the center of the domain, the diffusion equation Eq. (3) is integrated from 0 to 1 by using a fourth-order Runge-Kutta time scheme. The results obtained from the time integration of the finite-difference implementation Eq. (3) (left panel of b) and of the generated NN representation (right panel of b) are similar.

At the end, a python code is rendered from templates by using the *jinja2* package (see Fig. 1-(b)). The reason why templates are used is to facilitate the saving of the code in python modules and the modification of the code by the experimenter. Runtime computation of the class could be considered, but this is not implemented in the current version of *PDE-NetGen*. For the diffusion equation Eq. (3), when run, the code rendered from the `NNModelBuilder` class creates the `NNDiffusion2DHeterognous` class. Following the class diagram Fig. 2-(a), the `NNDiffusion2DHeterogeneous` class inherits from a `Model` class which implements the time evolution of an evolution dynamics by incorporating a time-scheme. Here several time-schemes are implemented, namely an explicit Euler scheme, a second and a fourth order Runge-Kutta scheme.
The numerical solutions of the finite difference and of the NN implementation, Fig. 2-(b) are identical

3. Estimation of an unknown physical term

As an illustration of the *PDE-NetGen* package, we consider a problem encountered in uncertainty prediction based on the parametric Kalman filter (PKF) [Pannekoucke et al., 2016, Pannekoucke et al., 2018]. We consider the dynamics

$$\begin{cases} \frac{\partial}{\partial t} u &= \kappa \frac{\partial^2}{\partial x^2} u - u \frac{\partial}{\partial x} u - \frac{\frac{\partial}{\partial x} \mathrm{V_u}}{2} \\ \frac{\partial}{\partial t} \mathrm{V_u} &= -\frac{\kappa \mathrm{V_u}}{\nu_{\mathrm{u,xx}}} + \kappa \frac{\partial^2}{\partial x^2} \mathrm{V_u} - \frac{\kappa \left(\frac{\partial}{\partial x} \mathrm{V_u}\right)^2}{2 \mathrm{V_u}} \\ & \quad - u \frac{\partial}{\partial x} \mathrm{V_u} - 2 \mathrm{V_u} \frac{\partial}{\partial x} u \\ \frac{\partial}{\partial t} \nu_{\mathrm{u,xx}} &= 4\kappa \, {\nu_{\mathrm{u,xx}}}^2 \, \mathbb{E}\left[\varepsilon_\mathrm{u} \frac{\partial^4}{\partial x^4} \varepsilon_\mathrm{u}\right] \\ & \quad - 3\kappa \frac{\partial^2}{\partial x^2} \nu_{\mathrm{u,xx}} - \kappa + \frac{6\kappa \left(\frac{\partial}{\partial x} \nu_{\mathrm{u,xx}}\right)^2}{\nu_{\mathrm{u,xx}}} \\ & \quad - \frac{2\kappa \, \nu_{\mathrm{u,xx}} \frac{\partial^2}{\partial x^2} \mathrm{V_u}}{\mathrm{V_u}} + \frac{\kappa \frac{\partial}{\partial x} \mathrm{V_u} \frac{\partial}{\partial x} \nu_{\mathrm{u,xx}}}{\mathrm{V_u}} + \\ & \quad \frac{2\kappa \, \nu_{\mathrm{u,xx}} \left(\frac{\partial}{\partial x} \mathrm{V_u}\right)^2}{{\mathrm{V_u}}^2} - u \frac{\partial}{\partial x} \nu_{\mathrm{u,xx}} + \\ & \quad 2 \, \nu_{\mathrm{u,xx}} \frac{\partial}{\partial x} u \end{cases} \tag{4}$$

that represent the uncertainty dynamics for the Burgers' equation $\partial_t u + u\partial_x u = \kappa \partial_x^2 u$, where $\mathbb{E}[\cdot]$ denotes the expectation operator.

Determination of unknown processes in uncertainty prediction

In this system of PDEs, the term $\mathbb{E}\left[\varepsilon_\mathrm{u} \frac{\partial^4}{\partial x^4} \varepsilon_\mathrm{u}\right]$ can not be determined from the known quantities u, $\mathrm{V_u}$ and $\nu_{\mathrm{u,xx}}$. This brings up a problem of closure, *i.e.* determining the unknown term as a function of the known quantities. A naive assumption would be to consider a zero closure (closure$(t,x) = 0$). However, while the tangent-linear evolution of the perturbations along the Burgers dynamics is stable, the dynamics of the diffusion coefficient $\nu_{\mathrm{u,xx}}$ would lead to unstable dynamics as the coefficient of the second order term $-3\kappa \frac{\partial^2}{\partial x^2} \nu_{\mathrm{u,xx}}$ is negative. This stresses further the importance of the unknown term to successfully predict the uncertainty.

Example of the implementation of a closure in PDE-NetGen: TrainableScalar

With *PDE-NetGen*, we can design a closure from the data. For the illustration we consider a candidate for the closure, given by

$$\text{closure}(t,x) \sim a \frac{\frac{\partial^2}{\partial x^2} \nu_{\mathrm{u,xx}}(t,x)}{{\nu_{\mathrm{u,xx}}}^2(t,x)} + b \frac{1}{{\nu_{\mathrm{u,xx}}}^2(t,x)} + c \frac{\left(\frac{\partial}{\partial x} \nu_{\mathrm{u,xx}}(t,x)\right)^2}{{\nu_{\mathrm{u,xx}}}^3(t,x)} \tag{5}$$

where (a,b,c) are unknown. An implementation in *PDE-NetGen* is given in Fig. 3.

(A)

Introduction of the closure ine the PKF dynamics

```python
from pdenetgen import TrainableScalar

# Set the closure by using TrainableScalar
a, b, c = [TrainableScalar(l) for l in 'abc']
closure_proposal = a*Derivative(nu,x,2)/nu**Integer(2)+b*1/nu**Integer(2)+\
                   c*Derivative(nu,x)**2/nu**Integer(3)
display(closure_proposal)
```

$$\frac{a\frac{\partial^2}{\partial x^2}\nu_{\mathrm{u,xx}}(t,x)}{{\nu_{\mathrm{u,xx}}}^2(t,x)} + \frac{b}{{\nu_{\mathrm{u,xx}}}^2(t,x)} + \frac{c\left(\frac{\partial}{\partial x}\nu_{\mathrm{u,xx}}(t,x)\right)^2}{{\nu_{\mathrm{u,xx}}}^3(t,x)}$$

```python
# Replace the closure(t,x) by the proposed closure
pkf_dynamics[2] = pkf_dynamics[2].subs(Function('closure')(t,x),closure_proposal)

# Generate the NN code leading to the ClosedPKFBurgers class.
exec(NNModelBuilder(pkf_dynamics,'ClosedPKFBurgers').code)
```

Sample of code generated to define the ClosedPKFBurgers class

```
    [..]
    pow_21 = keras.layers.multiply([div_17,div_17,] ,name='PowLayer_21')
    mul_28 = keras.layers.multiply([pow_21,Dnu_u_xx_x_o2],name='MulLayer_28')
    train_scalar_9 = TrainableScalarLayerFactory(input_shape=mul_28.shape, name='TrainableScalar_a',
                     init_value=0,use_bias=False,mean=0.0,stddev=1.0,seed=None,wl2=None)(mul_28)
                     #TrainableScalar name: 'a'
    add_8 = keras.layers.add([train_scalar_7,train_scalar_8,train_scalar_9],name='AddLayer_8')
    mul_26 = keras.layers.multiply([pow_17,add_8],name='MulLayer_26')
    [..]
```

(B)

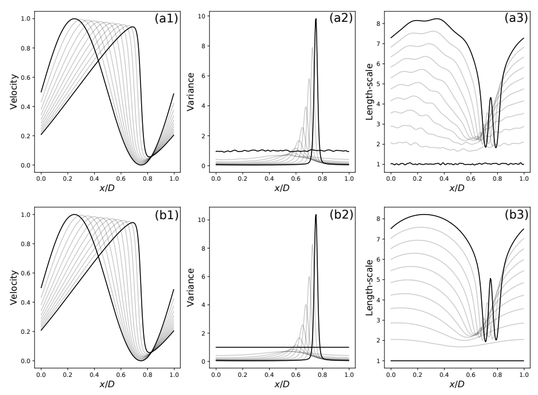


Figure: 3: (A) Implementation of the closure, by defining each unknown quantity as an instance of the class `TrainableScalar`, and the resulting generated NN code. This is a part of code available in the Jupyter notebook given as example in the package. (B) Uncertainty estimated from a large ensemble of 1000 members (a) with the expectation $\mathbb{E}[u]$ (a1), variance $\mathrm{V_u}$ (a2) and the length-scale (defined from the diffusion coefficient by $\sqrt{0.5\,\nu_{\mathrm{u,xx}}}$) (a3) ; and the uncertainty predicted from the PKF evolution equations closed from the data (b), where the same statistics are shown in (b1), (b2) and (b3). The fields are represented only for time $t = 0, 0.2, 0.4, 0.6, 0.8, 1$ *PDE-NetGen*.

A synthetic dataset involving 40000 samples has been created. To learn the trainable parameters (a,b,c), we minimize the one-step ahead prediction loss for the diffusion tensor ν_u. We use ADAM optimizer and a batch size of 32. Using an initial learning rate of 0.1, the training converges within 3 outer loops of 30 epochs with a geometrical decay of the learning rate by a factor of $1/10$ after each outer loop. The coefficients resulting from the training over 10 runs are $(a,b,c) = (0.93, 0.75, -1.80) \pm (5.1\ 10^{-5}, 3.6\ 10^{-4}, 2.7\ 10^{-4})$.
Fig. 3-(B) shows that the numerical solution based on the trained closure (a) is able to reproduce the main characteristics of the uncertainty dynamics diagnosed from an ensemble method (b).
There exists another strategy to learn the closure: in the NN implementation of the uncertainty dynamics it is possible to plug a neural network to be trained. This is not shown here but discussed in [Pannekoucke and Fablet, 2020].

4. Conclusions and Perspectives

We have introduced a neural network generator *PDE-NetGen*, which provides new means to bridge physical priors given as symbolic PDEs and learning-based NN frameworks. This package derives and implements a finite-difference version of a system of evolution equations, where the derivative operators are replaced by appropriate convolutional layers including the boundary conditions. The package has been developed in python using the symbolic mathematics library *sympy* and *keras*.
This work opens new avenues to make the most of existing physical knowledge and of recent advances in data-driven settings, and more particularly neural networks, for geophysical applications. This includes a wide range of applications, where such physically-consistent neural network frameworks could either lead to the reduction of the computational cost (e.g., GPU implementation embedded in deep learning frameworks) or provide new numerical tools to derive key operators (e.g., adjoint operator using automatic differentiation). Besides, these neural network representations also offer new means to complement known physics with the data-driven calibration of unknown terms. This is regarded as key to advance the state-of-the-art for the simulation, forecasting and reconstruction of geophysical dynamics through model-data-coupled frameworks.

5. References

Fablet, R., Ouala, S., and Herzet, C. (2017).
Bilinear residual neural network for the identification and forecasting of dynamical systems.
ArXiv.

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, Š., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., and Scopatz, A. (2017).
SymPy: symbolic computing in python.
PeerJ Computer Science, 3:e103.

Pannekoucke, O., Bocquet, M., and Ménard, R. (2018).
Parametric covariance dynamics for the nonlinear diffusive burgers' equation.
Nonlinear Processes in Geophysics, 2018:1–21.

Pannekoucke, O. and Fablet, R. (2020).
PDE-NetGen 1.0: from symbolic partial differential equation (PDE) representations of physical processes to trainable neural network representations.
Geoscientific Model Development, 13(7):3373–3382.

Pannekoucke, O., Ricci, S., Barthelemy, S., Ménard, R., and Thual, O. (2016).
Parametric kalman filter for chemical transport model.
Tellus, 68:31547.