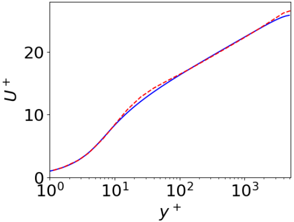
Using Physical Informed Neural Network (PINN) to Improve a $k - \omega$ Turbulence Model [2]

Lars Davidson, M2 Fluid Dynamics
Chalmers University of Technology
Gothenburg, Sweden

The Wilcox $k - \omega$ turbulence model reads [4]

$$
\begin{aligned}
\frac{\partial \bar{v}_i}{\partial x_i} &= 0 \\
\frac{\partial \bar{v}_i}{\partial t} + \frac{\partial \bar{v}_i \bar{v}_j}{\partial x_j} &= -\frac{1}{\rho}\frac{\partial \bar{p}}{\partial x_i} + \frac{\partial}{\partial x_j}\left[(\nu + \nu_t)\frac{\partial \bar{v}_i}{\partial x_j}\right] \\
\frac{\partial \bar{v}_j k}{\partial x_j} &= P^k + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right] - C_\mu k \omega \\
\frac{\partial \bar{v}_j \omega}{\partial x_j} &= C_{\omega_1}\frac{\omega}{k}P^k + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_\omega}\right)\frac{\partial \omega}{\partial x_j}\right] - C_{\omega 2}\omega^2 \\
P^k &= \nu_t\left(\frac{\partial \bar{v}_i}{\partial x_j} + \frac{\partial \bar{v}_j}{\partial x_i}\right)\frac{\partial \bar{v}_i}{\partial x_j}, \quad \nu_t = \frac{k}{\omega}
\end{aligned}
\tag{1}
$$

The standard coefficients are used, i.e. $C_{\omega 1} = 5/9$, $C_{\omega 2} = 3/40$, $\sigma_k = \sigma_\omega = 2$ and $C_\mu = 0.09$.

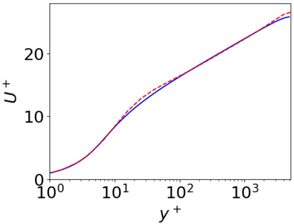


(A) Mean velocity.

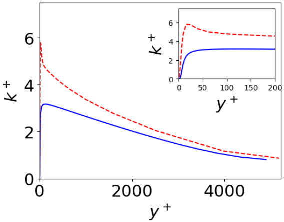
FIGURE: Fully-developed channel flow. Solid lines: $k-\omega$; dashed lines:DNS [3].

- The mean flow, shear stress (and hence the turbulent viscosity, ν_t) agree well

Fully-developed channel flow, $k-\omega$ model, $Re_\tau = 5\,200$



(A) Mean velocity.

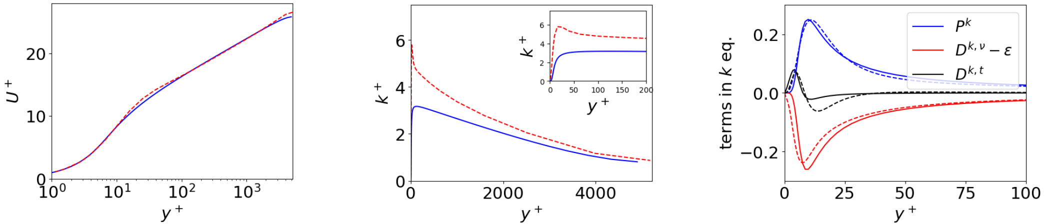


(B) Turb. kinetic energy.

FIGURE: Fully-developed channel flow. Solid lines: $k-\omega$; dashed lines:DNS [3].

- The mean flow, shear stress (and hence the turbulent viscosity, ν_t) agree well
- But not the turbulent, kinetic energy

FULLY-DEVELOPED CHANNEL FLOW, $k-\omega$ MODEL, $Re_\tau = 5\,200$



(A) Mean velocity. (B) Turb. kinetic energy. (C) Terms in k equation.

FIGURE: Fully-developed channel flow. Solid lines: $k-\omega$; dashed lines:DNS [3].

- The mean flow, shear stress (and hence the turbulent viscosity, ν_t) agree well
- But not the turbulent, kinetic energy
- It seems to be because the diffusion of k is poorly predicted

Find a New $\nu_{t,k}$

- Our ordinary differential equation reads in fully-developed channel flow

$$\frac{d}{dy}\left(\nu+\nu_{t,k}\frac{dk}{dy}\right)+P^k-\varepsilon=Q$$

Find a New $\nu_{t,k}$

- Our ordinary differential equation reads in fully-developed channel flow

$$\frac{d}{dy}\left(\nu + \nu_{t,k}\frac{dk}{dy}\right) + P^k - \varepsilon = Q$$

$$(\nu + \nu_{t,k})\,\frac{d^2 k_{DNS}}{dy^2} + \frac{dk_{DNS}}{dy}\frac{d\nu_{t,k}}{dy} + P^k_{DNS} - \varepsilon_{DNS} = Q \qquad (2)$$

where $\nu_{t,k}$ is the turbulent viscosity in the k_{DNS} equation and $Q = 0$.

- Our ordinary differential equation reads in fully-developed channel flow

$$\frac{d}{dy}\left(\nu + \nu_{t,k}\frac{dk}{dy}\right) + P^k - \varepsilon = Q$$

$$\left(\nu + \nu_{t,k}\right)\frac{d^2 k_{DNS}}{dy^2} + \frac{dk_{DNS}}{dy}\frac{d\nu_{t,k}}{dy} + P^k_{DNS} - \varepsilon_{DNS} = Q \qquad (2)$$

  where $\nu_{t,k}$ is the turbulent viscosity in the k_{DNS} equation and $Q = 0$.
- $\nu_{t,k}$ is the unknown

FIND A NEW $\nu_{t,k}$

- Our ordinary differential equation reads in fully-developed channel flow

$$\frac{d}{dy}\left(\nu+\nu_{t,k}\frac{dk}{dy}\right)+P^k-\varepsilon=Q$$

$$\left(\nu+\nu_{t,k}\right)\frac{d^2k_{DNS}}{dy^2}+\frac{dk_{DNS}}{dy}\frac{d\nu_{t,k}}{dy}+P^k_{DNS}-\varepsilon_{DNS}=Q \tag{2}$$

  where $\nu_{t,k}$ is the turbulent viscosity in the k_{DNS} equation and $Q=0$.
- $\nu_{t,k}$ is the unknown
- k_{DNS}, P^k_{DNS} and ε_{DNS} are known (taken from DNS),

FIND A NEW $\nu_{t,k}$

- Our ordinary differential equation reads in fully-developed channel flow

$$\frac{d}{dy}\left(\nu+\nu_{t,k}\frac{dk}{dy}\right)+P^k-\varepsilon=Q$$

$$(\nu+\nu_{t,k})\frac{d^2k_{DNS}}{dy^2}+\frac{dk_{DNS}}{dy}\frac{d\nu_{t,k}}{dy}+P^k_{DNS}-\varepsilon_{DNS}=Q \tag{2}$$

  where $\nu_{t,k}$ is the turbulent viscosity in the k_{DNS} equation and $Q=0$.
- $\nu_{t,k}$ is the unknown
- k_{DNS}, P^k_{DNS} and ε_{DNS} are known (taken from DNS),
- First I tried to use the finite volume method

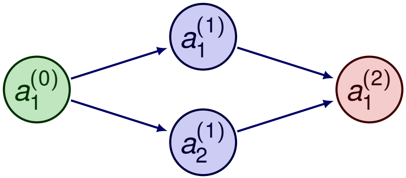
FIND A NEW $\nu_{t,k}$

- Our ordinary differential equation reads in fully-developed channel flow

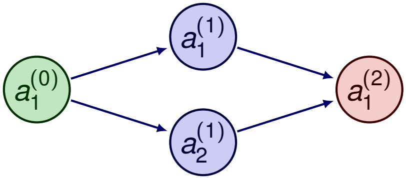
$$\frac{d}{dy}\left(\nu + \nu_{t,k}\frac{dk}{dy}\right) + P^k - \varepsilon = Q$$

$$\left(\nu + \nu_{t,k}\right)\frac{d^2 k_{DNS}}{dy^2} + \frac{dk_{DNS}}{dy}\frac{d\nu_{t,k}}{dy} + P^k_{DNS} - \varepsilon_{DNS} = Q \qquad (2)$$

  where $\nu_{t,k}$ is the turbulent viscosity in the k_{DNS} equation and $Q = 0$.
- $\nu_{t,k}$ is the unknown
- k_{DNS}, P^k_{DNS} and ε_{DNS} are known (taken from DNS),
- First I tried to use the finite volume method
- $\nu_{t,k} = \nu_{t,NN}$ in Eq. 2, will be predicted by PINN while minimizing the error Q^2.

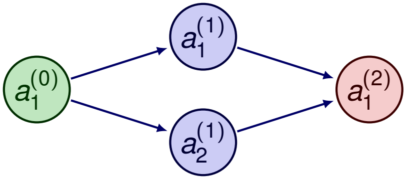


- I create a NN that finds a damping function, $Y \equiv f$, as a function of input $X \equiv y^+$
- 1 input ($X = a_1^{(0)}$), 1 hidden layer with 2 neurons ($a_1^{(1)}, a_2^{(1)}$) and 1 output ($Y = a_1^{(2)}$)

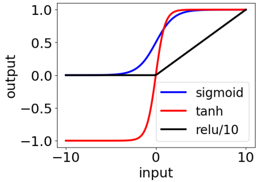


- I create a NN that finds a damping function, $Y \equiv f$, as a function of input $X \equiv y^+$
- 1 input ($X = a_1^{(0)}$), 1 hidden layer with 2 neurons ($a_1^{(1)}, a_2^{(1)}$) and 1 output ($Y = a_1^{(2)}$)

```
class NN(nn.Module):
 def super-__init__(self):
  self.layer_1=nn.Linear(1, 2) # Connection 0-1
  self.layer_2=nn.Linear(2, 1) # Connection 1-2
 def forward(self, x):
  y = torch.nn.functional.sigmoid(self.layer_1(x)) # a_1^{(1)}, a_2^{(1)}, hidden-layer
  output = torch.nn.functional.sigmoid(self.layer_2(y)) # a_1^{(2)}, output-layer
```

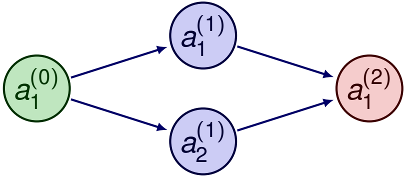


- I create a NN that finds a damping function, $Y \equiv f$, as a function of input $X \equiv y^+$
- 1 input ($X = a_1^{(0)}$), 1 hidden layer with 2 neurons ($a_1^{(1)}, a_2^{(1)}$) and 1 output ($Y = a_1^{(2)}$)




```
class NN(nn.Module):
 def super-__init__(self):
  self.layer_1=nn.Linear(1, 2) # Connection 0-1
  self.layer_2=nn.Linear(2, 1) # Connection 1-2
 def forward(self, x):
  y = torch.nn.functional.sigmoid(self.layer_1(x)) # a_1^{(1)}, a_2^{(1)}, hidden-layer
  output = torch.nn.functional.sigmoid(self.layer_2(y)) # a_1^{(2)}, output-layer
```

NEURAL NETWORK (NN). FORWARD



Activation 1: $$a_1^{(1)} = s_1^{(1)}\left(w_1^{(0)}a_1^{(0)} + b_1^{(0)}\right)$$

Activation 2: $$a_2^{(1)} = s_2^{(1)}\left(w_2^{(0)}a_1^{(0)} + b_2^{(0)}\right)$$

Output: $$a_1^{(2)} = s_1^{(2)}\left(w_1^{(1)}a_1^{(1)} + b_1^{(1)} + w_2^{(1)}a_2^{(1)} + b_2^{(1)}\right) \equiv Y$$

- s is an activation function (`linear`, `sigmoid`, `tanh`, ...)

The Python code for the simple NN model is given in the listing below

```
#  initiate the NN model
model = NN()
# define input, X
X=np.zeros(nj,1))
X[:,0] = scaler_yplus.fit_transform(yplus)[:,0]
# define output, Y (f is known)
Y = f
# Training loop
for epoch in range(max_no_epoch):
# Compute prediction and loss, L
    o = model(X) #prediction
    L = loss_fn(o, Y)  # L=|o-Y|_2
    L.backward()
```

- `loss.backward()` computes $dL/dw_1, dL/db_1, dL/ds_1, \ldots$
- They are used to get new improved $w_1, b_1, \ldots$

$$\left(\nu + \nu_{t,NN}\right) \frac{d^2 k_{DNS}}{dy^2} + \frac{dk_{DNS}}{dy}\frac{d\nu_{t,NN}}{dy} + P^k_{DNS} - \varepsilon_{DNS} = Q \qquad (3)$$

$$(\nu + \nu_{t,NN}) \frac{d^2 k_{DNS}}{dy^2} + \frac{dk_{DNS}}{dy}\frac{d\nu_{t,NN}}{dy} + P^k_{DNS} - \varepsilon_{DNS} = Q \tag{3}$$

- The loss function, `loss_fn`, on Slide 7 is replaced with Eq. 3. Python code:

$$\left(\nu + \nu_{t,NN}\right) \frac{d^2 k_{DNS}}{dy^2} + \frac{dk_{DNS}}{dy}\frac{d\nu_{t,NN}}{dy} + P^k_{DNS} - \varepsilon_{DNS} = Q \tag{3}$$

- The loss function, `loss_fn`, on Slide 7 is replaced with Eq. 3. Python code:

```
def ODE(y, nut):
   nut_y = torch.autograd(nut, y, torch.ones(x.size()[0], 1,),create_graph=True)[0]
# Differential equation loss
   ODE_loss = (nu+nut)*k_yy + k_y*nut_y + Pk - eps
   ODE_loss = torch.sum(ODE_loss ** 2)
# b.c. loss
   BC_loss = (nut[0] - nut_0) ** 2
   return ODE_loss, BC_loss
loss_ODE, loss_bc = ODE(y,nut)
```

$$\left(\nu + \nu_{t,NN}\right) \frac{d^2 k_{DNS}}{dy^2} + \frac{dk_{DNS}}{dy} \frac{d\nu_{t,NN}}{dy} + P^k_{DNS} - \varepsilon_{DNS} = Q \tag{3}$$

- The loss function, `loss_fn`, on Slide 7 is replaced with Eq. 3. Python code:

```
def ODE(y, nut):
   nut_y = torch.autograd(nut, y, torch.ones(x.size()[0], 1,),create_graph=True)[0]
# Differential equation loss
   ODE_loss = (nu+nut)*k_yy + k_y*nut_y + Pk - eps
   ODE_loss = torch.sum(ODE_loss ** 2)
# b.c. loss
   BC_loss = (nut[0] - nut_0) ** 2
   return ODE_loss, BC_loss
loss_ODE, loss_bc = ODE(y,nut)
```

- `nut` is the unknown

$$\left(\nu + \nu_{t,NN}\right) \frac{d^2 k_{DNS}}{dy^2} + \frac{dk_{DNS}}{dy}\frac{d\nu_{t,NN}}{dy} + P^k_{DNS} - \varepsilon_{DNS} = Q \tag{3}$$

- The loss function, `loss_fn`, on Slide 7 is replaced with Eq. 3. Python code:

```
def ODE(y, nut):
   nut_y = torch.autograd(nut, y, torch.ones(x.size()[0], 1,),create_graph=True)[0]
# Differential equation loss
   ODE_loss = (nu+nut)*k_yy + k_y*nut_y + Pk - eps
   ODE_loss = torch.sum(ODE_loss ** 2)
# b.c. loss
   BC_loss = (nut[0] - nut_0) ** 2
   return ODE_loss, BC_loss
loss_ODE, loss_bc = ODE(y,nut)
```

- `nut` is the unknown
- `k_y`, for example, is dk_{DNS}/dy.

$$(\nu + \nu_{t,NN}) \frac{d^2 k_{DNS}}{dy^2} + \frac{dk_{DNS}}{dy} \frac{d\nu_{t,NN}}{dy} + P^k_{DNS} - \varepsilon_{DNS} = Q \tag{3}$$

- The loss function, `loss_fn`, on Slide 7 is replaced with Eq. 3. Python code:

```
def ODE(y, nut):
   nut_y = torch.autograd(nut, y, torch.ones(x.size()[0], 1,),create_graph=True)[0]
# Differential equation loss
   ODE_loss = (nu+nut)*k_yy + k_y*nut_y + Pk - eps
   ODE_loss = torch.sum(ODE_loss ** 2)
# b.c. loss
   BC_loss = (nut[0] - nut_0) ** 2
   return ODE_loss, BC_loss
loss_ODE, loss_bc = ODE(y,nut)
```

- `nut` is the unknown
- `k_y`, for example, is dk_{DNS}/dy.
- Note that `k_y` and `k_yy` are known and constant (DNS).

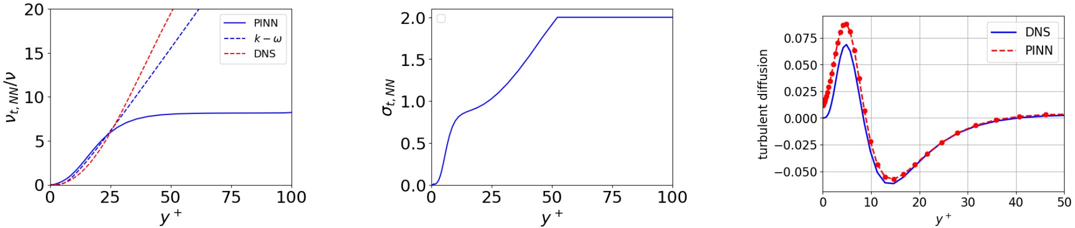
$$\left(\nu + \nu_{t,NN}\right) \frac{d^2 k_{DNS}}{dy^2} + \frac{dk_{DNS}}{dy}\frac{d\nu_{t,NN}}{dy} + P^k_{DNS} - \varepsilon_{DNS} = Q \tag{3}$$

- The loss function, `loss_fn`, on Slide 7 is replaced with Eq. 3. Python code:

```
def ODE(y, nut):
   nut_y = torch.autograd(nut, y, torch.ones(x.size()[0], 1,),create_graph=True)[0]
# Differential equation loss
   ODE_loss = (nu+nut)*k_yy + k_y*nut_y + Pk - eps
   ODE_loss = torch.sum(ODE_loss ** 2)
# b.c. loss
   BC_loss = (nut[0] - nut_0) ** 2
   return ODE_loss, BC_loss
loss_ODE, loss_bc = ODE(y,nut)
```

- `nut` is the unknown
- `k_y`, for example, is dk_{DNS}/dy.
- Note that `k_y` and `k_yy` are known and constant (DNS).
- There are two losses, `ODE_loss` and `BC_loss`

SOLVING EQ. 2 WITH PINN.



(A) Turbulent viscosity. (B) Prandtl number. (C) Turbulent diffusion.

FIGURE: k equation.

- Fully-developed flow in half a channel at $Re_\tau = 5\,200$.
- $\sigma_{t,NN} = \nu_t/\nu_{t,NN}$ (ν_t is the turbulent viscosity predicted by the Wilcox $k-\omega$ model)
- $\sigma_{t,NN}$ is limited to 2 (same as σ_k in the Wilcox $k-\omega$ model)

- The Python finite volume code pyCALC-RANS [1] is used.

- The Python finite volume code pyCALC-RANS [1] is used.
- Fully vectorized (i.e. no `for` loops).

- The Python finite volume code pyCALC-RANS [1] is used.
- Fully vectorized (i.e. no `for` loops).
- SIMPLEC and Wilcox $k-\omega$ model

CFD SOLVER

- The Python finite volume code pyCALC-RANS [1] is used.
- Fully vectorized (i.e. no `for` loops).
- SIMPLEC and Wilcox $k - \omega$ model
- Discretization: Hybrid first-order upwind/second-order central differencing

- The Python finite volume code pyCALC-RANS [1] is used.
- Fully vectorized (i.e. no `for` loops).
- SIMPLEC and Wilcox $k - \omega$ model
- Discretization: Hybrid first-order upwind/second-order central differencing
- The discretized equations are solved with Python sparse matrix solvers.

The equation below is solved is using **pyCALC-RANS**

$$\frac{d}{dy}\left(\nu + \nu_{t,NN}\frac{dk}{dy}\right) + P^k_{DNS} - \varepsilon_{DNS} = 0$$

where $\nu_{t,NN}$ is known (given by PINN) and P^k_{DNS} and ε_{DNS} are taken from DNS.

The equation below is solved is using **pyCALC-RANS**

$$\frac{d}{dy}\left(\nu + \nu_{t,NN}\frac{dk}{dy}\right) + P^k_{DNS} - \varepsilon_{DNS} = 0$$

where $\nu_{t,NN}$ is known (given by PINN) and P^k_{DNS} and ε_{DNS} are taken from DNS.

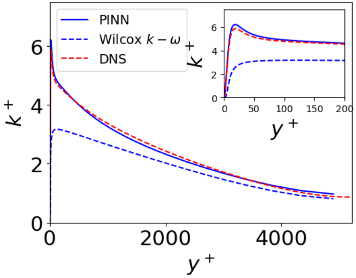


FIGURE: Turbulent kinetic energy.

FIND C_k AND $C_{\omega 2}$

- I have modified the turbulent Prandtl number in the k equation so that I get correct (larger) k

FIND C_k AND $C_{\omega 2}$

- I have modified the turbulent Prandtl number in the k equation so that I get correct (larger) k
- Recall that the standard $k-\omega$ gives correct $\nu_t = k/\omega = k_{DNS}/\omega_{DNS}$

FIND C_k AND $C_{\omega 2}$

- I have modified the turbulent Prandtl number in the k equation so that I get correct (larger) k
- Recall that the standard $k-\omega$ gives correct $\nu_t = k/\omega = k_{DNS}/\omega_{DNS}$
- I must predict a correct $\varepsilon = \varepsilon_{DNS}$, i.e.

Find C_k and $C_{\omega 2}$

- I have modified the turbulent Prandtl number in the k equation so that I get correct (larger) k
- Recall that the standard $k-\omega$ gives correct $\nu_t = k/\omega = k_{DNS}/\omega_{DNS}$
- I must predict a correct $\varepsilon = \varepsilon_{DNS}$, i.e.

$$\frac{d}{dy}\left(\frac{\nu_t}{\sigma_{t,NN}}\frac{dk_{DNS}}{dy}\right) + P^k_{DNS} - \underbrace{C_k C_\mu k_{DNS}\omega_{DNS}}_{\varepsilon_{DNS}} = 0$$

Find C_k and $C_{\omega 2}$

- I have modified the turbulent Prandtl number in the k equation so that I get correct (larger) k
- Recall that the standard $k-\omega$ gives correct $\nu_t = k/\omega = k_{DNS}/\omega_{DNS}$
- I must predict a correct $\varepsilon = \varepsilon_{DNS}$, i.e.

$$\frac{d}{dy}\left(\frac{\nu_t}{\sigma_{t,NN}}\frac{dk_{DNS}}{dy}\right) + P^k_{DNS} - \underbrace{C_k C_\mu k_{DNS}\omega_{DNS}}_{\varepsilon_{DNS}} = 0$$

- Finally, the ω equation in the new $k-\omega$ model must predict $\omega = \omega_{DNS}$

$$\frac{d}{dy}\left(\frac{\nu_t}{\sigma_\omega}\frac{d\omega_{DNS}}{dy}\right) + C_{\omega 1}\frac{\omega_{DNS}}{k_{DNS}}P^k_{DNS} - C_{\omega 2}\omega^2_{DNS} = 0$$

PLOT C_k AND $C_{\omega 2}$

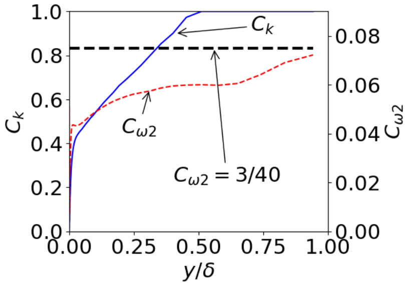
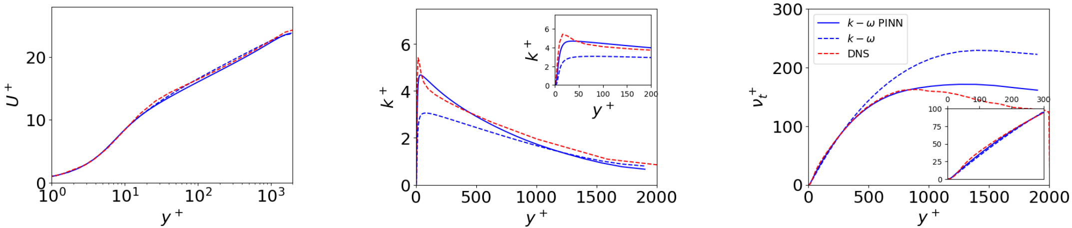


FIGURE: C_k and $C_{\omega 2}$ vs. y/δ.

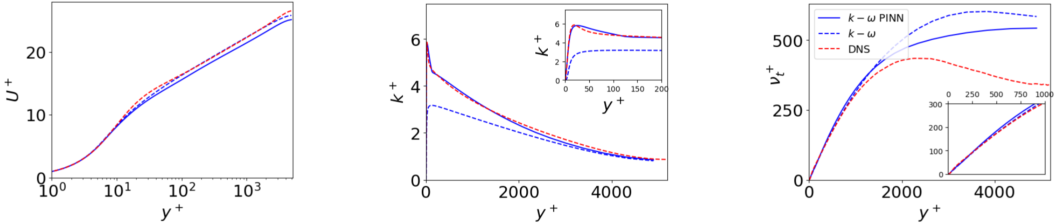
RESULTS. CHANNEL FLOW. $Re_\tau = 2\,000$



(A) Velocity. (B) Turb. kinetic energy. (C) Turbulent viscosity.

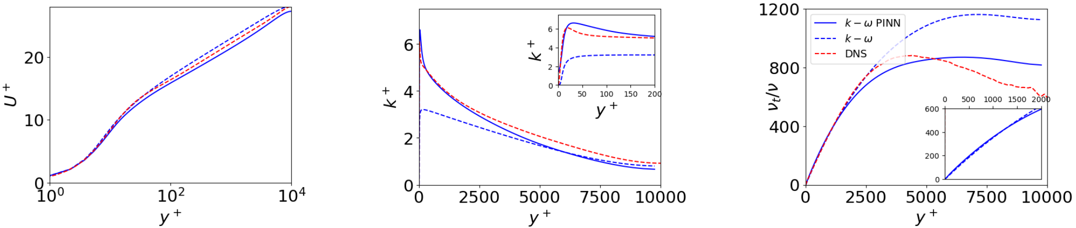
FIGURE: Fully-developed channel flow. $Re_\tau = 2\,000$.

RESULTS. CHANNEL FLOW. $Re_\tau = 5\,200$



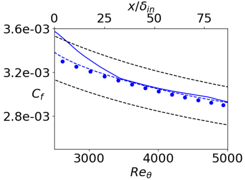
(A) Velocity. (B) Turb. kinetic energy. (C) Turbulent viscosity.

FIGURE: Fully-developed channel flow. $Re_\tau = 5\,200$.

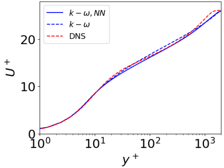


(A) Velocity. (B) Turb. kinetic energy. (C) Turbulent viscosity.

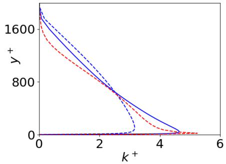
FIGURE: Fully-developed channel flow. $Re_\tau = 10\,000$.



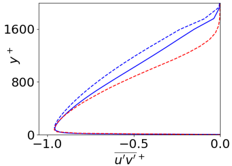
(A) Skin friction.



(B) Velocity.



(C) Turb. kinetic energy.



(D) Turbulent shear stress.

FIGURE: Flat-plate boundary layer. Profiles at $Re_\theta = 4\,500$.

- Inlet profiles from a pre-cursor RANS at $Re_\theta = 2\,500$

- I have made σ_k, C_K and $C_{\omega 2}$ functions of y/δ

- I have made σ_k, C_K and $C_{\omega 2}$ functions of y/δ
- Hence, the current formulation of the model is not applicable to re-circulating flow

- I have made σ_k, C_K and $C_{\omega 2}$ functions of y/δ
- Hence, the current formulation of the model is not applicable to re-circulating flow
- Using Neural Network (NN), I've tried to make them functions of different input parameters such a P_k/ε, P_k^+, $\nu_t/(yu_\tau)$, ...

- I have made σ_k, C_K and $C_{\omega 2}$ functions of y/δ
- Hence, the current formulation of the model is not applicable to re-circulating flow
- Using Neural Network (NN), I've tried to make them functions of different input parameters such a P_k/ε, P_k^+, $\nu_t/(yu_\tau)$, ...
- Finally, I found a good combination input parameters: $\overline{u'v'}/u_\tau^2$ and $\nu_t/(yu_\tau)$ (not shown)

- The $k-\omega$ model has been modified using PINN so that it accurately predicts the turbulent kinetic energy

CONCLUDING REMARKS

- The $k - \omega$ model has been modified using PINN so that it accurately predicts the turbulent kinetic energy
- I have modified σ_k and $C_{\omega 2}$ and introduced a new C_k

- The $k-\omega$ model has been modified using PINN so that it accurately predicts the turbulent kinetic energy
- I have modified σ_k and $C_{\omega 2}$ and introduced a new C_k
- It works well for channel flow and flat-plate boundary layer

- The $k-\omega$ model has been modified using PINN so that it accurately predicts the turbulent kinetic energy
- I have modified σ_k and $C_{\omega 2}$ and introduced a new C_k
- It works well for channel flow and flat-plate boundary layer
- Using NN, σ_k, $C_{\omega 2}$ and C_k are made are functions of $\overline{u'v'}/u_\tau^2$ and $\nu_t/(yu_\tau)$

Concluding Remarks

- The $k-\omega$ model has been modified using PINN so that it accurately predicts the turbulent kinetic energy
- I have modified σ_k and $C_{\omega 2}$ and introduced a new C_k
- It works well for channel flow and flat-plate boundary layer
- Using NN, σ_k, $C_{\omega 2}$ and C_k are made are functions of $\overline{u'v'}/u_\tau^2$ and $\nu_t/(yu_\tau)$
- You can download the ETMM15 paper, pyCALC-RANS and PINN scripts here or Google pyCALC-RANS PINN

- Neural Network and PINN in Python.
  - Good YouTube lectures: ”3Blue1Brown: But what is a neural network”; ”3Blue1Brown: gradient descent, how neural networks learn”; ”3Blue1Brown: backpropagation, intuitively”; ”3Blue1Brown: backpropagation, calculus”; ”Sebastian Lague: how to create a neural network”.

Download

- Download paper, Python scripts and CFD codes

References

[1] L. Davidson. pyCALC-RANS: a 2D Python code for RANS. Division of Fluid Dynamics, Dept. of Mechanics and Maritime Sciences, Chalmers University of Technology, Gothenburg
Download the code here, 2021.

[2] L. Davidson. Using physical informed neural network (PINN) to improve a k-omega turbulence model. In *15th International ERCOFTAC Symposium on Engineering Turbulence Modelling and Measurements (ETMM15), Dubrovnik on 22-24 September*, 2025.

[3] M. Lee and R. D. Moser. Direct numerical simulation of turbulent channel flow up to $Re_\tau \approx 5200$. *Journal of Fluid Mechanics*, 774:395–415, 2015.

[4] D. C. Wilcox. Reassessment of the scale-determining equation. *AIAA Journal*, 26(11):1299–1310, 1988.