Reports

By

Wentao Gong

2019.04~2021.02

Hokkaido University

Content

1D RANS Turbulent Modeling

This RANS simulation of fully developed turbulent channel flows including:
(a) Simplify the standard k − ε model to one-dimensional ODE equations for the fully developed channel flows.
(b) Perform numerical simulations of the fully developed channel flows based on the standard k − ε RANS model with near wall treatments .
(c) Compare the numerical solutions U +, k+, and ε+ with the DNS results (available in http://turbulence.ices.utexas.edu/).

Turbulent Kinetic Energy

To derive the turbulent kinetic energy equation, continuity equation and Navier-Stokes equation are used and shown on below formula.

$$\frac{\partial \widetilde{u}_i}{\partial x_i} = \frac{\partial u_i}{\partial x_i} = 0$$

$$\frac{\partial \widetilde{u}_i}{\partial t} + \widetilde{u}_j \frac{\partial \widetilde{u}_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial \widetilde{p}}{\partial x_i} + 2\nu \frac{\partial \widetilde{S}_{ij}}{\partial x_j}$$

where instaneous variables u, p are decomposed to mean parts (U_i, P) and fluctuation parts (u_i, p)

$\widetilde{u}_i = U_i + u_i$

$\widetilde{p} = P + p$

and define TKE(Turbulent Kinetic Energy) as $k = \frac{\overline{u_i}\,\overline{u_i}}{2}$ where the turbulent velocity component is the difference between the instantaneous and the average velocity $u' = \widetilde{u}_i - U$,whose mean and variance velocity are $\overline{u'} = \frac{1}{T}\int_0^T (\widetilde{u}(t) - U)dt = 0$ and $(\overline{u'})^2 = \frac{1}{T}\int_0^T (\widetilde{u}(t) - U)^2 dt \geq 0$ Respectively.

Firstly multiply fluctuation velocity u_i to each side of Navier-Stokes equation and second ensemble average then NSE are modified to below form.

$$\left\langle u_i \frac{\partial}{\partial t}\left(U_i + u_i\right) + u_i(U_i + u_i)\frac{\partial}{\partial x_j}\left(U_i + u_i\right) = -\frac{u_i}{\rho}\frac{\partial}{\partial x_j}\left(P + p\right) + 2\nu u_i \frac{\partial}{\partial x_j}\left(S_{ij} + s_{ij}\right)\right\rangle$$

Each term is simplified by below calculation. First time derivation term :

$$\left\langle u_i \frac{\partial}{\partial t}\left(U_i + u_i\right)\right\rangle = u_i \frac{\partial u_i}{\partial t} = \frac{\partial}{\partial t}\left(\frac{\overline{u_i u_i}}{2}\right) = \frac{\partial k}{\partial t}$$

Second non-linear advection term :

$$\begin{aligned}\left\langle u_i(U_i + u_i)\frac{\partial}{\partial x_j}\left(U_i + u_i\right)\right\rangle &= \overline{u_i U_i \frac{\partial U_i}{\partial x_j}} + \overline{u_i u_j \frac{\partial U_i}{\partial x_j}} + \overline{u_i U_j \frac{\partial u_i}{\partial x_j}} + \overline{u_i u_j \frac{\partial u_i}{\partial x_j}} \\ &= \overline{u_i u_j}\frac{\partial U_i}{\partial x_j} + U_j \frac{\partial}{\partial x_j}\left(\frac{\overline{u_i u_i}}{2}\right) + \frac{1}{2}\frac{\partial}{\partial x_j}\left(\overline{u_i u_i u_j}\right)\end{aligned}$$

Third pressure term :

$$\left\langle -\frac{u_i}{\rho}\frac{\partial}{\partial x_j}\left(P + p\right)\right\rangle = -\overline{\frac{u_i}{\rho}\frac{\partial p}{\partial x_i}} = -\frac{1}{\rho}\frac{\partial}{\partial x_i}\left(\overline{p u_i}\right)$$

Last viscosity term :

$$\begin{aligned}\left\langle 2\nu u_i \frac{\partial}{\partial x_j}\left(S_{ij} + s_{ij}\right)\right\rangle &= \left\langle 2\nu u_i \frac{\partial s_{ij}}{\partial x_j}\right\rangle \\ &= \left\langle 2\nu \frac{\partial}{\partial x_j}\left(u_i s_i j\right) - 2\nu \frac{\partial u_i}{\partial x_j} s_{ij}\right\rangle \\ &= \frac{\partial}{\partial x_j}\left(2\nu \overline{u_i s_{ij}}\right) - 2\nu \overline{s_{ij} s_{ij}}\end{aligned}$$

Sum up all the terms and put them together then, below equation is followed, which is same as turbulent kinetic energy equation.

TKE can be produced by fluid shear, friction or buoyancy, or through external forcing at low-frequency eddy scales(integral scale). Turbulence kinetic energy is then transferred down the turbulence energy cascade, and is dissipated by viscous forces at the Kolmogorov scale. This process of production, transport and dissipation can be expressed as:

$$\frac{\partial}{\partial t}\left(\frac{\overline{u_i u_i}}{2}\right) + U_j \frac{\partial}{\partial x_j}\left(\frac{\overline{u_i u_i}}{2}\right) = -\overline{u_i u_j}\frac{\partial U}{\partial x_j} - 2\nu \overline{s_{ij} s_{ij}} - \frac{\partial}{\partial x_j}\left(\frac{1}{2}\overline{u_i u_i u_j} + \frac{\overline{u_i p}}{\rho} - 2\nu \overline{u_i s_{ij}}\right)$$

$$\therefore \quad \frac{Dk}{Dt} + \nabla \cdot T' = \mathcal{P} - \epsilon$$

where

- Dk/Dt is the mean-flow material derivative of TKE;
- $\nabla \cdot T'$ is the turbulence transport of TKE;
- P is the production of TKE, and
- ε is the TKE dissipation.

$$k = \overline{u_i u_i}/2,\ p = -\overline{u_i u_j} S_{ij},\ \varepsilon = 2\nu \overline{s_{i,j} s_{i,j}},\ T_i = \frac{1}{2}\overline{u_i u_i u_j} + u_i p/\rho - 2\nu \overline{u_i s_{i,j}}$$

RANS

Governing Equations: Standard k-epsilon RANS Model

Standard K-epsilon RANS model consist of 4 equations
Continuity equation

$$\frac{\partial U_i}{\partial x_i} = 0$$

Reynolds Averaged Navier-Stokes equation

$$\frac{\partial U_i}{\partial t} + U_j \frac{\partial U_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial P}{\partial x_i} + \nu \frac{\partial^2 U_i}{\partial x_j \partial x_j} - \frac{\partial}{\partial x_j}\left(\overline{u_i}\,\overline{u_j}\right)$$

k equation

$$\frac{\partial k}{\partial t} + U_j \frac{\partial k}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\left(\nu + \frac{\nu_T}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right) + p - \varepsilon$$

ε equation

$$\frac{\partial \varepsilon}{\partial t} + U_j \frac{\partial \varepsilon}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\left(\nu + \frac{\nu_T}{\sigma_\varepsilon}\right)\frac{\partial \varepsilon}{\partial x_j}\right) + C_{\varepsilon 1}\frac{P_\varepsilon}{k} - C_{\varepsilon 2}\frac{\varepsilon^2}{k}$$

Eddy-viscosity model

$$-\overline{u_i u_j} = \nu_T \frac{\partial U_i}{\partial x_j}$$

and turbulent viscosity is defined as below formula in k-epsilon models

$$\nu_T = C_\mu \frac{k^2}{\varepsilon}$$

Fully-developed channel flows case

At fully developed channel flows case, V=W by continuity equation and it is steady state and homogeneous at x direction except pressure term, which are expressed as

$\frac{\partial}{\partial t} = 0$ Steady state

$\frac{\partial}{\partial z} = 0$ Homogeneous condition

$\frac{\partial}{\partial z} = 0$ Fully developed condition

V=W=0 Continuity equation and no-slip boundary condition

Using above 4 conditions, governing equations are simplified to 1D as below formulas in k-epsilon RANS model.

$$\begin{cases} -\overline{uv} = \nu_T \dfrac{\partial U}{\partial y} \\ -\dfrac{1}{\rho}\dfrac{dP_0}{dx} + \nu\dfrac{\partial^2 U}{\partial y^2} - \dfrac{\partial}{\partial y}\left(\overline{uv}\right) = 0 \\ \dfrac{\partial}{\partial y}\left(\left(\nu + \dfrac{\nu_T}{\sigma_k}\right)\dfrac{\partial k}{\partial y}\right) - \overline{uv}\dfrac{\partial U}{\partial y} - \epsilon = 0 \\ \dfrac{\partial}{\partial y}\left(\left(\nu + \dfrac{\nu_T}{\sigma_\epsilon}\right)\dfrac{\partial \epsilon}{\partial y}\right) + \dfrac{C_{\epsilon 1}}{k}\left(-\overline{uv}\dfrac{\partial U}{\partial y}\right)\epsilon - \dfrac{C_{\epsilon 2}}{k}\epsilon^2 = 0 \end{cases}$$

Where $C_\mu = 0.09, C_{\varepsilon 1} = 1.45, C_{\varepsilon 2} = 1.9, \sigma_k = 1.0, \sigma_\varepsilon = 1.3$

Finite difference method

$$\nu_{Ti} = C_\mu \frac{k^2{}_i}{\varepsilon}$$

$$P_i = \nu_{Ti}\left(\frac{U_{i+1} - U_{i-1}}{2\Delta y}\right)^2$$

$$\nu\frac{\partial^2 U}{\partial y^2} - \frac{\partial}{\partial y}\left(\overline{uv}\right) = \frac{1}{\rho}\frac{dP_0}{dx}$$

$$\Rightarrow \nu\left(\frac{U_{i+1} - 2U_i + U_{i-1}}{\Delta y^2}\right) + \frac{\partial\left(\nu_T \dfrac{\partial U}{\partial y}\right)}{\partial y}$$

$$= \nu\left(\frac{U_{i+1} - 2U_i + U_{i-1}}{\Delta y^2}\right) + \frac{1}{\Delta y}\left[\frac{\nu_{T\,i+1} + \nu_{T\,i}}{2}\left(\frac{U_{i+1} - U_i}{\Delta y}\right) - \frac{\nu_{T\,i} + \nu_{T\,i-1}}{2}\left(\frac{U_i - U_{i-1}}{\Delta y}\right)\right]$$

$$= \frac{1}{2\Delta y^2}\left[\left(2\nu + \nu_{T\,i+1} + \nu_{T\,i}\right)U_{i+1} + \left(-4\nu - \nu_{T\,i+1} - 2\nu_{Ti} - \nu_{Ti-1}\right)U_i + \left(2\nu + \nu_{T\,i} + \nu_{T\,i-1}\right)U_{i-1}\right]$$

$$\Rightarrow \left[\left(2\nu + \nu_{T\,i+1} + \nu_{T\,i}\right)U_{i+1} + \left(-4\nu - \nu_{T\,i+1} - 2\nu_{Ti} - \nu_{Ti-1}\right)U_i + \left(2\nu + \nu_{T\,i} + \nu_{T\,i-1}\right)U_{i-1}\right]$$

$$= 2\Delta y^2 \frac{1}{\rho}\left(\frac{dP_0}{dx}\right)$$

$\frac{1}{\rho}\left(\frac{\mathrm{dP}_0}{\mathrm{dx}}\right) = \frac{u_\tau^2}{h}$ where h means the channel-half height

$$\frac{\partial}{\partial x_j}\left(\left(\nu+\frac{\nu_T}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right)=\left(\varepsilon_{\mathrm{i}}-P_i\right)$$

$$\Rightarrow\left[\left(2\nu\sigma_k+\nu_{T\,i+1}+\nu_{T\,i}\right)k_{i+1}-\left(4\nu\sigma_k+\nu_{T\,i+1}+2\nu_{T\,i}+\nu_{T\,i-1}\right)\mathrm{k}_i+\left(2\nu\sigma_{\mathrm{k}}+\nu_{T\,i}+\nu_{T\,i-1}\right)k_{i-1}\right]$$

$$=2\Delta y^2\sigma_k\left(\varepsilon_{\mathrm{i}}-P_i\right)$$

$$\frac{\partial}{\partial x_j}\left(\left(\nu+\frac{\nu_T}{\sigma_\varepsilon}\right)\frac{\partial \varepsilon}{\partial x_j}\right)=\left[C_{\varepsilon 2}\frac{\varepsilon_{\mathrm{i}}^2}{k_i}-C_{\varepsilon 1}\frac{\varepsilon_{\mathrm{i}}}{k_i}P_i\right]$$

$$\Rightarrow\left[\left(2\nu\sigma_\varepsilon+\nu_{T\,i+1}+\nu_{T\,i}\right)\varepsilon_{i+1}-\left(4\nu\sigma_\varepsilon+\nu_{T\,i+1}+2\nu_{T\,i}+\nu_{T\,i-1}\right)\varepsilon_i+\left(2\nu\sigma_\varepsilon+\nu_{T\,i}+\nu_{T\,i-1}\right)\varepsilon_{i-1}\right]$$

$$=2\Delta y^2\sigma_\varepsilon\left[C_{\varepsilon 2}\frac{\varepsilon_{\mathrm{i}}^2}{k_i}-C_{\varepsilon 1}\frac{\varepsilon_{\mathrm{i}}}{k_i}P_i\right]$$

Then, we can obtain the below discretization formula

$$\begin{cases}\nu_{T\mathrm{i}}=C_\mu\dfrac{k^2{}_i}{\varepsilon}\\ P_{\mathrm{i}}=\nu_{T\mathrm{i}}\left(\dfrac{U_{i+1}-U_{i-1}}{2\Delta y}\right)^2\\ \left[\left(2\nu+\nu_{T\,i+1}+\nu_{T\,i}\right)U_{i+1}+\left(-4\nu-\nu_{T\,i+1}-2\nu_{Ti}-\nu_{Ti-1}\right)U_i+\left(2\nu+\nu_{T\,i}+\nu_{T\,i-1}\right)U_{i-1}\right]=2\Delta y^2\dfrac{1}{\rho}\left(\dfrac{\mathrm{d}P_0}{dx}\right)\\ \left[\left(2\nu\sigma_k+\nu_{T\,i+1}+\nu_{T\,i}\right)k_{i+1}-\left(4\nu\sigma_k+\nu_{T\,i+1}+2\nu_{T\,i}+\nu_{T\,i-1}\right)\mathrm{k}_i+\left(2\nu\sigma_{\mathrm{k}}+\nu_{T\,i}+\nu_{T\,i-1}\right)k_{i-1}\right]=2\Delta y^2\sigma_k\left(\varepsilon_{\mathrm{i}}-P_i\right)\\ \left[\left(2\nu\sigma_\varepsilon+\nu_{T\,i+1}+\nu_{T\,i}\right)\varepsilon_{i+1}-\left(4\nu\sigma_\varepsilon+\nu_{T\,i+1}+2\nu_{T\,i}+\nu_{T\,i-1}\right)\varepsilon_i+\left(2\nu\sigma_\varepsilon+\nu_{T\,i}+\nu_{T\,i-1}\right)\varepsilon_{i-1}\right]=2\Delta y^2\sigma_\varepsilon\left[C_{\varepsilon 2}\dfrac{\varepsilon_{\mathrm{i}}^2}{k_i}-C_{\varepsilon 1}\dfrac{\varepsilon_{\mathrm{i}}}{k_i}P_i\right]\end{cases}$$

For U

$$A_{P,\mathrm{j}}=2\nu+\nu_{T\,j+1}+\nu_{T\,j}$$

$$A_{C,\mathrm{j}}=-4\nu-\nu_{T\,j+1}-2\nu_{T\,j}-\nu_{Tj}$$

$$A_{M,\ \mathrm{j}}=2\nu+\nu_{T\,j}+\nu_{T\,j-1}$$

For k

$$A_{P,\mathrm{j}}=2\nu\sigma_{\mathrm{k}}+\nu_{T\,j+1}+\nu_{T\,j}$$

$$A_{C,\mathrm{j}}=-4\nu\sigma_{\mathrm{k}}-\nu_{T\,j+1}2\nu_{T\,j}-\nu_{Tj}$$

$$A_{M,\ \mathrm{j}}=2\nu\sigma_k+\nu_{T\,j}+\nu_{T\,j-1}$$

For ε

$$A_{P,\mathrm{j}}=2\nu\sigma_\varepsilon+\nu_{T\,j+1}+\nu_{T\,j}$$

$$A_{C,\mathrm{j}}=-4\nu\sigma_\varepsilon-\nu_{T\,j+1}2\nu_{T\,j}-\nu_{Tj}$$

$$A_{M,\ \mathrm{j}}=2\nu\sigma_\varepsilon+\nu_{T\,j}+\nu_{T\,j-1}$$

$$A_{P,j}\,U_{j+1}^{n} + A_{C,j}\,U_{j}^{n} + A_{M,j}\,U_{j-1}^{n} = Q_{j}^{n} \quad \Rightarrow \quad A\boldsymbol{U} = \boldsymbol{Q_u} \quad \text{where}$$

$$A = \begin{bmatrix} A_C & A_P & & & & \\ A_M & A_C & A_P & & & \\ & A_M & A_C & A_P & & \\ & & A_M & A_C & A_P & \\ & & & A_M & A_C & A_P \\ & & & & A_M & A_C \end{bmatrix} \qquad \boldsymbol{u} = \begin{bmatrix} U_1 \\ U_2 \\ . \\ . \\ . \\ . \\ U_n \end{bmatrix} \qquad \boldsymbol{Q_u} = \begin{bmatrix} Q_1 \\ Q_2 \\ . \\ . \\ . \\ . \\ Q_n \end{bmatrix}$$

Relaxation
Diagonal dominance

$$A^{*}{}_{C,\,\mathrm{j}} = A_{C,\mathrm{i}} / \alpha \,(0 \le \alpha \le 1), Q_{j}^{n,*} = Q_{j}^{n} + A_{C,\mathrm{j}} U_{j}^{n-1}\left(\frac{1-\alpha}{\alpha}\right)$$

Relaxation after calculation

$$U^{\mathrm{n},*} = \beta U^{n} + (1-\beta) U^{n-1} (0 \le \beta \le 1)$$

Boundary Conditions

Boundary Conditions at the wall
Mean velocity

$$\langle U \rangle_{\mathrm{w}} = 0$$

Turbulent kinetic energy

$$k_{w} = 0$$

Dissipation rate

$$\varepsilon_w \equiv 2\nu\langle s_{ij}s_{ij}\rangle = \nu\left\langle \frac{\partial u_i}{\partial x_j}\frac{\partial u_i}{\partial x_j} + \frac{\partial u_i}{\partial x_j}\frac{\partial u_j}{\partial x_i}\right\rangle = \nu\left\langle \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2\right\rangle$$

$$= \nu\left\langle \frac{1}{2}\frac{\partial^2 u^2}{\partial y^2} + \frac{1}{2}\frac{\partial^2 v^2}{\partial y^2} + \frac{1}{2}\frac{\partial^2 w^2}{\partial y^2} - \left(u\frac{\partial^2 u}{\partial y^2} + v\frac{\partial^2 v}{\partial y^2} + w\frac{\partial^2 w}{\partial y^2}\right)\right\rangle$$

$$= \nu\frac{\partial^2}{\partial y^2}\left\langle \frac{1}{2}(u^2+v^2+w^2)\right\rangle = \nu\frac{\partial^2 k}{\partial y^2} \approx \nu\frac{k_{offwall}}{\Delta y^2}$$

	U	k	diss	ν_T
Wall	$U = 0$	$k = 0$	$\epsilon = 2\nu \frac{d^2k}{dy^2}$	$\nu_T = 0$
Center	$\frac{dU}{dy} = 0$	$\frac{dk}{dy} = 0$	$\frac{d\epsilon}{dy} = 0$	$\frac{d\nu_T}{dy} = 0$

Boundary conditions for each variables in the fully developed channel flows.

Near Wall Model

the standard k-ε model is suitable for high-Reynolds number flows. For near-wall turbulent flows or stably stratified turbulent flows, the turbulent scales are not clearly separable. Hence, the use of $k^{3/2}/\varepsilon$ and k/ε for the length and time scales, respectively, is no longer appropriate in these regions.

Limiting behavior near the wall $y \to 0$

Taylor series expansion of velocities

$u = a_1 + a_2 y + a_3 y^2 + \ldots$

$v = b_1 + b_2 y + b_3 y^2 + \ldots$

$\langle U \rangle = c_1 + c_2 y + c_3 y^2 + \ldots$

$\langle V \rangle = d_1 + d_2 y + d_3 y^2 + \ldots$

No slip condition $a_1, b_1, c_1, d_1 = 0$

$u \sim y$

$v \sim y^2$

$U \sim y$

$V \sim y^2$

Continuity Equation $\left(\frac{\partial \langle U \rangle}{\partial x} + \frac{\partial \langle V \rangle}{\partial y} = 0, \frac{\partial u'}{\partial x} + \frac{\partial v'}{\partial y} = 0 \right) \Rightarrow b_2, d_2 = 0$

$k = \frac{1}{2}\left(u^2 + v^2\right) \sim y^2$

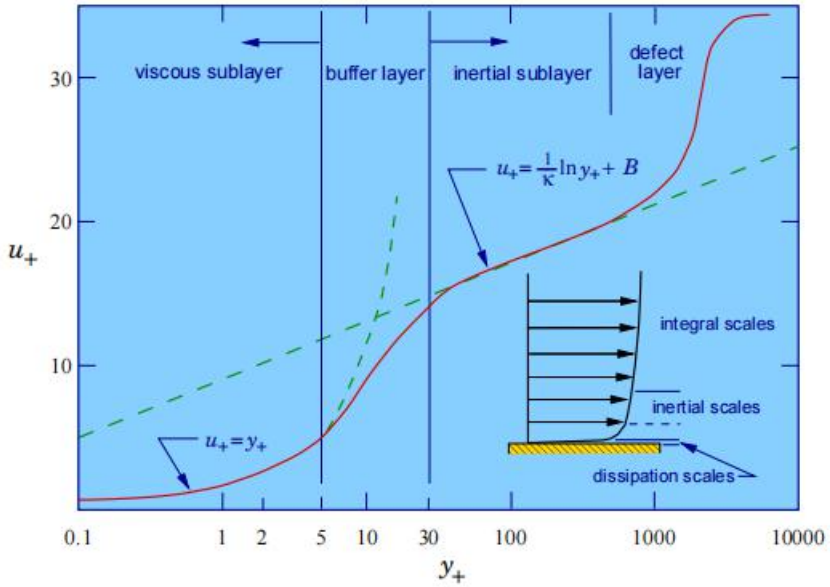
$\varepsilon = 2\nu \bar{s}_{i,j} \bar{s}_{i,j} \sim y^0$

Limiting behavior of analytic turbulent viscosity

$\nu_{t\_Analytic} = \frac{-\overline{u}\,\overline{v}}{d\langle U \rangle / dy} \sim y^3$

Limiting behavior of modeled turbulent viscosity

$$\nu_{\text{t\_Modeled}} \sim \frac{k^2}{\varepsilon} \sim y^4$$



Law of the Wall

Wall function model

Do not resolve viscous sub-layer, and the first grid node lie outside the viscous sub-layer ($y_p^+ \geq 20$). Boundary condition of the first grid is defined based on log-law.

To apply boundary conditions some distance away from the wall(in log-law region), so that the turbulence-model equations are not solved close to the wall. Once the y^+ is verified to lie within the log-law region, the velocity $\langle U \rangle$ can be specified using the equation below at the first grid point.

Mean Velocity

$$\langle U \rangle = u_\tau \left(\frac{1}{k} \ln y^+ + B \right) \qquad \text{(w)}$$

Where $y^+ = \frac{y u_\tau}{\nu}$ in the log-law region, u_τ is the friction velocity

Solving for the flow is effortless when the friction velocity u_τ is known. For example, the friction velocity $u_\tau = \sqrt{\tau_w / \rho}$ can be determined by equating the pressure gradient and the wall shear stress for fully developed turbulence under constant pressure gradient in pipe flow or channel flow.

$u^+ = \langle U \rangle / u_\tau$

For a more general case, in which the wall shear stress is determined from the flow, an iterative scheme is necessitated as presented below. We can define a function

F for an instance in time with a given $\langle U \rangle_p$ at y_p :

$$F(u_\tau) = \frac{\langle U \rangle_p}{u_\tau} - \frac{1}{\kappa} \ln \frac{y_p u_\tau}{\nu} - B = 0$$

where the objective is to determine the root u_τ ,the Newton-Raphson method

$u_\tau^{m+1} = u_\tau^m - \frac{F(u_\tau^m)}{F'(u_\tau^m)}, m = 0,1.2,...,$ can be used to solve above equation. Here the

$F'(u_\tau^m) = -(\overline{u}_p / u_\tau + 1/\kappa)/u_\tau$,When the iterative process has reached convergence, we have the frictional velocity u_τ . With this found solution, it should be checked whether y_p^+ for the found u_τ is within the log-law region.

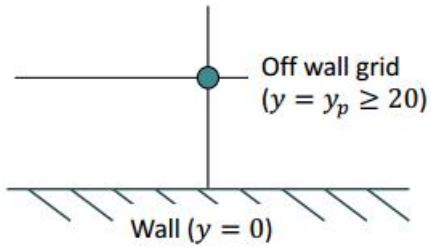
In the wall region, k and ε at y_p are provided by

Turbulent kinetic energy

$$-\langle \overline{uv} \rangle = u_\tau^2 = \left(C_\mu \frac{P}{\varepsilon} \right)^{1/2} k \Rightarrow k = \frac{u^2_\tau}{(C_\mu)^{1/2}} \text{ where } \varepsilon = P$$

Dissipation rate

$$\varepsilon \approx P = -\langle uv \rangle \frac{d\langle U \rangle}{dy} = \frac{u^3_\tau}{\kappa y_p} \text{ in log law region.}$$



We can further generalize the application of the law of the wall by replacing Eq. (w) with Spalding's law of the wall

$$y_p^+ = \langle U \rangle_p^+ + \exp(-\kappa B) \left[EXP\left(\kappa \langle U \rangle_p^+\right) - 1 - \kappa \langle U \rangle_p^+ - \frac{\left(\kappa \langle U \rangle_p^+\right)^2}{2} - \frac{\left(\kappa \langle U \rangle_p^+\right)^3}{6} \right]$$

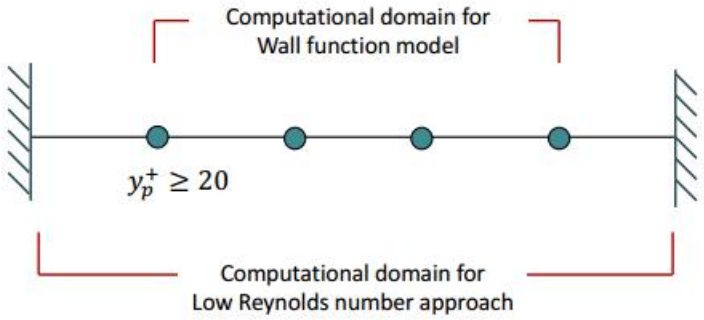
so that there would not be an issue with y_p^+ being in the buffer or viscous sub-layer. However, it

would not be advisable to have the first grid point lie in a region where it would be appropriate to specify a boundary condition for the higher-Reynolds number k-ε model.

Low Reynolds-number approach k-epsilon model

The wall law is not universal for all turbulent flows. For flows in which the log wall does not hold, the iterative scheme mentioned above would diverge or provide unphysical results. For problems where the various turbulent quantities are especially important near the near-wall regions, such as flows with heat transfer physics, it is necessary to prescribe the no-slip condition at the wall (for a stationary wall, k = 0 and $U_i = 0$) and compute all the way up to the wall boundary. A model that takes the low-Reynolds number effect near the wall into account and allow for computations of the overall flow field is referred to as the low-Reynolds number k-ε model.
The common formulation of the low-Reynolds number k-ε model involves the use of the damping function fμ for ν_T and the correction functions f1 and f2 for the production and dissipation terms in the ε-equation, and the wall correction terms D and E in the k and ε-equations, respectively

Governing equations are solved right up to the wall (viscous sublayer is resolved)
And wall proximity effects are accounted by applying damping function of the eddy viscosity ($\nu_t = c_\mu f_\mu k^2 / \varepsilon$)



Sufficient grid resolution is required near the wall to accurately reproduce turbulent flows. For high-Reynolds-number flows, it may be a challenge to produce (and perform computation on) such a large grid. Even if the grid resolution may not be as fine as one would like, results can be obtained with the no-slip boundary condition and a low-Reynolds number k-ε model (the grid size still needs to be reasonable). For models that satisfy the asymptotic behavior near the wall, the results do not appear to be significantly affected by the use of coarse grids, as long as y+ ≤ 2 to 4 for the first grid point.

Damping function for the eddy viscosity

To reflects the effects of wall proximity (To make $\nu_t \sim y^3$ near the wall)

$$\nu_t = c_\mu f_\mu k^2 / \varepsilon$$

van Driest damping function (van Driest, 1954)

$f_\mu = \left[1 - \exp\left(- y^+ / A^+\right)\right]^2$ where $y^+ = \frac{y u_\tau}{\nu}$, A^+ : model constant

Damping function (Launder and Sharma, 1974)

$f_\mu = \exp\left[\frac{-3.4}{\left(1 + R_t / 50\right)^2}\right]$ where $R_t = k^2 / \nu\varepsilon$

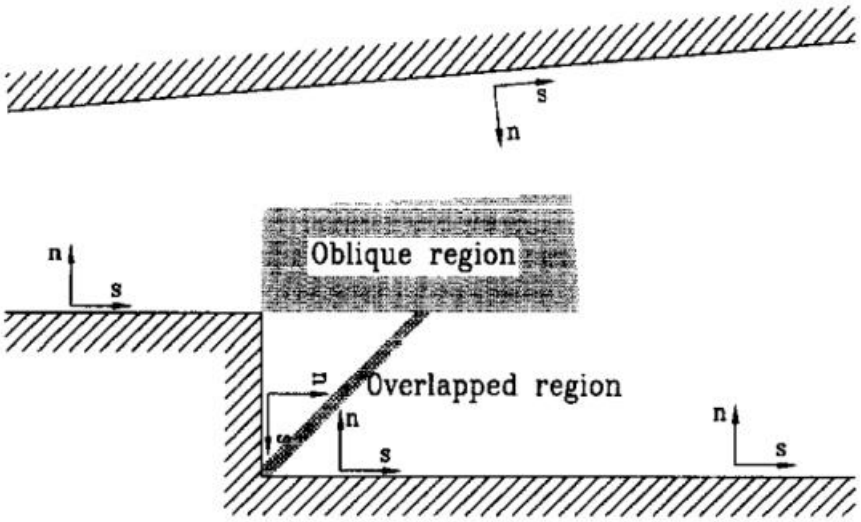
Damping function (Lam and Bremhorst, 1981)

$f_\mu = \left[1 - \exp\left(-0.0165 R_y\right)\right]^2 \left(1 + \frac{20.5}{R_t}\right)$ Where $R_y = \sqrt{k}\, y, R_t = k^2 / \nu\varepsilon$

General damping functions

$f_w = 1 - \exp\left(-\lambda / A\right)$ Where $\lambda = y u_\tau / \nu$

Difficulties are amplified in selecting an appropriate normal distance from the wall



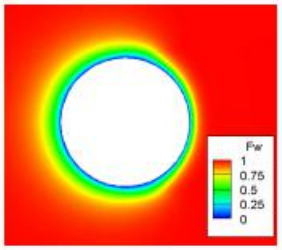
Schematics of the oblique and overlapped regions

General elliptic equation for é£ (Park et al., 1997)

$L^2 \nabla^2 f_w = \frac{R_t^{3/2}}{A^2}\left(f_w - 1\right)$

where

L : turbulent length scale
R_t : turbulent Re
A : model constant



Contour of f_w near wall

$k - \varepsilon - f_\mu$ model (Park et al., 1997)

Turbulent viscosity

$$\nu_t = c_\mu f_\mu k^2 / \varepsilon$$

Turbulent kinetic energy equation

$$\frac{\partial k}{\partial t} + \langle U \rangle_i \frac{\partial k}{\partial x_i} = \frac{\partial}{\partial x_i}\left\{\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_i}\right\} + \mathcal{P} - \varepsilon$$

Dissipation equation

$$\frac{\partial \varepsilon}{\partial t} + \langle U \rangle_i \frac{\partial \varepsilon}{\partial x_i} = \frac{\partial}{\partial x_i}\left\{\left(\nu + \frac{\nu_t}{\sigma_\varepsilon}\right)\frac{\partial \varepsilon}{\partial x_i}\right\} + C_{\epsilon 1}\frac{\mathcal{P}\varepsilon}{k} - C_{\epsilon 2} f_2 \frac{\varepsilon^2}{k} - E$$

Helmoltz equation for wall damping function

$$L^2 \nabla^2 f_w = \frac{R^{1.5}}{A^2}(f_w - 1)$$

where

$$f_\mu = f_{\mu 1} f_{\mu 2}$$

$$f_{\mu 1} = \left(1 + C_D \exp(-(R_t/120)^2)\, R_t^{-1.25}\right) f_w^2$$

$$f_{\mu 2} = 7.0 \frac{(4.5+0.3\mathcal{P}/\epsilon)}{(4.5+1.3\mathcal{P}/\epsilon)^2}$$

k-epsilon model for non-equilibrium effect

Non-equilibrium effect

In complex flows away from the wall, local equilibrium is not satisfied ($P \neq \varepsilon$).

$$C_\mu \neq 0.09, = f\left(P/\varepsilon\right)$$

$\mathrm{k} - \varepsilon - f_\mu$ model (Park et al., 1997)

Damping function for non-equilibrium effect

$$C_\mu^* = C_\mu f_{\mu 2} \quad \text{where} \quad f_{\mu 2} = 7.0 \frac{(4.5+0.3\mathcal{P}/\epsilon)}{(4.5+1.3\mathcal{P}/\epsilon)^2}$$

Results

RANS with k-ε model with no NWM(Near Wall Model)

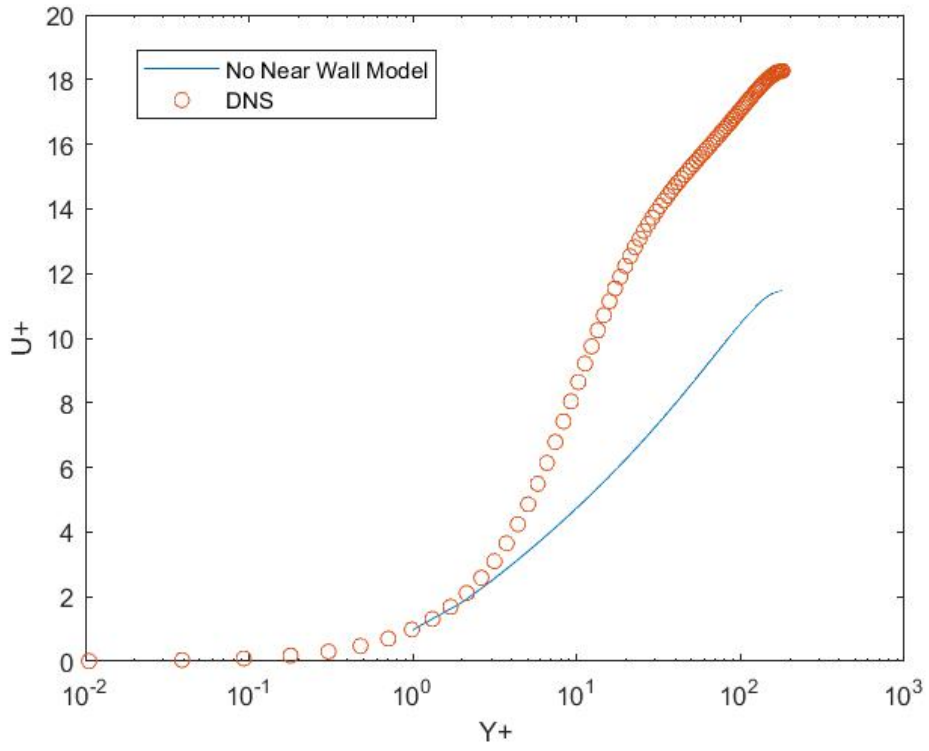
First RANS with k-ε model with no NWM(Near Wall Model) is applied to fully developed channel flows and the results of U +, k+, ε + are shown on below figure. As below profiles show, U + and k+ are under-estimated that it didn't simulate the flow well. Especially near wall region, viscous sublayer is not well resolved because limiting behaviors of analytic and modeled ν_T (eddy viscosity) are different. Therefore unresolved near wall region, including sublayer to log-layer, makes the gap between DNS data, even its value is half at velocity profile.

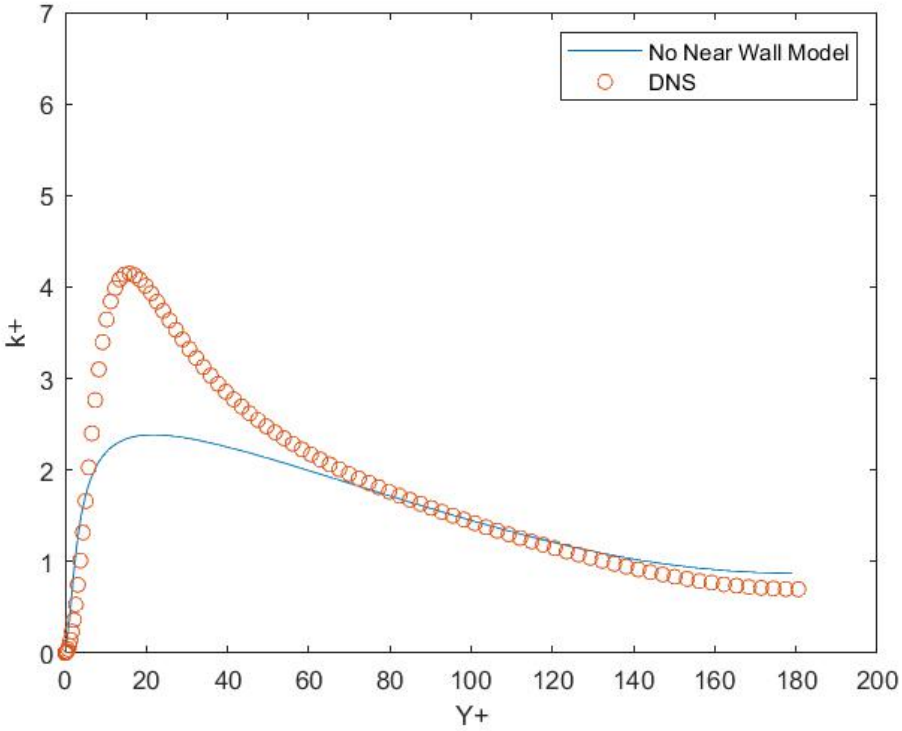
Constants for simulation	
Parameters	Value
maximum interation number	10000000
criteria for convergence	0
tolerance for convergence	tol = 1e-8
the number of grid cells	Ny = 360
the channel-half height	del = 1
grid size	(2*del)/Ny
Reynolds number based on friction velocity	Re_tau = 180
Kinematic viscosity of reference data	nu = 3.5000e-4
Friction velocity	u_tau = Re_tau*nu/del

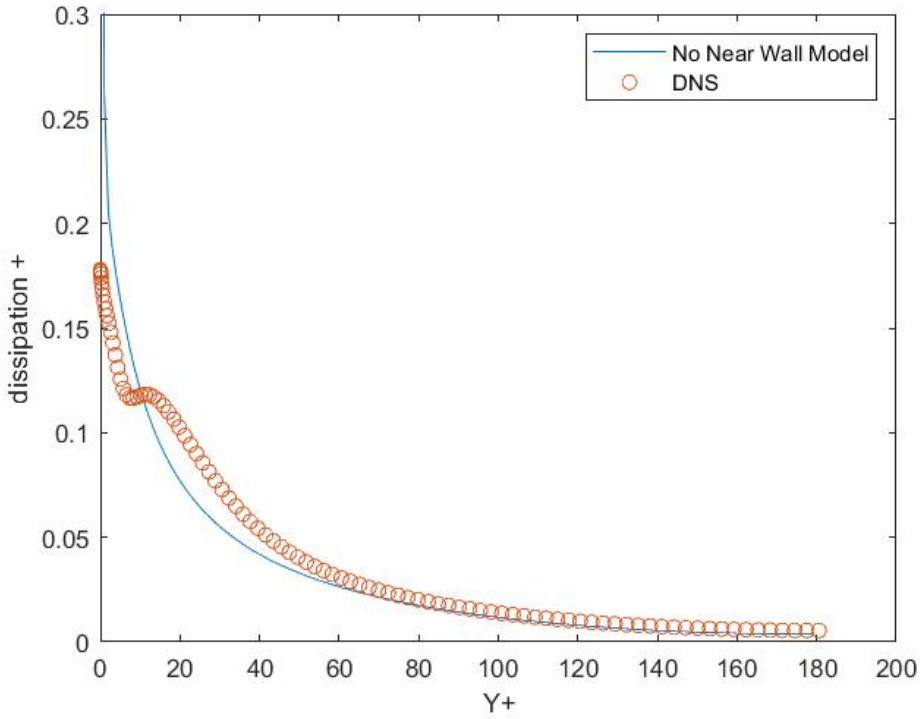
Constants for k-e model	
$C\mu$	0.09
$C_{\varepsilon 1}$	1.44
$C_{\varepsilon 2}$	1.92
σ_{k}	1.0
σ_{ε}	1.3

Relaxation factors	
alpha	0.1
beta	0.4

Initial Condition	
k(j)	0.1
$\sigma(j)$	0.03







van Driest(1957)

Since the limiting behaviour does not match their order, NWM have to be applied to resolve the near wall region flow to make simulation more accurate and realistic. Using van Driest damping function, RANS velocity profile agrees well with DNS reference data. But because of its simple damping function form, it is fail to implement the dissipation and kinetic energy well. Also at the

center of channel, velocity profile is a little underestimated then DNS dataset. Maybe it is guessed that the damping function distorted its velocity profiles.

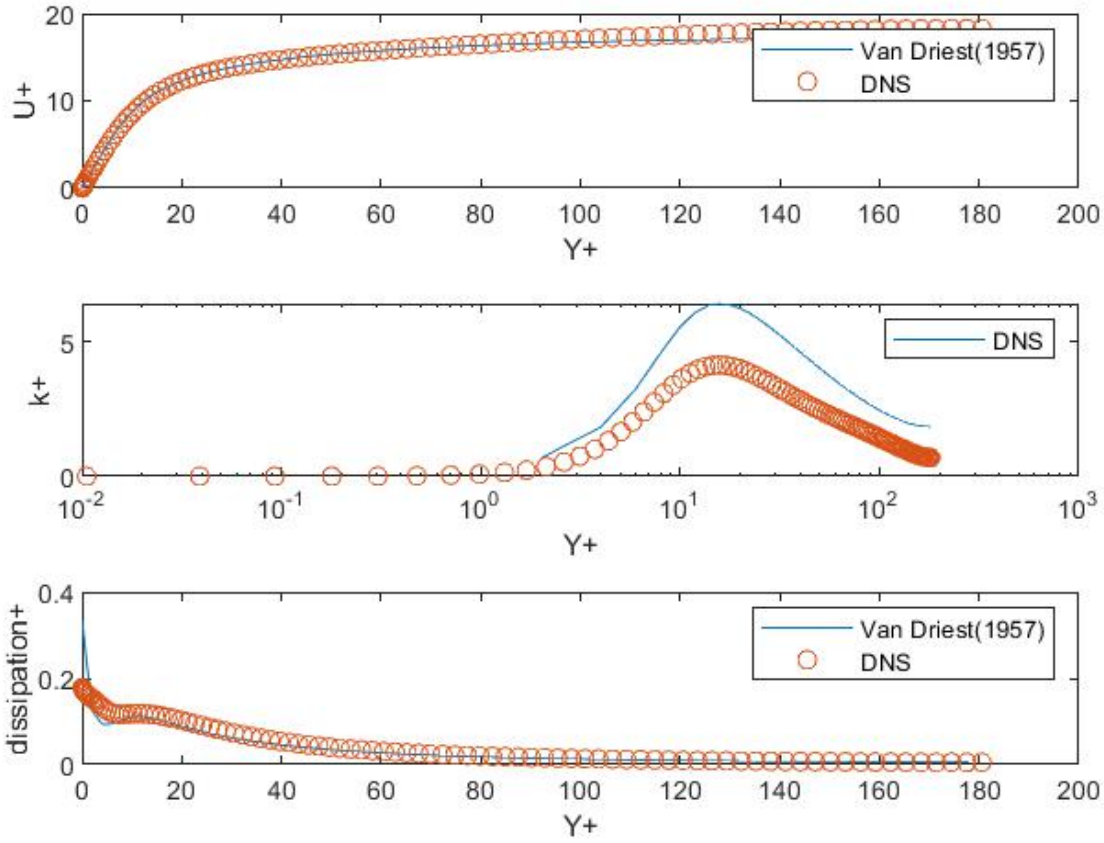
Constants for simulation	
Parameters	Value
maximum interation number	10000000
criteria for convergence	0
tolerance for convergence	tol = 1e-10
the number of grid cells	Ny = 180
the channel-half height	del = 1
grid size	(2*del)/Ny
Reynolds number based on friction velocity	Re_tau = 180
Kinematic viscosity of reference data	nu = 3.5000e-4
Friction velocity	u_tau = Re_tau*nu/del

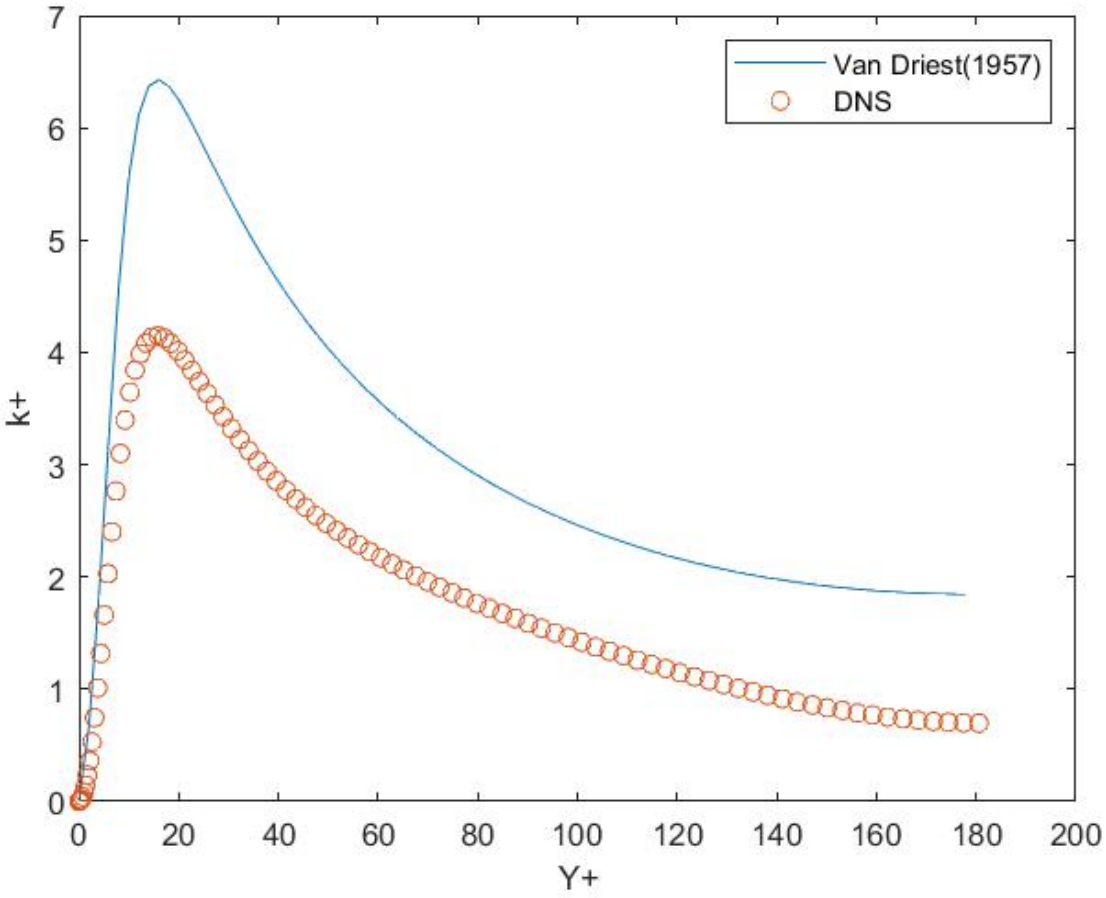
Constants for k-e model	
$C\mu$	0.09
$C_{\varepsilon 1}$	1.44
$C_{\varepsilon 2}$	1.92
σ_{k}	1.0
σ_{ε}	1.3
A0	42
A1	60
Cd	20
Cp	0.2
Ce	70

Relaxation factors	
alpha	0.1

beta 0.2

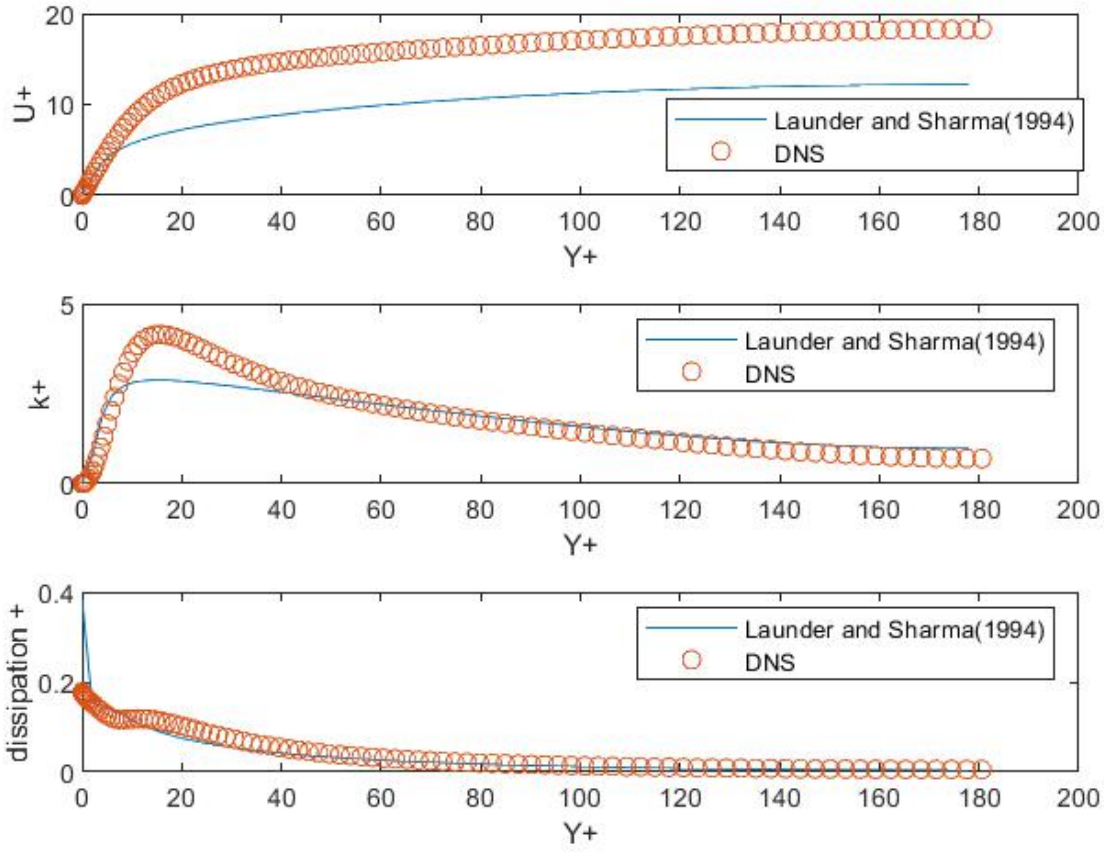
Initial Condition	
k(j)	0.01
$\sigma(j)$	0.005





Launder and Sharma(1994)

Using Launder and Sharma damping (1994) function, RANS velocity profile over-estimated with DNS reference data. k has litter gap between the RANS model and DNS data in $5 < y^+ < 40$ which belong to the buffer layer. it is also fail to implement the dissipation well



Park et al (1997)

Using general elliptic equation for damping function with Park's approach, all the profiles are more likely to DNS reference. Especially modifying dissipation equation, considering turbulent time scale and other production terms, shape and value of dissipation at RANS simulation is very similar to DNS. Although k has a little gap between RANS simulation and DNS data, its shape is similar and near the center channel, the values are almost same that this Park's approach is proper to adjust near wall region.

Constants for simulation

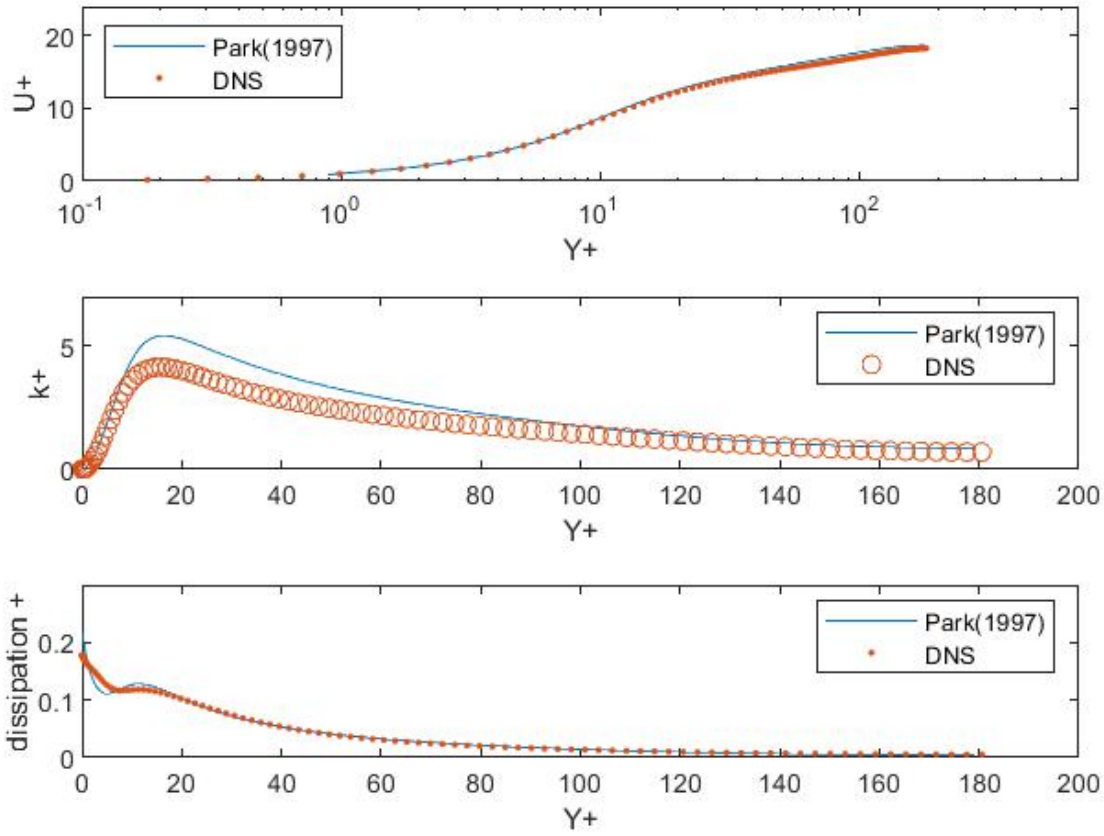
Parameters	Value
maximum interation number	10000000
criteria for convergence	0
tolerance for convergence	tol = 1e-10
the number of grid cells	Ny = 200
the channel-half height	del = 1
grid size	(2*del)/Ny
Reynolds number based on friction velocity	Re_tau = 180

Kinematic viscosity of reference data	nu = 3.5000e-4
Friction velocity	u_tau = Re_tau*nu/del

Constants for k-e model	
$C\mu$	0.09
$C_{\varepsilon 1}$	1.45
$C_{\varepsilon 2}$	1.8
σ_{k}	1.2
σ_{ε}	1.3
A1	60
A4	10
B0	0.25
C1	0.4
Cd	20
Cp	0.2
Ce	70
Ct	6

Relaxation factors	
alpha	0.3
beta	0.3

Initial Condition	
k(j)	0.1
$\sigma(j)$	0.05



Code

To solve the 1D fully developed channel flow with RANS model and k-epsilon model.

```
1.    PROGRAM RANS_main
2.
3.            USE RANS_module,
   &
4.                ONLY : path_name, itmax, resi, tol, dis_new, nu, u_tau,
   U, Ny
5.
6.            IMPLICIT NONE
7.            INTEGER :: it
8.            REAL(8) :: time_sta, time_end
9.
10.           !-------------------------------------------!
11.           !            Make Result folder
12.           !-------------------------------------------!
13.           path_name = 'RESULT'
```

```
14.           CALL SYSTEM('mkdir '//TRIM(path_name))
15.           CALL SYSTEM('rm -rf ./'//TRIM(path_name)//'/*')
16.
17.           !---------------------------------------------!
18.           !              Initial Setting
19.           !---------------------------------------------!
20.           CALL SETUP
21.           CALL POISEUILLE
22.
23.           !---------------------------------------------!
24.           !                 Main loop
25.           !---------------------------------------------!
26.           CALL CPU_TIME(time_sta)
27.           DO it = 0, itmax
28.             CALL GETNUT
29.             CALL GETU
30.             CALL GETPROD
31.             CALL GETK
32.             CALL GETDIS
33.
   WRITE(*,"(A,I7.7,A,E14.7,2X,F10.7,2X,E14.7)")'Iteration(',it,') : '
   &
34.                  ,resi,U(Ny/2)/u_tau,dis_new(0)/(nu*(u_tau**2/nu)**2)
35.             IF (resi < tol) EXIT
36.           END DO
37.           CALL CPU_TIME(time_end)
38.           WRITE(*,"(A,F12.7,A)")'Total calculation time :
   ',time_end-time_sta,'s'
39.
40.           !---------------------------------------------!
41.           !             Write final result
42.           !---------------------------------------------!
43.           CALL OUTPUT
44.
45.         END PROGRAM RANS_main
```

To Setup the 1D turbulent channel flow for RANS model

```
46.   SUBROUTINE SETUP
47.
48.             USE RANS_module,
   &
49.                 ONLY : Ny, del, dy, Re_tau, nu, u_tau,
   &
```

```
50.                       Cm, Ce1, Ce2, Sk, Se, alpha, beta, itmax, resi,
   tol
51.
52.             USE RANS_module,
   &
53.                 ONLY : U, U_exac, U_new, Y, k, k_new, dis, dis_new, nu_T,
   prod
54.
55.             IMPLICIT NONE
56.             INTEGER :: i,j
57.
58.             !-------------------------------------------------------
   ----!
59.             !                Constants for simulation
60.             !-------------------------------------------------------
   ----!
61.             itmax = 10000000       ! maximum interation number
62.             resi = 0              ! criteria for convergence
63.             tol = 1e-8           ! tolerance for convergence
64.
65.             Ny  = 360             ! the number of grid cells
66.             del = 1               ! the channel-half height
67.             dy  = (2*del)/Ny       ! grid size
68.
69.             Re_tau = 180          ! Reynolds number based on friction
   velocity
70.             nu     = 3.5000e-4     ! Kinematic viscosity of reference
   data
71.             u_tau  = Re_tau*nu/del ! Friction velocity
72.
73.             !-------------------------------------------------------
   ----!
74.             !                  Constants for k-e model
75.             !-------------------------------------------------------
   ----!
76.             Cm  = 0.09
77.             Ce1 = 1.44
78.             Ce2 = 1.92
79.             Sk  = 1.0
80.             Se  = 1.3
81.
82.             !-------------------------------------------------------
   ----!
83.             !                    Relaxation factors
```

```
84.            !-------------------------------------------------------
   ----!
85.            alpha = 0.1
86.            beta  = 0.4
87.
88.
   ALLOCATE( U(0:NY),U_new(0:Ny),U_exac(0:Ny),Y(0:Ny),prod(0:Ny) )
89.
   ALLOCATE( k(0:Ny),k_new(0:Ny),dis(0:Ny),dis_new(0:Ny),nu_T(0:Ny) )
90.
91.            !-------------------------------------------------------
   ----!
92.            !                     Initial Conditions
93.            !-------------------------------------------------------
   ----!
94.            DO j = 0,Ny
95.              Y(j)       = j*dy
96.              U(j)       = 0
97.              k(j)       = 0.1000
98.              dis(j)     = 0.0300
99.              nu_T(j)    = 0
100.             prod(j)    = 0
101.
102.             k_new(j)   = 0
103.             U_new(j)   = 0
104.             dis_new(j) = 0
105.             U_exac(j)   = -(nu/(2*del))*(Re_tau/del)**2 * Y(j) *
   (Y(j)-2*del)
106.           END DO
107.
108.       END SUBROUTINE SETUP
```

To make the initial poiseuille flow

```
109.    SUBROUTINE POISEUILLE
110.
111.        USE RANS_module,
   &
112.           ONLY : Ny, dy, del, Re_tau, nu
113.
114.        USE RANS_module,
   &
115.           ONLY : U
116.
117.        IMPLICIT NONE
```

```
118.          INTEGER :: i, j
119.          REAL(KIND=8),DIMENSION(:),ALLOCATABLE :: a,b,c,r,x
120.
121.          ALLOCATE ( a(0:Ny), b(0:Ny), c(0:Ny), r(0:Ny), x(0:Ny))
122.
123.          a(0:Ny) = 1
124.          b(0:Ny) = -2
125.          c(0:Ny) = 1
126.          x(0:Ny) = U(0:Ny)
127.          r(1:Ny-1) = -(nu/del)*(dy*Re_tau/del)**2
128.
129.          b(0)  = 1
130.          b(Ny) = 1
131.          a(Ny) = 0
132.          c(0)  = 0
133.          r(0)  = 0
134.          r(Ny) = 0
135.
136.          CALL TDMA_Solver(a,b,c,r,x,Ny)
137.
138.          U(0:Ny) = x(0:Ny)
139.          DEALLOCATE(a,b,c,r,x)
140.
141.      END SUBROUTINE POISEUILLE
```

Get f_m(damping function) using k-e model relation

```
142. SUBROUTINE GETFM
143.
144.            USE RANS_module,
   &
145.              ONLY : Ny, Cm, k, dis, nu_T, fm, prod, nu, u_tau, Y, del,
   mode,   &
146.                     dy, fw, A0, A1, Cd, Cp, Ce
147.
148.            IMPLICIT NONE
149.            INTEGER :: j
150.            REAL(KIND=8) :: L, fm1, fm2, C1
151.            REAL(KIND=8),DIMENSION(:),ALLOCATABLE ::
   Rt,Y_tmp,a,b,c,r,x
152.
153.            ALLOCATE( Y_tmp(0:Ny), Rt(0:Ny))
154.            ALLOCATE( a(0:Ny), b(0:Ny), c(0:Ny), r(0:Ny), x(0:Ny) )
155.
156.            IF (mode == 4) THEN
```

```
157.             !-----------------------------------------------------
   ------!
158.             !                    Set TDMA constants
159.             !-----------------------------------------------------
   ------!
160.             DO j = 1,Ny-1
161.               Rt(j) = k(j)**2. /(nu * dis(j))
162.               L     = Cp*sqrt(k(j)**3./dis(j)**2. +
   Ce**2.*(nu**3./dis(j))**(1./4.))
163.               C1    = Rt(j)**1.5/(A0*L)**2.
164.
165.               a(j)  =   1.
166.               b(j)  = -(2. + C1*dy**2.)
167.               c(j)  =   1.
168.               r(j)  = - C1*dy**2.
169.             END DO
170.             x(0:Ny) = fw(0:Ny)
171.
172.             !-----------------------------------------------------
   ------!
173.             !                    Boundary conditions
174.             !-----------------------------------------------------
   ------!
175.             b(0)  = 1.
176.             b(Ny) = 1.
177.             a(Ny) = 0.
178.             c(0)  = 0.
179.             r(0)  = 0.
180.             r(Ny) = 0.
181.
182.             Rt(0) = 0.
183.             Rt(Ny)= 0.
184.
185.             CALL TDMA_Solver(a,b,c,r,x,Ny)
186.             fw(0:Ny) = x(0:Ny)
187.           END IF
188.
189.           !-----------------------------------------------------
   ----!
190.           !                 Damping funnction setting
191.           !-----------------------------------------------------
   ----!
192.           DO j = 0,Ny
193.              IF (j<Ny/2) THEN
194.                 Y_tmp(j) = Y(j)
```

```
195.               ELSE
196.                 Y_tmp(j) = 2.*del - Y(j)
197.               END IF
198.             END DO
199.
200.              SELECT CASE (mode)
201.                 CASE(0) ! No wall model
202.                   fm(0:Ny) = 1.
203.
204.                 CASE(1) ! van Driest (1954)
205.                   fm(0:Ny) = ( 1. -
   exp(-Y_tmp(0:Ny)/(nu/u_tau)/A1))**2.
206.
207.                 CASE(2) ! Launder and Sharma (1974)
208.                   Rt(0:Ny) = k(0:Ny)**2. /(nu * dis(0:Ny))
209.                   fm(0:Ny) = exp( -3.4/(1. + Rt(0:Ny)/50.)**2. )
210.
211.                 CASE(4) ! Park et al (1997)
212.                   DO j = 1,Ny-1
213.                     fm1   = (1. +
   Cd*exp(-(Rt(j)/120.)**2.)*Rt(j)**(-3./4.))    &
214.                                *fw(j)**2.
215.                     fm2   = 7.0*(4.5 + 0.3*prod(j)/dis(j))
   &
216.                                /(4.5 + 1.3*prod(j)/dis(j))**2.
217.
218.                     fm(j) = fm1 * fm2
219.                     ! print*,fm(j),fm1,fm2
220.                   END DO
221.                   fm(0)  = 0.
222.                   fm(Ny) = 0.
223.              END SELECT
224.
225.             DEALLOCATE(Y_tmp)
226.
227.         END SUBROUTINE GETFM
```

Get Nu_t(turbulent kinematic viscosity) using k-e model relation

```
228.       SUBROUTINE GETNUT
229.
230.             USE RANS_module,
   &
231.               ONLY : Ny, Cm, k, dis, nu_T, fm
232.
```

```
233.            IMPLICIT NONE
234.            INTEGER :: j
235.            REAL(KIND=8) :: Rt, Ry
236.
237.            DO j = 0,Ny
238.              nu_T(j) = Cm *fm(j)* k(j)**2. / dis(j)
239.              ! print*,fm(j),nu_T(j)
240.            END DO
241.
242.        END SUBROUTINE GETNUT
```

Get U(mean velocity) using k-e model relation

```
243.SUBROUTINE GETU
244.
245.            USE RANS_module,
   &
246.              ONLY : Ny, dy, nu, del, Re_tau, u_tau ,alpha, beta, resi
247.
248.            USE RANS_module,
   &
249.              ONLY : U, U_new, nu_T
250.
251.            IMPLICIT NONE
252.            INTEGER :: i,j
253.            REAL(KIND=8) :: C0
254.
255.            REAL(KIND=8),DIMENSION(:),ALLOCATABLE :: a,b,c,r,x
256.
257.            ALLOCATE ( a(0:Ny), b(0:Ny), c(0:Ny), r(0:Ny), x(0:Ny))
258.
259.            resi = 0
260.            C0   = u_tau**2 / del
261.
262.            !-------------------------------------------------------
   ----!
263.            !                    Set TDMA constants
264.            !-------------------------------------------------------
   ----!
265.            DO j = 1,Ny-1
266.              a(j) =   2*nu + nu_T(j-1) + nu_T(j)
267.              b(j) = -(4*nu + nu_T(j+1) + 2*nu_T(j) + nu_T(j-1))
268.              c(j) =   2*nu + nu_T(j+1) + nu_T(j)
269.              r(j) = -2*dy**2*C0
```

```
270.           END DO
271.           r(0:Ny) = r(0:Ny) + b(0:Ny) * U(0:Ny)*(1-alpha)/alpha
272.           b(0:Ny) = b(0:Ny) / alpha
273.           x(0:Ny) = U(0:Ny)
274.
275.           !-------------------------------------------------------
   ----!
276.           !                    Boundary conditions
277.           !-------------------------------------------------------
   ----!
278.           b(0)  = 1
279.           b(Ny) = 1
280.           a(Ny) = 0
281.           c(0)  = 0
282.           r(0)  = 0
283.           r(Ny) = 0
284.
285.           !-------------------------------------------------------
   ----!
286.           !                      Calculate TDMA
287.           !-------------------------------------------------------
   ----!
288.           CALL TDMA_Solver(a,b,c,r,x,Ny)
289.
290.           !-------------------------------------------------------
   ----!
291.           !                   Relaxation & Update
292.           !-------------------------------------------------------
   ----!
293.           U_new(0:Ny) = beta * x(0:Ny) + (1-beta) * U(0:Ny)
294.           DO j = 0,Ny
295.             resi = resi + (U_new(j) - U(j))**2
296.           END DO
297.           resi = sqrt(resi)/Ny
298.
299.           U(0:Ny) = U_new(0:Ny)
300.           DEALLOCATE(a,b,c,r,x)
301.
302.       END SUBROUTINE GETU
```

```
303.  SUBROUTINE TDMA_Solver(a,b,c,r,x,n)
304.
305.           IMPLICIT NONE
306.           INTEGER :: it, n
307.           REAL(8) :: a(0:n), b(0:n), c(0:n), r(0:n), x(0:n)
```

```
308.
309.           a(0) = 0.0
310.           c(n) = 0.0
311.
312.           !---Forward Sweeping-----
313.
314.           DO it = 1,n
315.               b(it) = b(it-1) * b(it) - a(it) * c(it-1)
316.               c(it) = b(it-1) * c(it)
317.               r(it) = b(it-1) * r(it) - a(it) * r(it-1)
318.           END DO
319.
320.           !---Backward Sweeping----
321.
322.           x(n) = r(n) / b(n)
323.
324.           DO it = n-1,0,-1
325.               x(it) = (r(it) - c(it) * x(it+1))/b(it)
326.           END DO
327.
328.       END SUBROUTINE TDMA_Solver
```

Get P(TKE Production) using k-e model relation

```
329.  SUBROUTINE GETPROD
330.
331.           USE RANS_module,
   &
332.             ONLY : Ny, dy, u_tau, nu
333.
334.           USE RANS_module,
   &
335.             ONLY : U_new, nu_T, prod
336.
337.           IMPLICIT NONE
338.           INTEGER :: j
339.
340.           DO j = 1,Ny-1
341.             prod(j) = nu_T(j) * ((U_new(j+1) -
   U_new(j-1))/(2.*dy))**2.
342.           END DO
343.
344.           !-------------------------------------------------------
   ----!
```

```
345.            !                    Boundary conditions
346.            !-------------------------------------------------------
   ----!
347.            prod(0)  = nu_T(0)  * (u_tau**2/nu)**2.
348.            prod(Ny) = nu_T(Ny) * (u_tau**2/nu)**2.
349.
350.         END SUBROUTINE GETPROD
```

Get k(turbulent kinetic energy) using k-e model relation

```
351. SUBROUTINE GETK
352.
353.            USE RANS_module,
   &
354.              ONLY : Ny, dy, nu, Sk, alpha, beta
355.
356.            USE RANS_module,
   &
357.              ONLY : k, k_new, dis, U_new, nu_T, prod
358.
359.            IMPLICIT NONE
360.            INTEGER :: j
361.
362.            REAL(KIND=8),DIMENSION(:),ALLOCATABLE :: a,b,c,r,x
363.
364.            ALLOCATE ( a(0:Ny), b(0:Ny), c(0:Ny), r(0:Ny), x(0:Ny))
365.
366.            !-------------------------------------------------------
   ----!
367.            !                    Set TDMA constants
368.            !-------------------------------------------------------
   ----!
369.            DO j = 1,Ny-1
370.              a(j) =   Sk*2.*nu + nu_T(j-1) + nu_T(j)
371.              b(j) = -(Sk*4.*nu + nu_T(j+1) + 2.*nu_T(j) + nu_T(j-1))
372.              c(j) =   Sk*2.*nu + nu_T(j+1) + nu_T(j)
373.              r(j) = 2.*dy**2.*Sk* ( dis(j) - prod(j) )
374.            END DO
375.            r(0:Ny) = r(0:Ny) + b(0:Ny) * k(0:Ny)*(1.-alpha)/alpha
376.            b(0:Ny) = b(0:Ny) / alpha
377.            x(0:Ny) = k(0:Ny)
378.
379.            !-------------------------------------------------------
   ----!
380.            !                    Boundary conditions
```

```
381.           !-------------------------------------------------------
   ----!
382.           b(0)  = 1.
383.           b(Ny) = 1.
384.           a(Ny) = 0.
385.           c(0)  = 0.
386.           r(0)  = 0.
387.           r(Ny) = 0.
388.
389.           !-------------------------------------------------------
   ----!
390.           !                      Calculate TDMA
391.           !-------------------------------------------------------
   ----!
392.           CALL TDMA_Solver(a,b,c,r,x,Ny)
393.
394.           !-------------------------------------------------------
   ----!
395.           !                   Relaxation & Update
396.           !-------------------------------------------------------
   ----!
397.           k_new(0:Ny) = beta * x(0:Ny) + (1.-beta) * k(0:Ny)
398.           k(0:Ny) = k_new(0:Ny)
399.           DEALLOCATE(a,b,c,r,x)
400.
401.        END SUBROUTINE GETK
```

Get dis(dissipation) using k-e model relation

```
402.  SUBROUTINE GETDIS
403.
404.           USE RANS_module,
   &
405.             ONLY : Ny, dy, nu, Se, Ce1, Ce2, alpha, beta
406.
407.           USE RANS_module,
   &
408.             ONLY : k_new, dis, dis_new, U_new, nu_T, prod
409.
410.           IMPLICIT NONE
411.           INTEGER :: j
412.
413.           REAL(KIND=8),DIMENSION(:),ALLOCATABLE :: a,b,c,r,x
414.
415.           ALLOCATE ( a(0:Ny), b(0:Ny), c(0:Ny), r(0:Ny), x(0:Ny))
416.
```

```
417.            !-------------------------------------------------------
   ----!
418.            !                    Set TDMA constants
419.            !-------------------------------------------------------
   ----!
420.            DO j = 1,Ny-1
421.              a(j) =   Se*2.*nu + nu_T(j-1) + nu_T(j)
422.              b(j) = -(Se*4.*nu + nu_T(j+1) + 2.*nu_T(j) + nu_T(j-1))
423.              c(j) =   Se*2.*nu + nu_T(j+1) + nu_T(j)
424.              r(j) = 2.*dy**2.*Se* ( Ce2*dis(j)**2./k_new(j)
   &
425.                                   - Ce1*dis(j)/k_new(j)*prod(j) )
426.            END DO
427.
428.            x(0:Ny) = dis(0:Ny)
429.
430.            !-------------------------------------------------------
   ----!
431.            !                    Boundary conditions
432.            !-------------------------------------------------------
   ----!
433.            b(0)  = 1.
434.            b(Ny) = 1.
435.            a(Ny) = 0.
436.            c(0)  = 0.
437.            r(0)  = 2.*nu*k_new(1)/(dy**2.)
438.            r(Ny) = 2.*nu*k_new(Ny-1)/(dy**2.)
439.
440.            r(0:Ny) = r(0:Ny) + b(0:Ny) * dis(0:Ny)*(1.-alpha)/alpha
441.            b(0:Ny) = b(0:Ny) / alpha
442.
443.            !-------------------------------------------------------
   ----!
444.            !                     Calculate TDMA
445.            !-------------------------------------------------------
   ----!
446.            CALL TDMA_Solver(a,b,c,r,x,Ny)
447.
448.            !-------------------------------------------------------
   ----!
449.            !                  Relaxation & Update
450.            !-------------------------------------------------------
   ----!
451.            dis_new(0:Ny) = beta * x(0:Ny) + (1.-beta) * dis(0:Ny)
452.            dis(0:Ny) = dis_new(0:Ny)
```

```
453.            DEALLOCATE(a,b,c,r,x)
454.
455.        END SUBROUTINE GETDIS
```

Poisson Equation

Considering this equation and its boundary conditions

$$\frac{d^2\phi}{dx^2} = f \tag{1.1}$$

$$\phi(0) = 0, \phi(1) = 1 \tag{1.2}$$

Considering the effects of number nodes on the calculation results

Uniform grid and Error Analysis Using Matlab

(1) $f = e^{-x}$

Integrate the Eq(1.1) once , we could obtain

$$\frac{d\phi}{dx} = -e^{-x} + C_1 \tag{1.3}$$

Integrate the Eq(1.3), we could obtain

$$\phi = -e^{-x} + C_1 x + C_2 \tag{1.4}$$

Considering the boundary conditions,

$$\phi(0) = -e^{-0} + C_2 = 0 \tag{1.5}$$

$$\phi(1) = -e^{-1} + C_1 + C_2 = 1 \tag{1.6}$$

$$\Rightarrow C_1 = 2 - e^{-1}, C_2 = -1 \tag{1.7}$$

Then we obtain the exact solution of this equation

$$\phi = e^{-x} + \left(2 - e^{-1}\right)x - 1 \tag{1.8}$$

From the Taylor expansion

$$f(x) = f(x_0) + (x - x_0)\frac{df(x)}{dx} + \frac{(x - x_0)^2}{2}\frac{d^2 f(x)}{dx^2} + \frac{(x - x_0)^3}{6}\frac{d^3 f}{dx^3} + .. \tag{1.9}$$

We obtain

$$\phi(i+1)=\phi(i)+(\Delta x)\frac{d\phi}{dx}+\frac{(\Delta x)^2}{2}\frac{d^2\phi}{dx^2}+\frac{(\Delta x)^3}{6}\frac{d^3\phi}{dx^3}+.. \qquad (1.10)$$

$$\phi(i-1)=\phi(i)-(\Delta x)\frac{d\phi}{dx}+\frac{(\Delta x)^2}{2}\frac{d^2\phi}{dx^2}-\frac{(\Delta x)^3}{6}\frac{d^3\phi}{dx^3}+.. \qquad (1.11)$$

Eq(1.10) plus Eq(1.11), we obtain

$$\phi(i+1)+\phi(i-1)=2\phi(i)+2\frac{(\Delta x)^2}{2}\frac{d^2\phi}{dx^2}+\mathrm{O}(\Delta x)^4.. \qquad (1.12)$$

$$\left(\frac{d^2\phi}{dx^2}\right)_i=\frac{\phi(i+1)-2\phi(i)+\phi(i-1)}{(\Delta x)^2}+o(\Delta x)^2 \quad (1.13)$$

By approximation,

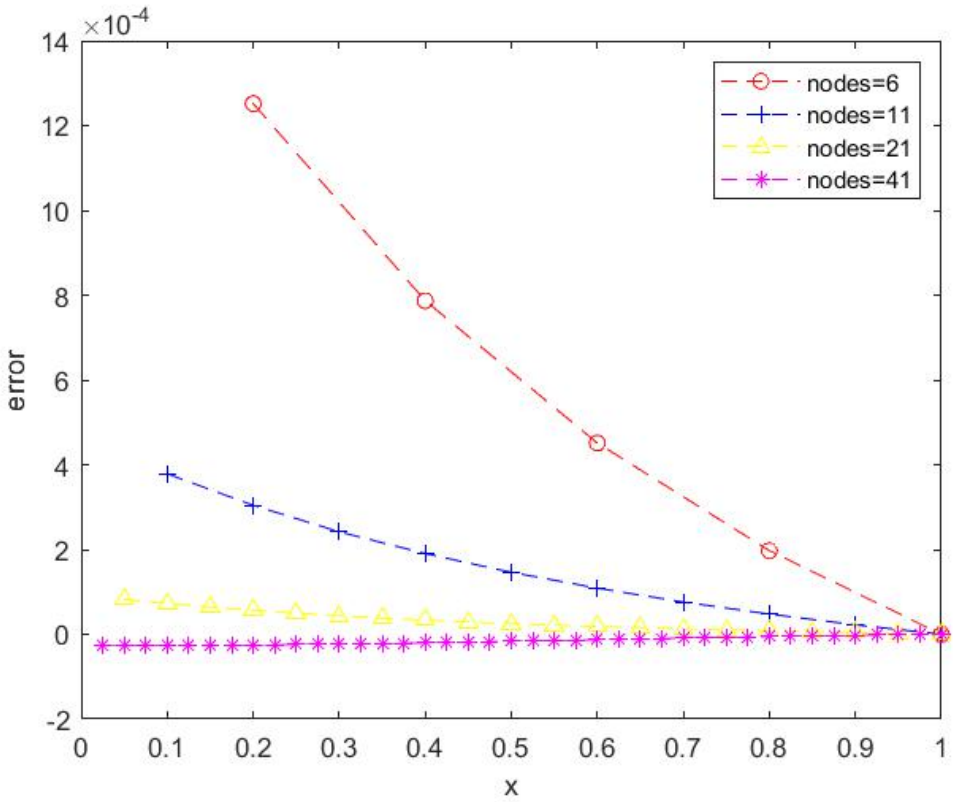
$$\phi(i+1)-2\phi(i)+\phi(i-1)=e^{-x_i}(\Delta x)^2 \quad (1.14)$$

$$\phi(i)=\frac{1}{2}\left(\phi(i+1)+\phi(i-1)-e^{-x_i}(\Delta x)^2\right) (1.15)$$

Calculate the error under 6,11,21,41 nodes using Matlab.
We found that the more nodes, the more accurate results could be obtained.

```
clc;
Domain_length=1;
nodes=[6, 11, 21, 41];
iter_error=1;
conver_error=1e-6;
shape=["r--o", "b--+", "y--^", "m--*"];
for n=1:4
    dx=zeros(1, 4);
    phi_old=zeros(1, nodes(n));
    phi_new=zeros(1, nodes(n));
    dx(n)=Domain_length/((nodes(n)-1));
    f=zeros(1, nodes(n));
    e=zeros(1, nodes(n));
    f(1) = 0;
    f(nodes(n))=1;
    phi_old(1)=0;
    phi_old(nodes(n))=1;
    phi_new(1)=phi_old(1);
    phi_new(nodes(n))= phi_old(nodes(n));
```

```
while iter_error>conver_error

    for i=2:(nodes(n)-1)
       phi_new(i)=0.5*(phi_old(i+1)+phi_old(i-1)-exp(-(i-1)*dx(n))*dx(n)^2);
    end
     iter_error=0;
     for i= 2:(nodes(n)-1)
           iter_error=iter_error + abs(phi_old(i)-phi_new(i));
     end
        phi_old=phi_new;
end
    for i=2:(nodes(n)-1)
        f(i)=exp(-(i-1)*dx(n))+(2-exp(-1))*(i-1)*dx(n)-1;
        e(i)=(phi_old(i)-f(i))/phi_old(i);
    end
    iter_error=1;
    x=((2:nodes(n))-1)*dx(n);
    plot(x,e(2:nodes(n)),shape(n));
    hold on;
end
    legend("nodes=6","nodes=11","nodes=21","nodes=41");
    xlabel('x');
    ylabel('error');
```



Non-uniform Grid and Error Analysis Using C

(2) considering

$$f = 2x - 1$$

Integrate the Eq(2.1) once , we could obtain

$$\frac{d\phi}{dx} = x^2 - x + C_1 \qquad (2.1)$$

Integrate the Eq(2.1) on both side,

$$\phi(x) = \frac{1}{3}x^3 - \frac{1}{2}x^2 + C_1 x - C_2 \qquad (2.2)$$

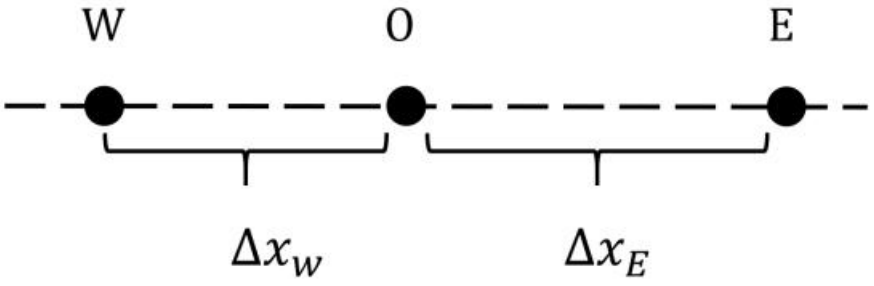
Substitute the Eq(1.2) into the Eq(2.2),

$$C_1 = \frac{7}{6}, C_2 = 0 \qquad (2.3)$$

So, the exact solution is

$$\phi(x) = \frac{1}{3}x^3 - \frac{1}{2}x^2 + \frac{7}{6}x \qquad (2.4)$$

Discretization: non-uniform grid in space



Using the Taylor expansion, We obtain the Eq(2.5) and Eq(2.6)

$$\phi_w = \phi_o + \frac{(-\Delta x_w)}{1!}\frac{d\phi}{dx} + \frac{(-\Delta x_w)^2}{2!}\frac{d^2\phi}{dx^2} + \frac{(-\Delta x_w)^3}{3!}\frac{d^3\phi}{dx^3} + \frac{(-\Delta x_w)^4}{4!}\frac{d^4\phi}{dx^4} + o(\Delta x^5)$$

$$\phi_E = \phi_o + \frac{(\Delta x_E)}{1!}\frac{d\phi}{dx} + \frac{(\Delta x_E)^2}{2!}\frac{d^2\phi}{dx^2} + \frac{(\Delta x_E)^3}{3!}\frac{d^3\phi}{dx^3} + \frac{(\Delta x_E)^4}{4!}\frac{d^4\phi}{dx^4} + o(\Delta x^5)$$

Using Δx_E multiple Eq(2.5) plus Δx_w multiple Eq(2.6), we can remove the first order derivative,

$$\Delta x_E \phi_W + \Delta x_W \phi_E = (\Delta x_E + \Delta x_W)\phi_O + \frac{\Delta x_E \Delta x_W (\Delta x_E + \Delta x_w)}{2}\frac{d^2\phi}{dx^2} + o(\Delta x^4) \quad (2.7)$$

$$\frac{d^2\phi}{dx^2} = \frac{2\phi_E}{\Delta x_E(\Delta x_E + \Delta x_W)} - \frac{2\phi_O}{\Delta x_E \Delta x_W} + \frac{2\phi_W}{\Delta x_W(\Delta x_E + \Delta x_W)} - \frac{\Delta x_E - \Delta x_W}{3}\frac{d^3\phi}{dx^3} - \frac{\Delta x_E{}^2 - \Delta x_E \Delta x_W + \Delta x_W{}^2}{12}\frac{d^4\phi}{dx^4} + o(\Delta x^3) \quad (2.8)$$

If $\Delta x_E = \Delta x_W = \Delta x$, it becomes uniform grid

For uniform grid, $\sum_{i=1}^{N} \Delta x_i = L$, N represents the number of nodes.

Assume $\Delta x_{i+1} = s\Delta x_i$ and satisfy the function

$$\frac{s - s^N}{1 - s} = \frac{L}{\Delta x_0} \quad (2.9)$$

We set s=1.02, substitute the formula $\Delta x_E = s\Delta x_W$ into Eq(2.8)

$$\left(\frac{d^2\phi}{dx^2}\right)_{x=o} = \frac{2\phi_E}{s(1+s)\Delta x_w{}^2} - \frac{2\phi_o}{s\Delta x_w{}^2} + \frac{2\phi_w}{(1+s)\Delta x_w{}^2} = f \quad (2.10)$$

Using the matrix form,

$$\phi_E - (1+s)\phi_o + s\phi_w = f(1+s)s\frac{\Delta x_w{}^2}{2} \quad (2.11)$$

The Dirichlet boundary conditions are given, so we could obtain the matrix form,

$$\begin{bmatrix} 1 & & & & & & \\ s & -(1+s) & 1 & & & & \\ & s & -(1+s) & 1 & & & \\ & & \cdots & \cdots & \cdots & & \\ & & & s & -(1+s) & 1 & \\ & & & & & 1 & \end{bmatrix} \begin{bmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \\ \cdots \\ \phi_{N-2} \\ \phi_{N-1} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_{\cdots} \\ f_{N-2} \\ f_{N-1} \end{bmatrix}$$

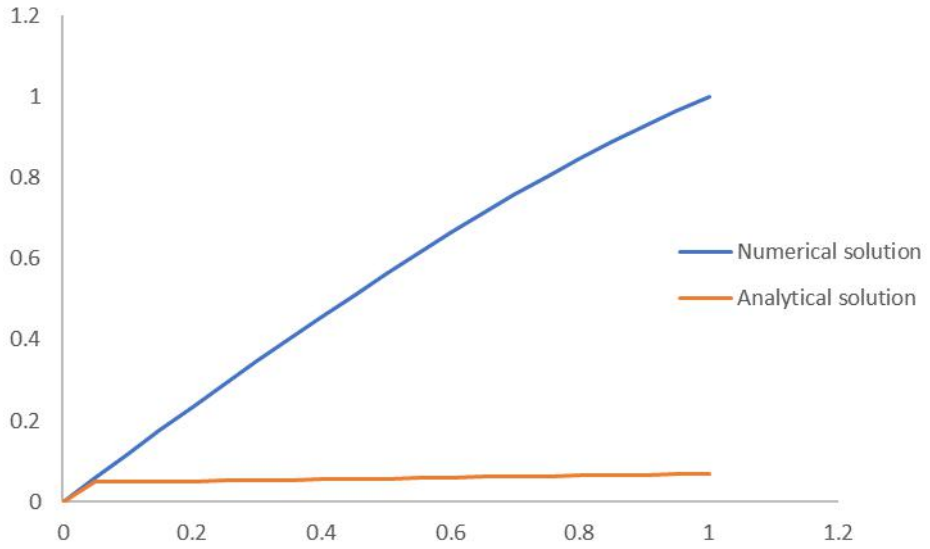


Figure 2.1 Numerical solution and analytical solution

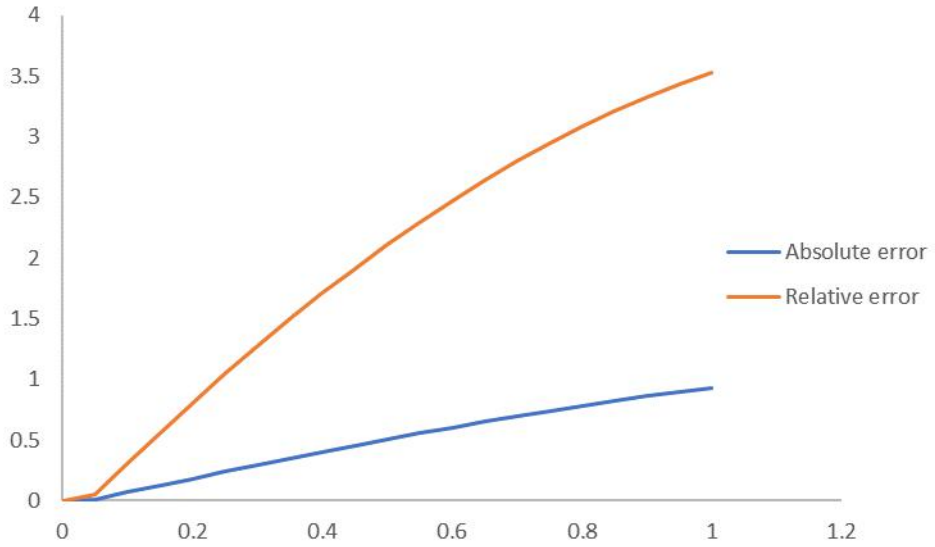


Figure 2.2 Absolute error and relative error

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>


int coordinate(const int x, const int y,const int N){
return (x+y*N);
}

void Gaussian_Elimination(double* A, double* B, double* res, const int N){
for (int j=0;j<N-1;++j){
    for(int i=j+1;i<N;++i){
```

```
        double f_eli=A[coordinate(j,i,N)]/A[coordinate(j,j,N)];
        B[i]=B[i]-f_eli*B[j];
        for(int k=0;k<N;++k){
A[coordinate(k,i,N)]=A[coordinate(k,i,N)]-f_eli*A[coordinate(k,j,N)]
;
        }
    }
}
for(int j=N-1;j>-1;--j){
    res[j]=B[j];
    for(int i=j+1;i<N;++i){
        if(i!=j){
            res[j]-=A[coordinate(i,j,N)]*res[i];
        }
    }
    res[j]=res[j]/A[coordinate(j,j,N)];
}
}


int main()
{


    const double L=1.0;
    const int N =21;
    //*******NON UNIFORM MESH*************
    const double s = 1.02;
    const double x0=L*(1-s)/(1-pow(s,N-1));
    //UNIFORM MESH
    //const double s =1.00;
    //const double x0=L/(N-1);
    printf("the first dx is:x0 = %f\n",x0);

    double* Coefficient_Matrix=(double*) malloc(N*N*sizeof(double));
    double* f=(double*) malloc((N)*sizeof(double));
    double* f_x = (double*) malloc((N)*sizeof(double));
    double* dx=(double*) malloc((N-1)*sizeof(double));

    double* phi_num = (double*) malloc((N)*sizeof(double));
    double* phi_ana = (double*) malloc((N)*sizeof(double));
    //***************INITIAL CONDITION************
    dx[0] =x0;
    phi_ana[0]= 0;
```

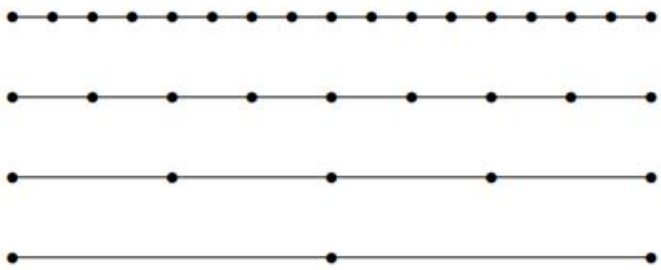
```
    f_x[0]=0;
    f[0]= 2*f_x[0]-1;


    FILE *fp;
    fp=fopen("abs_error1.csv","w");
    FILE *fo;
    fo=fopen("rela_error1.csv","w");
    FILE *fr;
    fr=fopen("phi_ana.csv1","w");
    FILE *fs;
    fs=fopen("phi_num.csv1","w");

    //*********exact solution************
    for(int i =1;i<N;++i){
        if(i<N-1){
            dx[i] = s*dx[i-1];
        }
        f_x[i]= dx[i-1]*s;

        phi_ana[i]= 1.0/3.0*f_x[i]*f_x[i]*f_x[i]-1.0/2.0*f_x[i]*f_x[i]+7.0/6.0*f_x[i];
        f[i]= 2*f_x[i]-1;

    }
    //**************construct coefficient matrix of Dirichlet boundary condition
    for(int j=1;j<N-1;++j){

        for (int i=0;i<N; ++i){

            const int coordinate_ = coordinate(i,j,N);
            Coefficient_Matrix[coordinate_]=0;
            if(i==j){
                Coefficient_Matrix[coordinate_]=-(1.0+s);

            }
            if(i+1==j){
                Coefficient_Matrix[coordinate_]=s;

            }
            if(i-1==j){
                Coefficient_Matrix[coordinate_]=1.0;
            }

```

```
99.         }
100.    }
101.    for(int i=0;i<N;++i){
102.        const int coordinate_T=coordinate(i,0,N);
103.        const int coordinate_B= coordinate(i,N-1,N);
104.        Coefficient_Matrix[coordinate_T]=0;
105.        Coefficient_Matrix[coordinate_B]=0;
106.    }
107.    Coefficient_Matrix[coordinate(0,0,N)]=1;
108.    Coefficient_Matrix[coordinate(N-1,N-1,N)]=1;
109.
110.    for(int i=1;i<N-1;++i){
111.        f[i] *=dx[i-1]*dx[i-1]*s*(1+s)/2.0;
112.    }
113.    f[0]=0;
114.    f[N -1]=1.0;
115.    Gaussian_Elimination(Coefficient_Matrix,f,phi_num,N);
116.    //************OUTPUT RESULTS***********
117.    //std::ostringstream name;
118.   // name<<N<<"NODES_RESULT"<<".dat";
119.    //std::ofstream fout(name.str().c_str());
120.
121.    double* abs_err = (double*) malloc((N)*sizeof(double));
122.    double* rela_err = (double*) malloc((N)*sizeof(double));
123.
124.    for (int i=0;i<N;++i){
125.        abs_err[i]=fabs(phi_ana[i]-phi_num[i]);
126.
   rela_err[i]=sqrt((phi_ana[i]-phi_num[i])*(phi_ana[i]-phi_num[i])/phi
   _ana[i]);
127.        printf("i = %d\n",i);
128.        printf("abs_err = %f\n",abs_err[i]);
129.        printf("rela_err = %f\n",rela_err[i]);
130.        printf("phi_ana = %f\n",phi_ana[i]);
131.        printf("phi_num = %f\n",phi_num[i]);
132.
133.
134.        fprintf(fp,"%f\n",abs_err[i]);
135.        fprintf(fo,"%f\n",rela_err[i]);
136.        fprintf(fr,"%f\n",phi_ana[i]);
137.        fprintf(fs,"%f\n",phi_num[i]);
138.    }
139.    fclose(fp);
140.    fclose(fo);
141.    fclose(fr);
```

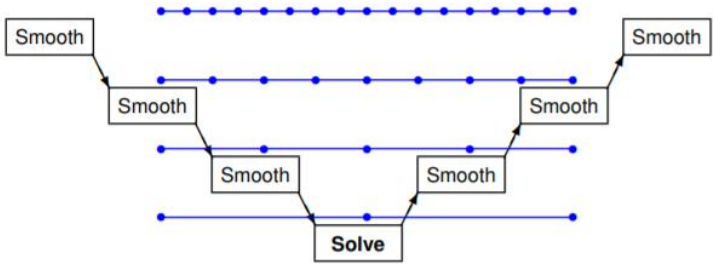
```
142.     fclose(fs);
143.     free(abs_err);
144.     free(rela_err);
145.
146.
147.     free(Coefficient_Matrix);
148.     free(f);
149.     free(f_x);
150.     free(dx);
151.     free(phi_ana);
152.     free(phi_num);
153.      printf("Hello world!\n");
154.     return 0;
155.
156. }
```

Multi-grid Method with python

Multi 1d grid



The V-cycle



What is needed for MG?

1. Sequence of grids
2. Intergrid transfer operators
3. Smoothing operator
4. Solver for coarsest grid

Geometrical multigrid

- Simple iterative methods tend to damp high (spatial) frequency errors fast.
- After a few smoothing steps of a simple method, map the current error out to a coarser grid.
- Errors will have relatively higher spatial frequency there.
- Take a few more steps of a simple method on the coarser grid.
- Continue mapping to coarser grids until grid is coarse enough to solve.
- Interpolate back to the next finer grid and do few smoothing steps
- Continue to the finest grid
- Repeat until converged.

Interpolation or prolongation If a solution is known on a grid, how should it be transferred to the next finer grid?
For fine grid points that agree with coarse points, copy.
For fine grid points between two coarse points, average.
Interpolation matrix 5 pts to 9 pts

$$P_{9\times5} = \begin{pmatrix} 1 & & & & \\ .5 & .5 & & & \\ & 1 & & & \\ & .5 & .5 & & \\ & & 1 & & \\ & & .5 & .5 & \\ & & & 1 & \\ & & & .5 & .5 \\ & & & & 1 \end{pmatrix}$$

The V-cycle steps
1. pre-smoothing

$Ax = b$ (x:kx1)

$r^k = b - Ax^k$ (A:kXk)

2.Restrictive

$\mathrm{r}_{2h} = Rr_h$

$$\begin{bmatrix} r_{2h}^1 \\ r_{2h}^2 \\ r_{2h}^3 \end{bmatrix} = [R]_{3X7} \begin{bmatrix} r_h^1 \\ r_h^2 \\ r_h^3 \\ r_h^4 \\ r_h^5 \\ r_h^6 \\ r_h^7 \end{bmatrix}$$

3.Solve e

$A_{2\mathrm{h}} e_{2h} = r_{2h}$

$[A_{2\mathrm{h}}]_{(\mathrm{k}-1)/2X(\mathrm{k}-1)/2} = R[R]_{(\mathrm{k}-1)/2X\mathrm{k}}[A_h]_{kXk}[R^T]_{kX(\mathrm{k}-1)/2}$

4.Prolongation

$\mathrm{e}_h = R^T e_{2h}$

5 Post-smoothing

$\mathrm{x}_{\mathrm{k}} = x_k + e_h$

Calculate the residual error

$r = b - Ax$

If $r = b - Ax_k < \varepsilon$ stop

If $r = b - Ax_k > \varepsilon$ continue to iterate

Calculate the residual error

$r = b - Ax_{2k}$

If the $r = b - Ax_{2k} > \varepsilon$

Continue the first step for x_{3k}

Model Problem:

$\phi'' = \sin x, \phi(0) = 0, \phi(pi) = 0$

Analytical Solution

$\phi(x) = -\sin(x) + x\sin(x[-1])$

Relaxation

Recall from our lecture on derivatives that a second-order accurate difference for the second derivative is:

$$\phi'' = \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2}$$

Our 1-d Poisson equation becomes:

$$\frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} = f_i$$

Solve for a single zone:

$$\phi_i = \frac{1}{2}\left(\phi_{i+1} + \phi_{i-1} - \Delta x^2 f_i\right)$$

Instead of a direct matrix solve, we'll use an iterative method

Jacobi iteration

Pick initial guess: $\phi_i^{(0)}$

Improve the guess through relaxation:

$$\phi_i^{k+1} = \frac{1}{2}\left(\phi^k{}_{i+1} + \phi^k{}_{i-1} - \Delta x^2 f_i\right)$$

Assess the error, if needed iterate

There we wrote our system as:

$$\begin{array}{ccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \ldots & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \ldots & a_{2n}x_n & = & b_2 \\ & & & & \vdots & & & \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \ldots & a_{nn}x_n & = & b_n \end{array}$$

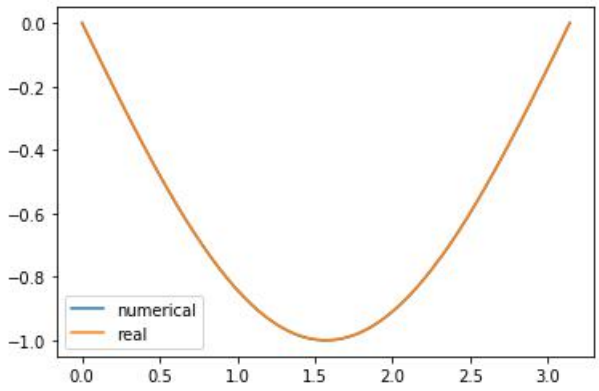
And iterated as:

$$\begin{array}{ccccccccc} x_1^{(k+1)} & = & -\frac{1}{a_{11}}(& a_{12}x_2^{(k)} & + & a_{13}x_3^{(k)} & \ldots & a_{1n}x_n^{(k)} & - \quad b_1) \\ x_2^{(k+1)} & = & -\frac{1}{a_{22}}(& a_{21}x_1^{(k)} & + & a_{23}x_3^{(k)} & \ldots & a_{2n}x_n^{(k)} & - \quad b_2) \\ & & & & \vdots & & & & \\ x_n^{(k+1)} & = & -\frac{1}{a_{nn}}(& a_{n1}x_1^{(k)} & + & a_{n2}x_2^{(k)} & \ldots & a_{n,n-1}x_{n-1}^{(k)} & - \quad b_n) \end{array}$$

We still need to define the error that we are taking the norm of – For our test problems, we can compare to the analytic solution, but that's not general – Only other measure: how well we satisfy the discrete equation—this is the residual

$$r_i \equiv f_i - \frac{\phi_{i+1}^k - 2\phi_i^k + \phi_{i-1}^k}{\Delta x^2}$$

Result



code

```
import math
import numpy as np
import matplotlib.pyplot as plt

def coarsen2(grid0):
    #dstep = int(2**(level-1))
    n_grid = int((grid0.size-1)/2 + 1) #int((grid0.size-1)/dstep + 1)
    grid1 = np.zeros(n_grid,  dtype  =  float)

    for i in range(1, grid1.size-1):
        i2 = 2*i
        grid1[i] = 0.5*grid0[i2] + 0.25*grid0[i2-1] + 0.25*grid0[i2 + 1]

    grid1[0] = grid0[0]
    grid1[-1] = grid0[-1]
    return grid1

def h_level(level):
    h0 = h
    dstep = int(2**(level-1))
    h2_grid = int((h0.size)/dstep)
    h2 = np.zeros(h2_grid,  dtype  =  float)
    for  i in range(0, h2.size):
        j0 = dstep*i
        h2_step = 0.
        for j in range(dstep):
            h2_step = h2_step + h0[j0 + j]
        h2[i] = h2_step
    return h2

def Relax2( b,  phi,  h, level_total, leveli):
    om = 1.85
```

```
    ite = 4**leveli      # the iteration increase with the level to get higher
precision
    j  =  0
    print("relax method")
    n2 = int(b.size/2)        #   choose the middle point to compare the err
in each iteration
    while(j <ite): #控制迭代次数
        i  =  1                  # when the boundary is known,  set i  =  1
        while( i < phi.size-1):
            phi_1 = np.copy(phi[i])
            phi[i]  =   om*0.5*(phi[i  +  1]  +  phi[i-1]-(h[i]*h[i])*b[i])  +
(1.-om)*phi_1# Inhomogeneous grid
            i  =  i + 1
        j  =  j + 1
    return phi

def residual(f, v, h):
    r = np.zeros(f.size,  dtype  =  float)
    for i in range(1, v.size-1):
        r[i]   =   f[i]-2*(v[i   +   1]-2.*v[i]   +   v[i-1])/(h[i]*h[i]   +
h[i-1]*h[i-1])#calculate the residual
    return r

def fine(grid_orig, level): # h is the step of the fine grid
    h = h_level(level)#      要被处理的网格所在的 level
    h2 = h_level(level-1)    #需要生成的网格所在的 level
    n_fine =  (grid_orig.size-1)*2 + 1
    grid_fine  =  np.zeros(n_fine,  dtype  =  float)

    for i in range(0, grid_orig.size-1):

        i2 = int(i*2)
        grid_fine[i2] = grid_orig[i]
        grid_fine[i2 + 1] = grid_orig[i]*(h2[i2 + 1]/h[i]) + grid_orig[i +
1]*(h2[i2]/h[i])
    grid_fine[-1] = grid_orig[-1]
    return grid_fine

def MG(phi, x, b, h):
    level_total  =  int(math.log2(phi.size-1))-5   #at  least  2**5  grids  to
capture small fluctuation
    print(level_total)
    h0 = h_level(1)
    vh = Relax2(b, phi, h, level_total, 1)
    rh = residual(b, vh, h0)
```

```
    eh =  np.zeros(b.size,  dtype  =  float) # 初始化

    for i in range (1, level_total):   #coarse and iterate
        print(i)
        h1  =  h_level(i + 1)
        r2h  =  coarsen2(rh)
        e2h0  =  coarsen2(eh)
        e2h  =  Relax2(r2h, e2h0, h1, level_total, i)
        v2h  =  coarsen2(vh) + e2h
        b2h = coarsen2(b)
        r2h = residual(b2h, v2h, h1)
        rh  =  r2h
        eh  =  e2h
        vh  =  v2h
        b = b2h

    for i in range (1, level_total):  # fine and itearate
        print(i)
        leveli = level_total-i + 1
        h1 = h_level(leveli)
        h2 = h_level(leveli-1)
        r2h  =  fine(rh, leveli)
        e2h0  =  fine(eh, leveli)
        e2h  =  Relax2(r2h, e2h0, h2, level_total, leveli)
        b2h = fine(b, leveli)
        v2h  =  fine(vh, leveli) + e2h
        r2h = residual(b2h, v2h, h2)
        rh  =  r2h
        eh  =  e2h
        vh  =  v2h
        b = b2h

    return vh


n_grid = 1025  #at least 512 grids to reach enough depth
xs = 0.0
xe = math.pi
h = (xe-xs)/(n_grid-1)* np.ones(n_grid-1,  dtype  =  float)

x = np.zeros(n_grid,  dtype  =  float)
x[0] = xs
x[-1] = xe
for i in range(1, n_grid-1):
    x[i] = x[i-1] + h[i-1]
```

```
phi  =  np.ones(x.size,  dtype  =  float)*0.1
phi[0] = 0.
phi[-1] = 0.#final point is zero

b = np.sin(x)
result2 = -np.sin(x)+x*np.sin(x[-1])   #precise solution
result =   MG(phi, x, b, h)


plt.figure(1)
plt.plot(x, result, label = 'numerical')
plt.plot(x, result2, label = 'real')
plt.legend()
plt.show()
plt.close()
```

Linear Multiple Step Method using Python

https://en.wikipedia.org/wiki/Linear_multistep_method
Taylor series expansion

$$\phi_{i+1} = \phi_i + (\phi_{i+1} - \phi_i)\frac{d\phi}{dx}\bigg|_{x=i} + \frac{(\phi_{i+1} - \phi_i)^2}{2!}\frac{d^2\phi}{dx^2}\bigg|_{x=i} + \frac{(\phi_{i+1} - \phi_i)^3}{3!}\frac{d^3\phi}{dx^3}\bigg|_{x=i} + \frac{(\phi_{i+1} - \phi_i)^4}{4!}\frac{d^4\phi}{dx^4}\bigg|_{x=i} + \ldots$$

Euler's Method

Euler's Method is the simplest Runge-Kutta method, which is

$$y_{n+1} = y_n + hf(t_n, y_n)$$

Given the initial value problem

$$y' = y, y(0) = 1$$

We would like to use the Euler method to approximate $y(t=1)$

Using the step size h=1/4

The analytical solution is $y(\mathrm{t}) = e^t$

The Euler method is $y_{n+1} = y_n + hf(t_n, y_n)$

$f(t_0, y_0) = f(0,1) = y' = y(0) = 1$

$y_{1/4} = y_0 + hf(t_0, y_0) = 1 + 1/4 * 1 = 1.25$

$y_{1/2} = y_1 + hf(t_{1/4}, y_{1/4}) = 1.25 + 1/4 * 1.25 = 1.5625$

$y_{3/4} = y_{1/2} + hf(t_{1/2}, y_{1/2}) = 1.5625 + 1/4 * 1.5625 = 2.34375$

$y_1 = y_{3/4} + hf(t_{3/4}, y_{3/4}) = 2.34375 + 1/4 * 2.34375 = 2.9296875$

$e^1 = 2.7182818284$

The forward Euler method tends to "overshot" the exact solution and Backward Euler method tends to "undershoot".

Stability

For example, the linear IVP given by

$y' = -ay$

With $y(0) = 1$ and a>0. the exact solution is $y' = e^{-ax}$

Applying the explicit forward Euler's method:

$y_{n+1} = y_n - ahy_n = (1 - ah)y_i = (1 - ah)^{i+1} y_0$

The solution is decaying (stable) if

$|1 - ah| < 1$

To prevent the amplification of the errors in the iteration process, we require, for stability of the forward Euler's method:

$h < \frac{2}{a}$

Backward Euler method using Newton's method

$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$

Use Newton's method to solve the implicit equation

$\frac{y_{n+1} - y_n}{\Delta x} = f(x_{n+1}, y_{n+1})$

Use Taylor series expansion

$f(x_{n+1}, y_{n+1}) = f(x_n, y_n) + \frac{y_{n+1} - y_n}{1!} \left.\frac{df}{dy}\right|_{y_n}$

Then we could obtain

$$\frac{y_{n+1}-y_n}{\Delta x}=f(x_n,y_n)+(y_{n+1}-y_n)\left.\frac{df}{dy}\right|_{y_n}$$

$$\left(\frac{1}{\Delta x}-\left.\frac{df}{dy}\right|_{y_n}\right)y_{n+1}=\frac{y_n}{\Delta x}+f(x_n,y_n)-y_n\left.\frac{df}{dy}\right|_{y_n}$$

Using this method we can obtain the approximation which is just a guess,
So we will iterate the formula many times as following algorithm:

for $i=1,2,3,...$

$$\frac{y_{n+1}^{(i)}-y_n}{\Delta x}=f^{i-1}(x_n,y_n)+\left(y_{n+1}^{(i)}-y_n^{(i-1)}\right)\left.\frac{df}{dy}\right|_{y_n}$$

Euler's Method is the simplest Runge-Kutta method.

Stability

Same IVP problem, we can obtain

$$y_{n+1}=\frac{1}{1+ah}y^i=\left(\frac{1}{1+ah}\right)^{i+1}y_0$$

The solution is decaying(stable) if $|1+ah|>1$

Runge-Kutta Method

Extended the Euler's method into Runge-kutta Method

$$y_{i+1}=y_i+h[\lambda_1K_1+\lambda_2K_2]$$

$$K_1=f(x_i,y_i)$$

$$K_2=f(x_i+ph,y_i+phK_1)$$

To determine the coefficients λ_1 and λ_2 , and achieve the second order accuracy,

$$R_i=y(x_{i+1})-y_{i+1}=o(h^3)$$

Step1:

$$K_2=f(x_i+ph,y_i+phK_1)$$
$$=f(x_i,y_i)+phf_x(x_i,y_i)+phK_1f_y(x_i,y_i)+o(h^2)$$
$$=y'(x_i)+phy''(x_i)+o(h^2)$$

Then we could obtain,

$$y_{i+1}=y_i+h\{\lambda_1y'(x_i)+\lambda_2[y'(x_i)+phy''(x_i)+o(h^2)]\}$$
$$=y_i+(\lambda_1+\lambda_2)hy'(x_i)+\lambda_2ph^2y''(x_i)+o(h^3)$$

Compare y_{i+1} and $y(x_{i+1})$ at x_i point in Taylor series expansion

$$y_{i+1} = y_i + (\lambda_1 + \lambda_2)hy'(x_i) + \lambda_2 ph^2 y''(x_i) + o(h^3)$$

$$y(x_{i+1}) = y(x_i) + hy'(x_i) + \frac{h^2}{2} y''(x_i) + o(h^3)$$

To satisfy $R_i = y(x_{i+1}) - y_{i+1} = o(h^3)$,

$$\lambda_1 + \lambda_2 = 1, \lambda_2 p = \frac{1}{2}$$

We obtain $\lambda_1 = \lambda_2 = \frac{1}{2}, p = 1$

To use this method, we could obtain the higher order accuracy.
The classical Runge-Kutta method:

$$y_{i+1} = y_i + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4)$$

$$K_1 = f(x_i, y_i)$$

$$K_2 = f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}K_1\right)$$

$$K_3 = f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}K_2\right)$$

$$K_4 = f(x_i + h, y_i + hK_3)$$

Adams-Bashforth Explicit Three-step Technique

The three steps Adams-Bashforth method is

$$y_{n+1} = y_n + \frac{h}{12}[23f(x_n, y_n) - 16f(x_{n-1}, y_{n-1}) + 5f(x_{n-2}, y_{n-2})]$$

Drawback of Multi-step Methods

One drawback of multistep methods is that because they rely on values of the solution from previous time steps, they cannot be used during the first time steps, because not enough values are available. Therefore, it is necessary to use a one-step method, with the same order of accuracy, to compute enough starting values of the solution to be able to use the multistep method. For example, to use the three-step Adams-Bashforth method, it is necessary to first use a one-step method such as the fourth-order Runge-Kutta method to compute y1 and y2, and then the Adams-Bashforth method can be used to compute y3 using y2, y1 and y0.

Fourth Order Adams-Moulton Formula

$$y_{n+1} = y_n + \frac{h}{24}[9f(x_{n+1}, y_{n+1}) - 19f(x_n, y_n) + 5f(x_{n-1}, y_{n-1})]$$

This is a implicit formula because y_{n+1} appears in $f(x_{n+1}, y_{n+1})$

Predictor-Corrector system

In fact, numerical analysts have attempted to achieve both simplicity and accuracy by combining the two formulas in what is called a predictor-corrector method. Once $y_{n-3}, y_{n-2}, y_{n-1}$ and y_n are known, we can compute $f_{n-3}, f_{n-2}, f_{n-1}$ and f_n, then use Adams-Bashforth to obtain value y_{n+1}. Then we compute f_{n+1} and use the Adams-Moulton (corrector) formula, which is implicit, to obtain an improved value of y_{n+1}, we can continue to use the corrector formula to change in y_{n+1} is too large. However, if it is necessary to use the corrector formula more than once or twice, it means that the step size h is too large and should be reduced.

In order to use any of multi-step methods it is necessary first to calculate a few y_j by some other method. For example, the fourth order Adams-Moulton method requires values for y_1 and y_2,while the fourth order Adams-Bsgforth method also requires a value for y_3. one way is to use a one-step method of of comparable accuracy to calculate the necessary starting values. Thus, for a fourth order multi-step method, one might use the fourth order Runge-Kutta method to calculate the starting values.

Another approach is to use a low order method with a very small **h** to calculate y_1 and then to increase gradually both the order and the step size until enough starting values have been determined.

Initial Value Problem and Code Using Python

Consider the ODE,

$$y' = -x^2 y^2 \left(0 \le x \le 1.5\right)$$

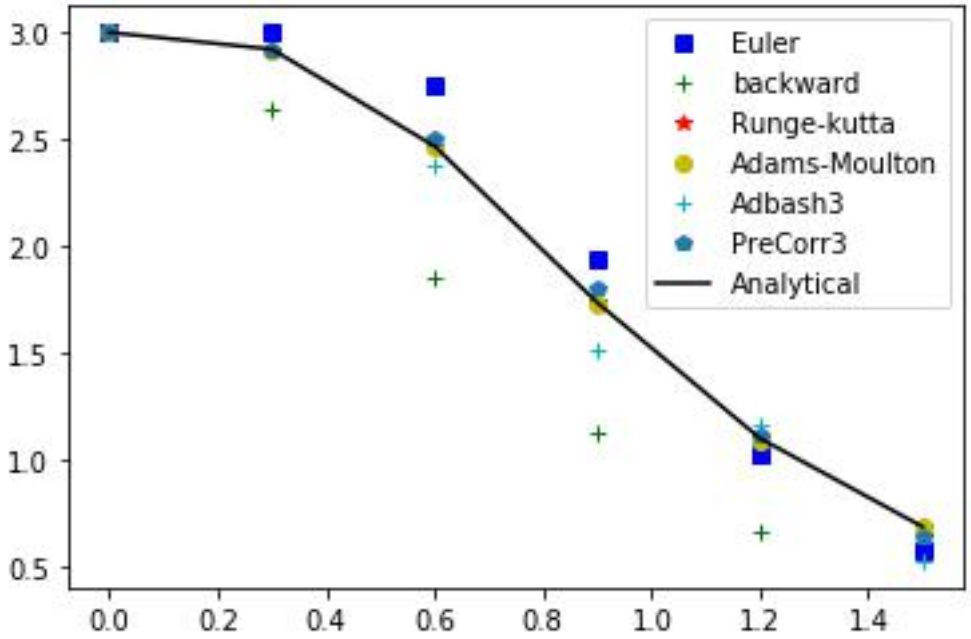
$$y(0) = 3$$

The exact solution is

$$y(x) = 3/\left(1 + x^3\right)$$

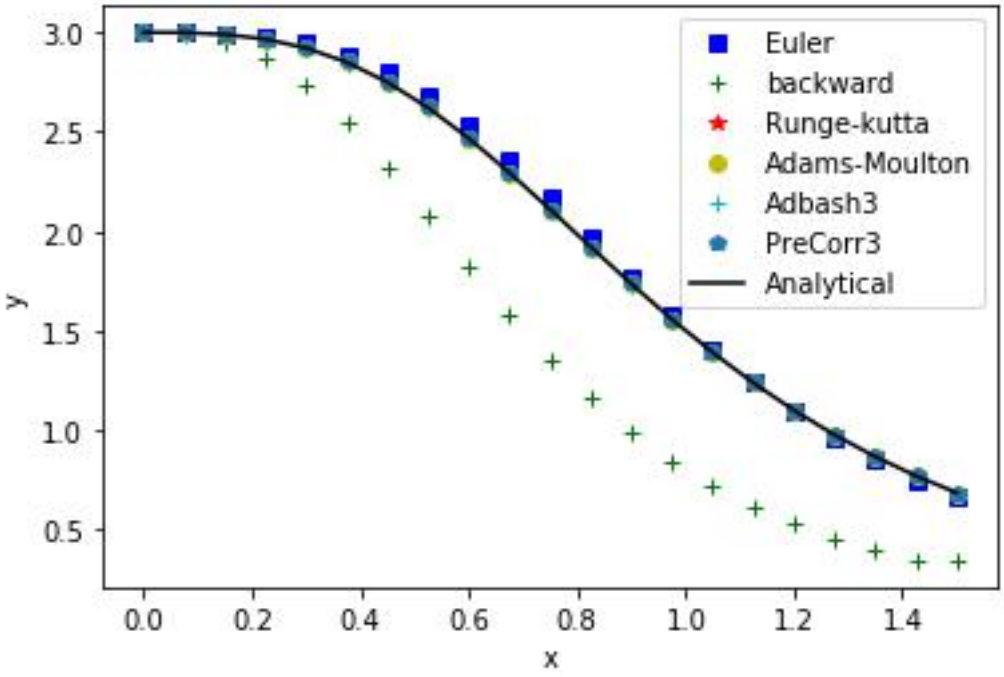
Results

When n=5, the results:



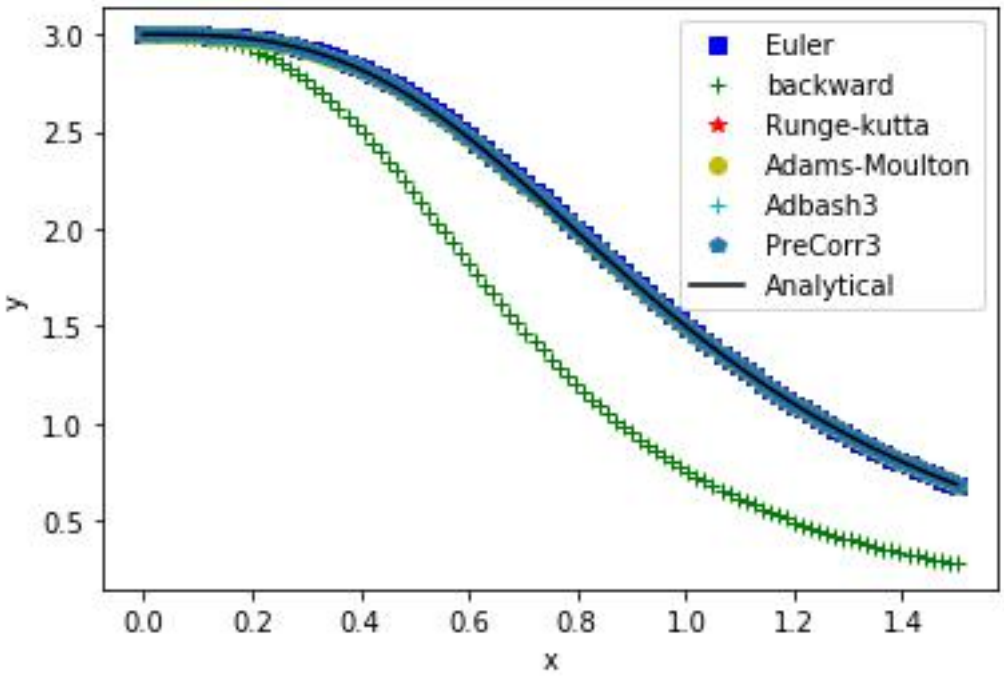
N=5	Total error	The value of Last point	Computational Time
Euler	0.7587932835368925	0.571154895251965	0.028s
Backward Euler	1.9556455857821415	0.6686336181984497	
Runge-Kutta	0.004375547302604921	0.6873916196103184	
Adams-Moulton	0.025281473969523738	1.7236088057055543	
Adams-Bashforth	0.5324652151821698	1.0877105441738686	
Predictor-Corrector	0.16296845527145987	0.6867335327458179	
Exact solution	0	0.6867335327458179	

N=20



N=20	**Total error**	**The value of Last point**	Computational Time
Euler	0.5873350395662289	0.666187327343244	0.029s
Backward Euler	9.555430542671823	0.3448899948504519	
Runge-Kutta	4.34298873956962e-05	0.6857197414764753	
Adams-Moulton	0.0005246104804780272	0.685725232889458	
Adams-Bashforth	0.028339716372171364	0.6862768528158518	
Predictor-Corrector	0.001692312765727566	0.6856536307705692	
Exact solution	0	0.6857142857142857	

N=100



N=100	Total error	The value of Last point	Computational Time
Euler	0.5644909793015918	0.6821451900700749	0. 036s
Backward Euler	48.37747420665726	0.2834897322043548	
Runge-Kutta	3.055268631424468e-07	0.685714293751177	
Adams-Moulton	4.5962141532740475e-06	0.6857143034526011	
Adams-Bashforth	0.0011949964639274002	0.6857176705912752	
Predictor-Corrector	1.2088866112480723e-05	0.6857142139818135	
Exact solution	0	0.6857142857142857	

Code

```
# -*- coding: utf-8 -*-
"""
Created on Sun Feb 14 17:40:22 2020

@author: Wentao Gong
"""
import math
import numpy as np
from numpy import *
```

```
10. import matplotlib.pyplot as plt
11. import time
12. 
13. #from matplotlib.patches import Patch
14. 
15. def Secant(y3,x3,y2,x2,y1,x1,y0,x0,h):
16.     eps=1.e-12
17.     i = 0
18.     dx0 = 0.1
19.     while(abs(dx0)>eps and i < 20):
20.          dfx0 = (differ_equa(y3+dx0,x3,y2,x2,y1,x1,y0,x0,h)-differ_e
    qua(y3,x3,y2,x2,y1,x1,y0,x0,h))/dx0
21.         if dfx0==0:
22.             #print('dfx0=0,y3=',y3)
23.             return y3
24.          y31 = y3-differ_equa(y3,x3,y2,x2,y1,x1,y0,x0,h)/dfx0
25.         dx0 = y31-y3
26.         y3 = y31
27.         i = i+1
28.      if abs(differ_equa(y3,x3,y2,x2,y1,x1,y0,x0,h))>10e-4:
29.         return 999999
30.     #print(y3)
31.     return y3
32. def differ_equa(y3,x3,y2,x2,y1,x1,y0,x0,h):
33.     fn3=-x3*x3*y3*y3
34.     fn2=-x2*x2*y2*y2
35.     fn1=-x1*x1*y1*y1
36.     fn0=-x0*x0*y0*y0
37.     f =  y3-y2-h/24.*(9.*fn3+19.*fn2-5.*fn1+fn0)
38.     return f
39. 
40. 
41. #Adams-Bashforth 3/Moulton 4 Step Predictor/Corrector
42. def PreCorr3(y0,a,b,n):
43.     h = (b-a)/n
44.     x = np.arange(a0, b0+h, h)
45.     y = np.zeros(x.size)
46.  #Calculate initial steps with Runge-Kutta 4
47.     y[0:3]=RK(y0, a, a+3*h, 3)
48.     K1=rhs(x[0],y[0],h)
49.     K2=rhs(x[1],y[1],h)
50.     for i in range(2, x.size,1):
51.         K3=K2
```

```
52.          K2=K1
53.          K1=rhs(x[i-1],y[i-1],h)
54.  #Adams-Bashforth Predictor
55.          y[i] = y[i-1] + h*(23*K1-16*K2+5*K3)/12
56.  #Adams-Moulton Corrector
57.          K0=rhs(x[i],y[i],h)
58.          y[i] = y[i-1] + h*(9*K0+19*K1-5*K2+K3)/24
59.
60.      return y
61.
62.
63.  def Adbash3(y0,a,b,n):
64.      h = (b-a)/n
65.      x = np.arange(a0, b0+h, h)
66.      y = np.zeros(x.size)
67.      y[0] = y0
68.
69.      y[0:2]=RK(y0, a, a+2*h, 2)
70.      K1=rhs(x[0],y[0],h)
71.      K2=rhs(x[1],y[1],h)
72.      for i in range(2, x.size,1):
73.          K3=K2
74.          K2=K1
75.          K1=rhs(x[i-1],y[i-1],h)
76.          y[i] = y[i-1] + h*(23*K1-16*K2+5*K3)/12
77.
78.      return y
79.
80.  def RK(y0,  a, b, n):
81.      h = (b-a)/n
82.      y = np.zeros(n)
83.      y[0] = y0
84.      for i in range(1, n, 1):
85.
86.          x0 = a+(i-1)*h
87.          k1 = rhs(x0, y0, h)
88.          k2 = rhs(x0+h/2., y0+h/2.*k1, h)
89.          k3 = rhs(x0+h/2., y0+h/2.*k2, h)
90.          k4 = rhs(x0+h, y0+h*k3, h)
91.          y0 = y0+h/6.*(k1+2.*k2+2.*k3+k4)
92.          y[i] = y0
93.          i = i+1
94.      return y
95.
96.  def Adams_Moulton(a0,b0,y00,h):
```

```
97.       x = np.arange(a0, b0+h, h)
98.       y = np.zeros(x.size)
99.       y[0] = y00
100.      n = 3
101.      y0= RK(y[0],  a0, a0+3.*h, n)
102.      y[0:3] = y0
103.      for i in range(n, x.size,1):
104.           y[i] = Secant(y[i],x[i],y[i-1],x[i-1],y[i-2],x[i-2],y[i-3],
    x[i-3],h)
105.      return y
106.
107.
108.
109. def Euler(y0,  a, b, n):
110.      h = (b-a)/n
111.      y = np.zeros(n)
112.      x = np.zeros(n)
113.      y[0] = y0
114.      for i in range(1, n, 1):
115.
116.          x[i-1] = a+(i-1)*h
117.          y[i] = y[i-1]+h*rhs(x[i-1],y[i-1],h)
118.          i = i+1
119.      return y
120.
121. def beuler(y0,  a, b, n):
122.      h = (b-a)/n
123.      y = np.zeros(n)
124.      x = np.zeros(n)
125.      y[0] = y0
126.      for i in range(1, n, 1):
127.          x[i-1] = a+(i-1)*h
128.          err=1
129.          iteration=0
130.          #y[i] = y[i-1]+h*rhs(x[i-1],y[i-1],h)
131.          #x[i] = a+(i)*h
132.         #Use Newton's Method to solve implicit equation for y[i]
133.           while err>10**(-4) and iteration<3:#NM is limited to 6 iter
    ations
134.                y_new=(y[i-1]/h+rhs(x[i-1],y[i-1],h)-y[i-1]*dfy(x[i-1],
    y[i-1]))/(1/h-dfy(x[i-1],y[i-1]))
135.                #y_new=(y[i]/h+rhs(x[i],y[i],h)-y[i]*dfy(x[i],y[i]))/(1
    /h-dfy(x[i],y[i]))
136.              #err=abs(y_new-y[i])
137.              err=abs(y_new-y[i-1])
```

```
            y[i-1]=y_new
            iteration +=1
        y[i]=y_new

    return y

def rhs(x, y, h):

    lanmb=-x*x*y
    f = lanmb*y
    if (lanmb*h < -2):
        print('h should smaller than ',  abs(2/lanmb),  h)
    return f
def dfy(x,y):
    return -2*x*x*y

start = time.clock()
a0 = 0.
b0 = 1.5
y0 = 3.
n = 100
h = 0.1
print('the value of last point')
y_euler = Euler(y0,  a0, b0+ (b0-a0)/n, n+1)
x_euler = np.arange(a0, b0+ (b0-a0)/n, (b0-a0)/n)
print(y_euler[-1])
y_be = beuler(y0,a0,b0,n+1)
x_be = np.arange(a0, b0+ (b0-a0)/n, (b0-a0)/n)
print(y_be[-1])
y_RK = RK(y0,  a0, b0+ (b0-a0)/n, n+1)
x_RK = np.arange(a0, b0+ (b0-a0)/n, (b0-a0)/n)
print(y_RK[-1])
y_Adams =Adams_Moulton(a0,b0,y0,(b0-a0)/n)
x_Adams = np.arange(a0, b0+ (b0-a0)/n, (b0-a0)/n)
print(y_Adams[-1])
y_Ab3 =Adbash3(y0,a0,b0,n)
x_Ab3 = np.arange(a0, b0+ (b0-a0)/n, (b0-a0)/n)
print(y_Ab3[-1])
y_p = PreCorr3(y0,a0,b0,n)
x_p = np.arange(a0, b0+ (b0-a0)/n, (b0-a0)/n)
print(y_p[-1])
y_analy=3./(1+x_RK**3)
print(y_analy[-1])
plt.figure(1)

```

```
#plt.plot(xx, yy)

print(x_euler.shape,  y_euler.shape)
print(x_be.shape,  y_be.shape)
print(x_RK.shape,  y_RK.shape)
print(x_Adams.shape,  y_Adams.shape)
print(x_Ab3.shape,  y_Ab3.shape)
print(x_p.shape,  y_p.shape)
print(x_RK.shape,  y_analy.shape)
#plt.plot(nn, ee)
#plt.plot(x_euler, y_A, 'bs',x_RK, y_p, 'r--',x_RK, y_analy, 'g^')

plt.plot(x_euler, y_euler, 'bs',x_be,y_be,'g+',x_RK, y_RK, 'r*',x_Adams,y_Adams,'yo',x_Ab3,y_Ab3,'c+',x_p,y_p,'p',x_RK, y_analy, 'black')
plt.legend(['Euler','backward','Runge-kutta','Adams-Moulton','Adbash3','PreCorr3','Analytical']);
plt.xlabel('x')
plt.ylabel('y')
#plt.scatter(xx,yyy)
print('the error')
delta_euler=np.sum(abs(y_analy-y_euler))
print(delta_euler)
delta_beuler=np.sum(abs(y_analy-y_be))
print(delta_beuler)
delta_RK=np.sum(abs(y_analy-y_RK))
print(delta_RK)
delta_AM=np.sum(abs(y_analy-y_Adams))
print(delta_AM)
delta_AB=np.sum(abs(y_analy-y_Ab3))
print(delta_AB)
delta_PC=np.sum(abs(y_analy-y_p))
print(delta_PC)

end = time.clock()
print('time=',end-start)

plt.show()
plt.savefig("out.jpg")
```

Boundary Value Problem

Classical ODE problems
Initial Value Problem (IVP) vs Boundary Value Problem (BVP)

（1)IVP equation

$$y''+ay'+by=0, y(0)=y_1, y'(0)=y_2$$

(2）BVP equation

$$y''+ay'+by=0, y(0)=y_1, y(L)=y_2$$

1D Reaction-Diffusion Equation

Forward Time Central Space(FTCS)- Dirichlet problem

Heat/diffusion equation is an example of parabolic differential equations. The general 1D form of heat equation is given by

$$u_t = Du_{xx} + f(u,x,t)$$

which is accompanied by initial and boundary conditions in order for the equation to have a unique solution. We approximate temporal- and spatial-derivatives separately. Using explicit or forward Euler method, the difference formula for time derivative is

$$u_t = \frac{u(x_i,t_{j+1})-u(x_i,t_j)}{\Delta t} + \mathrm{O}(t)$$

And the difference formula for spatial derivative is

$$u_{xx} = \frac{u(x_{i-1},t_j)-2u(x_i,t_j)+u(x_{i+1},t_j)}{\Delta x^2} + \mathrm{O}(\Delta x^2)$$

We consider a simple heat/diffusion equation of the form

$$u_t = Du_{xx}$$

that we want to solve in a 1D domain $0<x<L$ within time interval $0<t<T$.

The initial and boundary conditions are given by

$$u(x,0)=f(x)$$

$$u(0,\mathrm{t})=g_1(t)$$

$$u(L,\mathrm{t})=g_2(t)$$

Droping the terms $o(\Delta t), o(\Delta x^2)$, then we could obtain

$$\frac{U_{i,j+1}-U_{i,j}}{\Delta t} = D\frac{U_{i-1,j}-2U_{i,j}+U_{i+1,j}}{\Delta x^2}$$

For $i = 0,1,2,3...M$ and $j = 0,1,2,3...N$

Here, M and N are the number of grid points, are grid sizes (length of subintervals) along the t-axis and x-axis, respectively.
Letting

$$r = \frac{D\Delta t}{\Delta x^2}$$

Then, we get

$$U_{i,j+1} = \frac{D\Delta t}{\Delta x^2}\left(U_{i-1,j} - 2U_{i,j} + U_{i+1,j}\right) + U_{i,j}$$

$$U_{i,j+1} = rU_{i-1,j} + (1-2r)U_{i,j} + rU_{i+1,j}$$

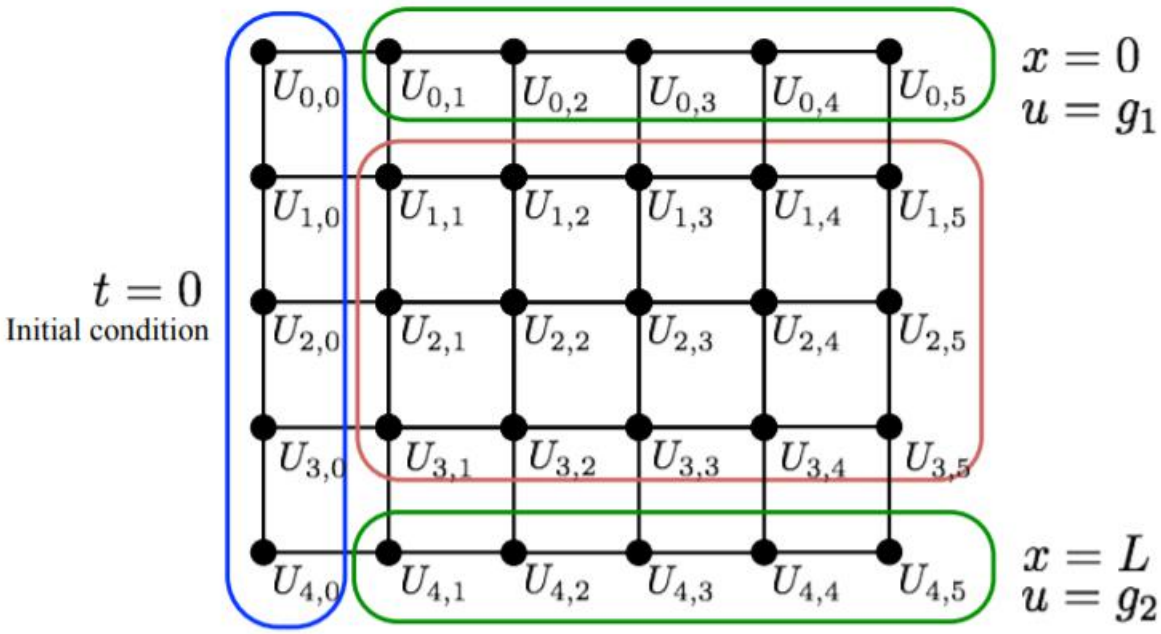
This explicit formula is decaying(stable) only if $r = \frac{D\Delta t}{\Delta x^2} \le \frac{1}{2}$

This means that the grid size Δt must satisfy

$$\Delta t \le \frac{\Delta x^2}{2D}$$

If this condition is not fulfilled, errors committed in one line $\{U_{i,j}\}$ may be magnified in subsequent lines $\{U_{i,p}\}$ for some $p > j$

Steps to solve the problem
Step 1: Define problem parameters such as - domain size - number of grid points (or subintervals) - grid size
Step 2: Define condition for stability, abort function if condition is not met
Step 3: Generate grids
Step 4: Initialize matrix for solution
Step 5: Fill in initial and boundary conditions
Step 6: Iteration/solve the linear algebraic equations
Step 7: Visualization
Grid to solve 1D heat equation with Dirichlet BC



Solve the following 1D heat/diffusion equation

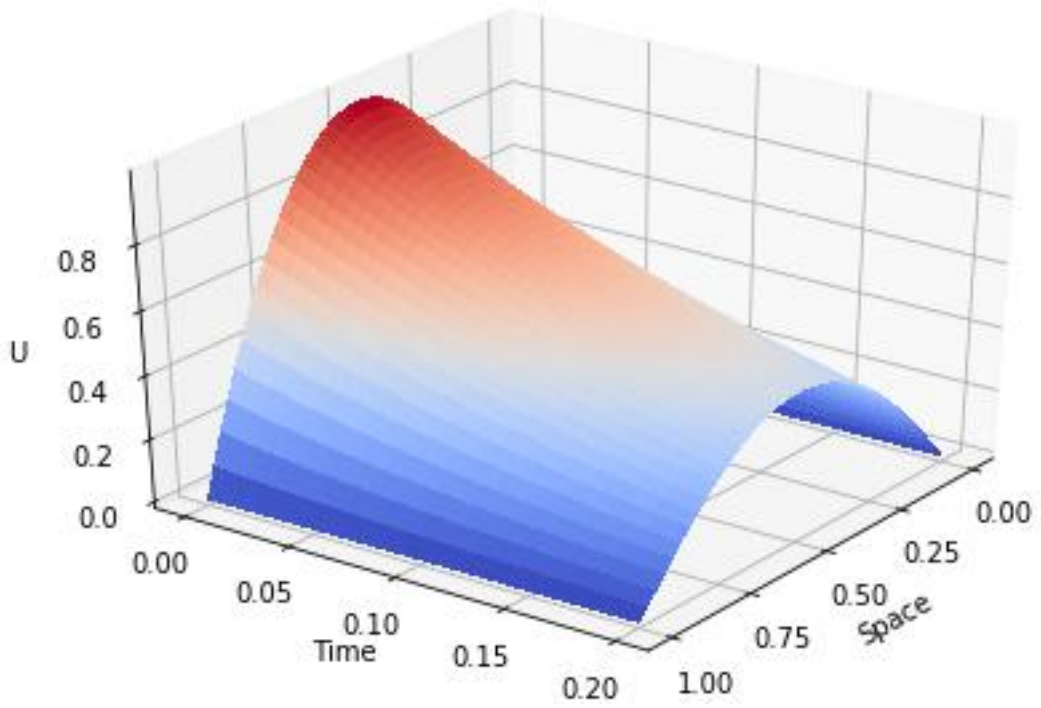
$u_t = u_{xx}$

in a unit domain $0 < x < L$ within time interval $0 < t < T$ subject to:

initial condition: $u(0, x) = 4x - 4x^2$

boundary condition: $u(t,0) = 0$ and $u(t,1) = 0$

and Approximate with explicit/forward finite difference method and use the following: M = 12 (number of grid points along x-axis)
N = 100 (number of grid points along t-axis)
Results:
r = 0.44086956521739135




```
1.  import numpy as np
2.  from scipy import sparse
3.  from mpl_toolkits.mplot3d import Axes3D
4.  import matplotlib.pyplot as plt
5.  from matplotlib import cm
6.
7.
8.  M = 40 # GRID POINTS on space interval
9.  N = 70 # GRID POINTS on time interval
10.
11. x0 = 0
12. xL = 1
13.
14. # ----- Spatial discretization step -----
15. dx = (xL - x0)/(M - 1)
16.
17. t0 = 0
18. tF = 0.2
19.
20. # ----- Time step -----
21. dt = (tF - t0)/(N - 1)
22.
```

```
D = 0.1  # Diffusion coefficient
alpha = -3 # Reaction rate

r = dt*D/dx**2
s = dt*alpha;

print(f'r = {r}')

# ----- Creates grids -----
xspan = np.linspace(x0, xL, M)
tspan = np.linspace(t0, tF, N)

# ----- Initializes matrix solution U -----
U = np.zeros((M, N))

# ----- Initial condition -----
U[:,0] = 4*xspan - 4*xspan**2

# ----- Dirichlet Boundary Conditions -----
U[0,:] = 0.0
U[-1,:] = 0.0

# ----- Equation (15.8) in Lecture 15 -----
for k in range(0, N-1):
    for i in range(1, M-1):
        U[i, k+1] = r*U[i-1, k] + (1-2*r+s)*U[i,k] + r*U[i+1,k]

T, X = np.meshgrid(tspan, xspan)

fig = plt.figure()
ax = fig.gca(projection='3d')

surf = ax.plot_surface(X, T, U, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)

ax.set_xticks([0, 0.25, 0.5, 0.75, 1.0])
ax.set_yticks([0, 0.05, 0.1, 0.15, 0.2])

ax.set_xlabel('Space')
ax.set_ylabel('Time')
ax.set_zlabel('U')
ax.view_init(elev=33, azim=36)
plt.tight_layout()
plt.show()
```

BTCS - Dirichlet Problem

The method is called implicit or backward Euler method. In this method the formula for time derivative is given by

$$u_t = \frac{u(x_i, t_j) - u(x_i, t_{j-1})}{\Delta t} + \mathrm{O}(t)$$

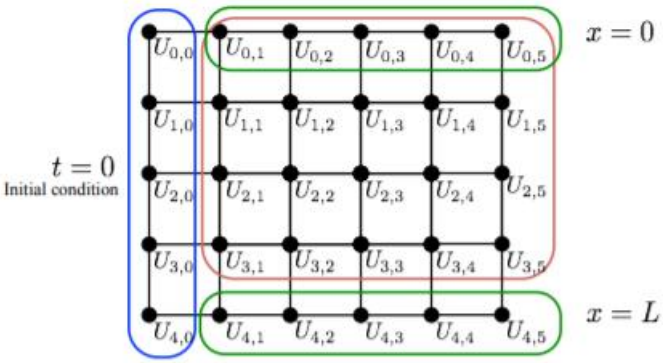
The approximation of heat equation (15.5) becomes

$$\frac{U(x_i, t_j) - U(x_i, t_{j-1})}{\Delta t} = \frac{U_{i-1,j} - 2U_{i,j} + U_{i+1,j}}{\Delta x^2}$$

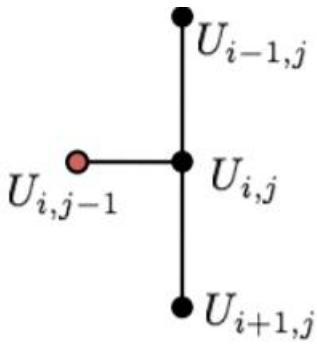
Letting $r = \frac{D\Delta t}{\Delta x^2}$

$$U_{i,j-1} = -rU_{i-1,j} + (1+2r)U_{i,j} - rU_{i+1,j}$$

Grid for 1D heat equation



The computational stencil represents the implicit method is illustrated as in the figure.



If the values at x end points are given from the Dirichlet type of boundary condition, the simulation equations as in

$$\begin{bmatrix} 1+2r & -r & & & \\ -r & 1+2r & -r & & \\ & \cdots & \cdots & \cdots & \\ & & -r & 1+2r-r & \\ & & & -r & 1+2r \end{bmatrix} \begin{bmatrix} U_{1,\mathrm{j}} \\ U_{2,\mathrm{j}} \\ \cdots \\ U_{M-3,j} \\ U_{M-2,j} \end{bmatrix} = \begin{bmatrix} rU_{0,j} & + & U_{1,j-1} \\ 0 & + & U_{2,j-1} \\ \cdots & + & \cdots \\ 0 & + & U_{M-3,j-1} \\ rU_{M-1,j} & + & U_{M-2,j-1} \end{bmatrix}$$

Solve the following 1D heat/diffusion equation

$u_t = u_{xx}$

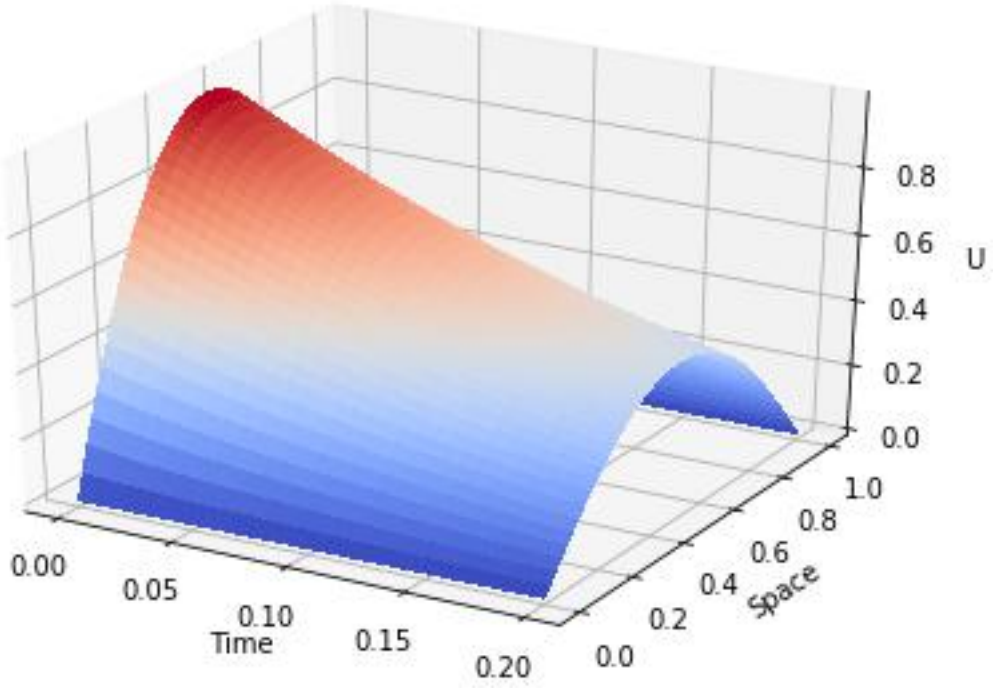
in a unit domain $0 < x < L$ within time interval $0 < t < T$ subject to:

initial condition: $u(0,x) = 4x - 4x^2$

boundary condition: $u(t,0) = 0$ and $u(t,1) = 0$

and Approximate with explicit/forward finite difference method and use the following: M = 50(number of grid points along x-axis)
N = 60(number of grid points along t-axis)

Results:



Code :

```
1.  import numpy as np
2.  from scipy import sparse
3.  from mpl_toolkits.mplot3d import Axes3D
4.  import matplotlib.pyplot as plt
```

```
from matplotlib import cm


M = 50 # GRID POINTS on space interval
N = 60 # GRID POINTS on time interval

x0 = 0
xL = 1

# ----- Spatial discretization step -----
dx = (xL - x0)/(M - 1)

t0 = 0
tF = 0.2

# ----- Time step -----
dt = (tF - t0)/(N - 1)

D = 0.1  # Diffusion coefficient
alpha = -3 # Reaction rate

r = dt*D/dx**2
s = dt*alpha;

xspan = np.linspace(x0, xL, M)
tspan = np.linspace(t0, tF, N)

main_diag = (1 + 2*r - s)*np.ones((1,M-2))
off_diag = -r*np.ones((1, M-3))

a = main_diag.shape[1]

diagonals = [main_diag, off_diag, off_diag]

A = sparse.diags(diagonals, [0,-1,1], shape=(a,a)).toarray()

# ----- Initializes matrix U -----
U = np.zeros((M, N))

#----- Initial condition -----
U[:,0] = 4*xspan - 4*xspan**2

#----- Dirichlet boundary conditions -----
U[0,:] = 0.0
U[-1,:] = 0.0
```

```
50.
51.  for k in range(1, N):
52.      c = np.zeros((M-4,1)).ravel()
53.      b1 = np.asarray([r*U[0,k], r*U[-1,k]])
54.      b1 = np.insert(b1, 1, c)
55.      b2 = np.array(U[1:M-1, k-1])
56.      b = b1 + b2  # Right hand side
57.      U[1:M-1, k] = np.linalg.solve(A,b)  # Solve x=A\b
58.
59.  # ----- Checks if the solution is correct:
60.  g = np.allclose(np.dot(A,U[1:M-1,N-1]), b)
61.  print(g)
62.
63.  # ----- Surface plot -----
64.  X, T = np.meshgrid(tspan, xspan)
65.
66.  fig = plt.figure()
67.  ax = fig.gca(projection='3d')
68.
69.  surf = ax.plot_surface(X, T, U, linewidth=0,
70.                         cmap=cm.coolwarm, antialiased=False)
71.
72.  ax.set_xticks([0, 0.05, 0.1, 0.15, 0.2])
73.
74.  ax.set_xlabel('Time')
75.  ax.set_ylabel('Space')
76.  ax.set_zlabel('U')
77.  plt.tight_layout()
78.  plt.show()
```

BTCS - Neumann Problem

If Neumann boundary condition is applied, where $\partial_x u = g$ at this type of boundary is approximated by

$$-\left(\frac{U_{i+1,j}-U_{i-1,j}}{2\Delta x}\right)=g_{i,j}$$

At $i=0$ the formula is rearranged to get

$$U_{-1,j}-U_{1,j}=2\Delta x g_{i,j}$$

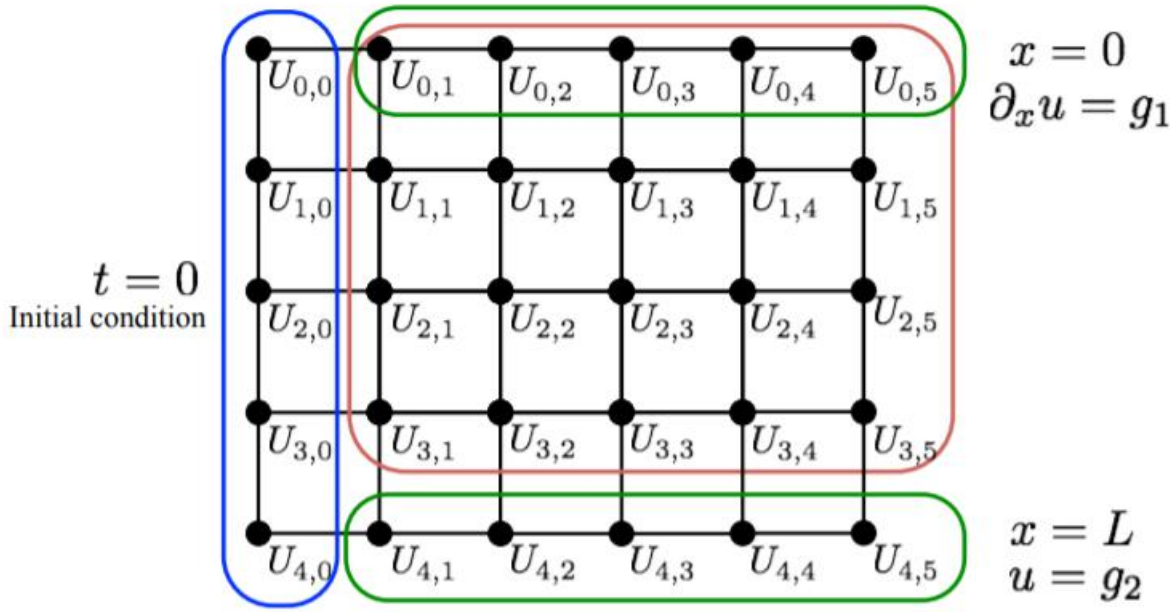
Hence along the x=0 axis, the approximation becomes

$$U_{0,j-1} = (1+2r)U_{0,j} - 2rU_{1,j} - 2r\Delta x g_{0,j}$$

And the simultaneous equations in matrix-vector notation:

$$\begin{bmatrix} 1+2r & -2r & & & & \\ -r & 1+2r & -r & & & \\ & \cdots & \cdots & \cdots & & \\ & & & 1+2r & -r & \\ & & & -r & 1+2r & \end{bmatrix} \begin{bmatrix} U_{0,j} \\ U_{1,j} \\ \cdots \\ U_{M-2,j} \\ U_{M-1,j} \end{bmatrix} = \begin{bmatrix} U_{0,j-1} + 2r\Delta x g_{0,j} \\ U_{1,j-1} \\ \cdots \\ U_{M-2,j-1} \\ U_{M-1,j-1} + rU_{M,j} \end{bmatrix}$$

Grid for 1D heat equation with Neumann + Dirichlet BCs



If both boundaries at x = 0 and x = L are equipped with Neumann boundary conditions:
Neumann boundary conditions:

$\partial_x u = g$ at x=0

$\partial_x u = f$ at x=L

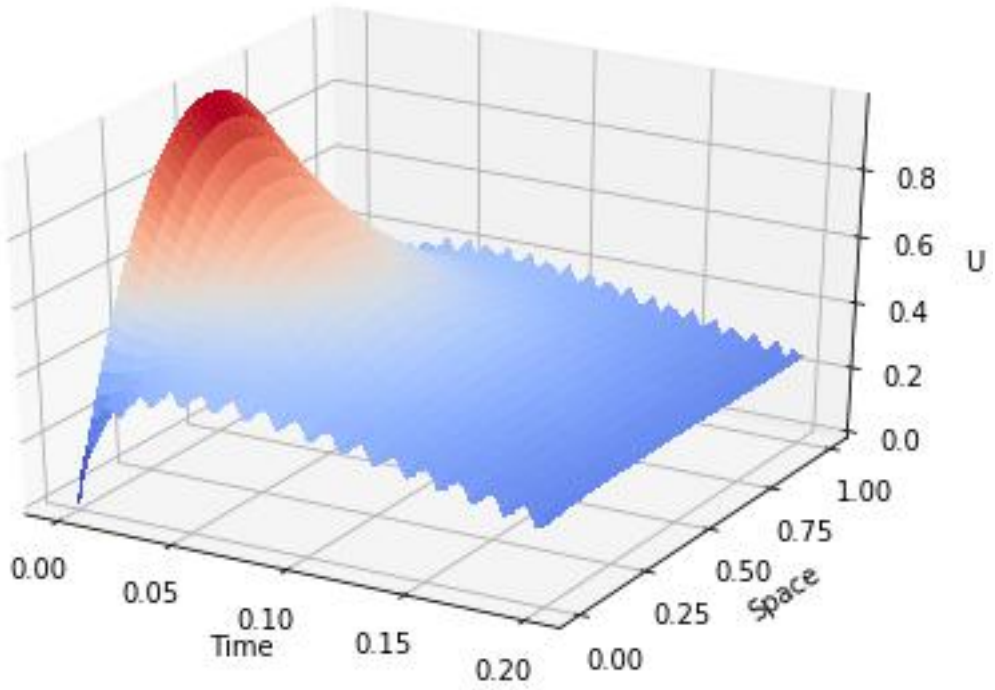
the simultaneous equations in matrix-vector notation:

$$\begin{bmatrix} 1+2r & -2r & & & & \\ -r & 1+2r & -r & & & \\ & \cdots & \cdots & \cdots & & \\ & & & 1+2r & -r & \\ & & & -2r & 1+2r & \end{bmatrix} \begin{bmatrix} U_{0,j} \\ U_{1,j} \\ \cdots \\ U_{M-2,j} \\ U_{M-1,j} \end{bmatrix} = \begin{bmatrix} U_{0,j-1} + 2r\Delta x g_{0,j} \\ U_{1,j-1} \\ \cdots \\ U_{M-2,j-1} \\ U_{M-1,j-1} + 2r\Delta x f_{M-1,j} \end{bmatrix}$$

Backward method to solve 1D reaction-diffusion equation:

u_t = D * u_xx + alpha * u

with Neumann boundary conditions
at x=0: u_x = sin(pi/2)
at x=L: u_x = sin(3*pi/4) with L=1
and initial condition u(x,0) = 4*x - 4*x**2




```
Created on  Feb 19 22:41:06 2020

@author: Wentao Gong
"""

'''''
Backward method to solve 1D reaction-diffusion equation:
    u_t = D * u_xx + alpha * u

with Neumann boundary conditions
at x=0: u_x = sin(pi/2)
at x=L: u_x = sin(3*pi/4) with L=1
and initial condition u(x,0) = 4*x - 4*x**2
'''

import numpy as np
from scipy import sparse
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm


M = 50 # GRID POINTS on space interval
```

```
N = 60 # GRID POINTS on time interval

x0 = 0
xL = 1

# ----- Spatial discretization step -----
dx = (xL - x0)/(M - 1)

t0 = 0
tF = 0.2

# ----- Time step -----
dt = (tF - t0)/(N - 1)

D = 0.5  # Diffusion coefficient
alpha = -5 # Reaction rate

r = dt*D/dx**2
s = dt*alpha
a = 1 + 2*r - s


xspan = np.linspace(x0, xL, M)
tspan = np.linspace(t0, tF, N)

main_diag = (1 + 2*r - s)*np.ones((1,M))
off_diag = -r*np.ones((1, M-1))

a = main_diag.shape[1]

diagonals = [main_diag, off_diag, off_diag]

A = sparse.diags(diagonals, [0,-1,1], shape=(a,a)).toarray()
A[0,1] = -2*r
A[M-1,M-2] = -2*r

# ----- Initializes matrix U -----
U = np.zeros((M, N))

#----- Initial condition -----
U[:,0] = 4*xspan - 4*xspan**2

#----- Neumann boundary conditions -----
leftBC = np.arange(1, N+1)
f = np.sin(leftBC*np.pi/2)
```

```
69.
70.  rightBC = np.arange(1, N+1)
71.  g = np.sin(3*rightBC*np.pi/4)
72.
73.
74.  for k in range(1, N):
75.      c = np.zeros((M-2,1)).ravel()
76.      b1 = np.asarray([2*r*dx*f[k], 2*r*dx*g[k]])
77.      b1 = np.insert(b1, 1, c)
78.      b2 = np.array(U[0:M, k-1])
79.      b = b1 + b2  # Right hand side
80.      U[0:M, k] = np.linalg.solve(A,b)  # Solve x=A\b
81.
82.  # ----- Checks if the solution is correct:
83.  gc = np.allclose(np.dot(A,U[0:M,N-1]), b)
84.  print(gc)
85.
86.  # ----- Surface plot -----
87.  X, T = np.meshgrid(tspan, xspan)
88.
89.  fig = plt.figure()
90.  ax = fig.gca(projection='3d')
91.
92.  surf = ax.plot_surface(X, T, U, linewidth=0,
93.                         cmap=cm.coolwarm, antialiased=False)
94.
95.  ax.set_yticks([0, 0.25, 0.5, 0.75, 1.0])
96.  ax.set_xticks([0, 0.05, 0.1, 0.15, 0.2])
97.
98.  ax.set_xlabel('Time')
99.  ax.set_ylabel('Space')
100. ax.set_zlabel('U')
101. plt.tight_layout()
102. plt.show()
```

CN - Neumann Problem

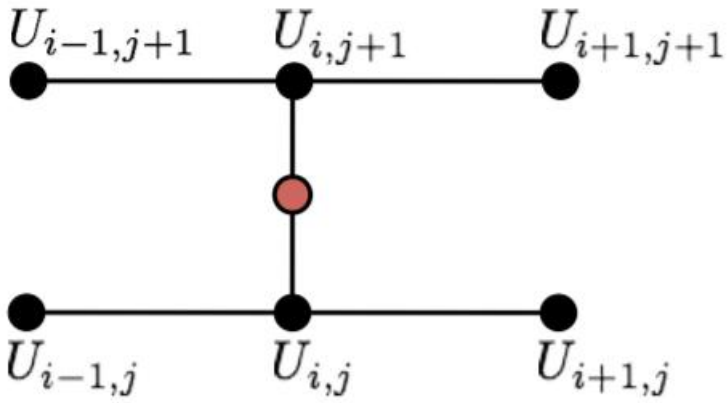
An implicit scheme, invented by John Crank and Phyllis Nicolson, is based on numerical approximations for solutions of differential equation at the point $\left(x_j, t_{j+\frac{\Delta t}{2}}\right)$ that lies between the rows in the grid.
The approximation formula for time derivative is given by

$$u\left(x_j, t_{j+\frac{\Delta t}{2}}\right) = \frac{u\left(x_j, t_{j+1}\right) - u\left(x_j, t_j\right)}{\Delta t} + o\left(\Delta t^2\right)$$

And for spatial derivative

$$u_{xx}\left(x_i, t_{j+\frac{\Delta t}{2}}\right) = \frac{1}{2\Delta x^2}\left(u\left(x_{i-1}, t_{j+1}\right) - 2u\left(x_i, t_{j+1}\right) + u\left(x_{i+1}, t_{j+1}\right) + u\left(x_{i-1}, t_j\right) - 2u\left(x_i, t_j\right) + u\left(x_{i+1}, t_j\right)\right) + \mathrm{O}\left(\Delta x\right.$$

The Crank-Nicolson method in numerical stencil is illustrated as in the figure.



In a similar fashion to the previous derivation, the difference equation for Crank-Nicolson method is

$$\frac{U_{i,j+1} - U_{i,j}}{\Delta t} = \frac{D}{2\Delta x^2}\left(U_{i-1,j+1} - 2U_{i,j+1} + U_{i+1,j+1} + U_{i-1,j} - 2U_{i,j} + U_{i+1,j}\right)$$

And rearranging gives

$$-rU_{i-1,j+1} + (1+2r)U_{i,j+1} - rU_{i+1,j+1} = rU_{i-1,j} + (1-2r)U_{i,j} + rU_{i+1,j}$$

For diffusion equation with Dirichlet boundary conditions,the equation can be represented in matrix-vector notation

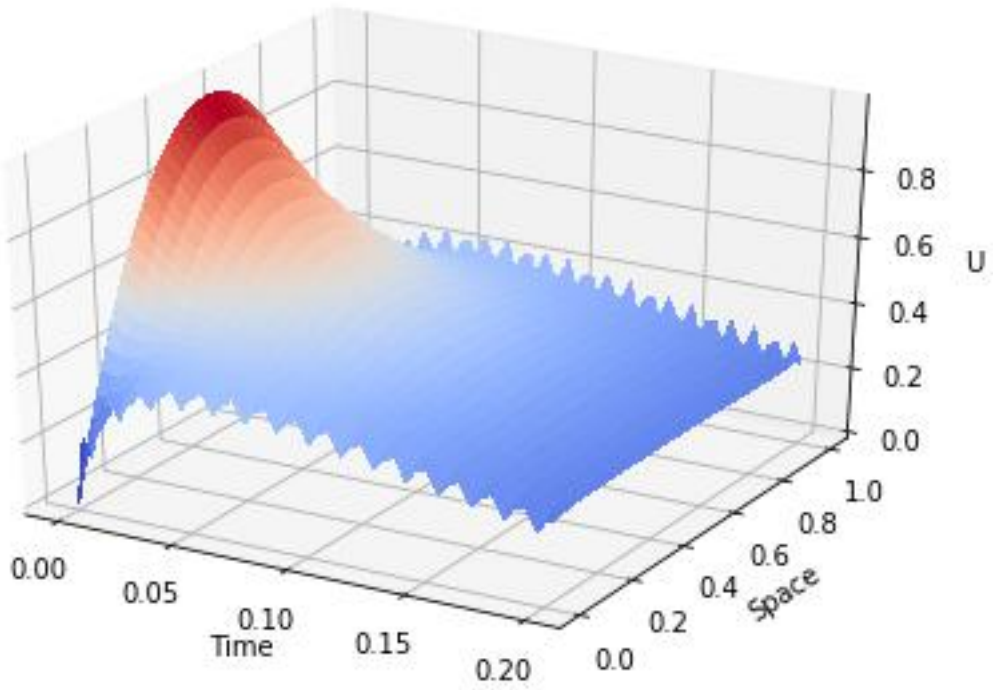
$$\begin{bmatrix} 1+2r & -r & & & \\ -r & 1+2r & -r & & \\ & \cdots & \cdots & \cdots & \\ & & -r & 1+2r & -r \\ & & & -r & 1+2r \end{bmatrix} \begin{bmatrix} U_{1,j+1} \\ U_{2,j+1} \\ \cdots \\ U_{M-3,j+1} \\ U_{M-2,j+1} \end{bmatrix} = \begin{bmatrix} 1-2r & r & & & \\ r & 1-2r & r & & \\ & \cdots & \cdots & \cdots & \\ & & r & 1-2r & r \\ & & & r & 1-2r \end{bmatrix} \begin{bmatrix} U_{1,j} \\ U_{2,j} \\ \cdots \\ U_{M-3,j} \\ U_{M-2,j} \end{bmatrix}$$

$$+ \begin{bmatrix} rU_{0,j} \\ 0 \\ \cdots \\ 0 \\ rU_{M-1,\mathrm{j}} + rU_{M-1,j+1} \end{bmatrix}$$

Crank-Nicolson method to solve 1D reaction-diffusion equation:

u_t = D * u_xx + alpha * u

with Neumann boundary conditions
at x=0: u_x = sin(pi/2)
at x=L: u_x = sin(3*pi/4) with L=1
and initial condition u(x,0) = 4*x - 4*x**2
Result:



Code:

```
@author: Wentao Gong
"""

'''''
Crank-Nicolson method to solve 1D reaction-diffusion equation:
    u_t = D * u_xx + alpha * u

with Neumann boundary conditions
at x=0: u_x = sin(pi/2)
at x=L: u_x = sin(3*pi/4) with L=1
and initial condition u(x,0) = 4*x - 4*x**2
'''

import numpy as np
from scipy import sparse
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
```

```
19.
20.
21. M = 50 # GRID POINTS on space interval
22. N = 60 # GRID POINTS on time interval
23.
24. x0 = 0
25. xL = 1
26.
27. # ----- Spatial discretization step -----
28. dx = (xL - x0)/(M - 1)
29.
30. t0 = 0
31. tF = 0.2
32.
33. # ----- Time step -----
34. dt = (tF - t0)/(N - 1)
35.
36. D = 0.5  # Diffusion coefficient
37. alpha = -5.0 # Reaction rate
38.
39. r = dt*D/(2*dx**2)
40. s = dt*alpha/2
41. a0 = 1 + 2*r - s
42. c0 = 1 - 2*r + s
43.
44.
45. xspan = np.linspace(x0, xL, M)
46. tspan = np.linspace(t0, tF, N)
47.
48. main_diag_a0 = a0*np.ones((1,M))
49. off_diag_a0 = -r*np.ones((1, M-1))
50.
51. main_diag_c0 = c0*np.ones((1,M))
52. off_diag_c0 = r*np.ones((1, M-1))
53.
54. # Left-hand side tri-diagonal matrix
55. a = main_diag_a0.shape[1]
56. diagonalsA = [main_diag_a0, off_diag_a0, off_diag_a0]
57. A = sparse.diags(diagonalsA, [0,-1,1], shape=(a,a)).toarray()
58. A[0,1] = -2*r
59. A[M-1,M-2] = -2*r
60.
61. # Right-hand side tri-diagonal matrix
62. c = main_diag_c0.shape[1]
63. diagonalsC = [main_diag_c0, off_diag_c0, off_diag_c0]
```

```
A_rhs = sparse.diags(diagonalsC, [0,-1,1], shape=(c,c)).toarray()
A_rhs[0,1] = 2*r
A_rhs[M-1,M-2] = 2*r

# ----- Initializes matrix U -----
U = np.zeros((M, N))

#----- Initial condition -----
U[:,0] = 4*xspan - 4*xspan**2

#----- Neumann boundary conditions -----
leftBC = np.arange(1, N+1)
f = np.sin(leftBC*np.pi/2)

rightBC = np.arange(1, N+1)
g = np.sin(3*rightBC*np.pi/4)
#U[0,:] = np.sin(leftBC*np.pi/2)
#U[-1,:] = 0.0

for k in range(1, N):
    ins = np.zeros((M-2,1)).ravel()
    b1 = np.asarray([4*r*dx*f[k], 4*r*dx*g[k]])
    b1 = np.insert(b1, 1, ins)
    b2 = np.matmul(A_rhs, np.array(U[0:M, k-1]))
    b = b1 + b2  # Right hand side
    U[0:M, k] = np.linalg.solve(A,b)  # Solve x=A\b

# ----- Checks if the solution is correct:
gc = np.allclose(np.dot(A,U[0:M,N-1]), b)
print(gc)

# ----- Surface plot -----
X, T = np.meshgrid(tspan, xspan)

fig = plt.figure()
ax = fig.gca(projection='3d')

surf = ax.plot_surface(X, T, U, linewidth=0,
                       cmap=cm.coolwarm, antialiased=False)

ax.set_xticks([0, 0.05, 0.1, 0.15, 0.2])

ax.set_xlabel('Time')
ax.set_ylabel('Space')
ax.set_zlabel('U')
```

```
plt.tight_layout()
plt.show()
```

Advection Equation

$$\frac{\partial u}{\partial t}+a\frac{\partial u}{\partial x}=0$$

where a is a constant. For the Cauchy* problem, the equation is completed with initial condition

$$u(x,0)=\eta(x)$$

This is the simplest example of a hyperbolic equation, where its exact solution is:

$$u(x,t)=\eta(x-at)$$

Cauchy problem: $-\infty<x<\infty$ on (without boundary)
If the spatial dependence is approximated using the central difference and the temporal dependence is approximated using the forward difference, we get

$$\frac{U_{i,j+1}-U_{i,j}}{\Delta t}=-\frac{a}{2\Delta x}\left(U_{i+1,j}-U_{i-1,j}\right)$$

Which can be rewritten as

$$U_{i,j+1}=U_{i,j}-\frac{a\Delta t}{2\Delta x}\left(U_{i+1,j}-U_{i-1,j}\right)$$

Replace $U_{i,j}$ on the right-hand side by the average

$$1/2\left(U_{i+1,j}+U_{i-1,j}\right)$$

Then we obtain the Lax-Friedrichs Method

$$U_{i,j+1}=1/2\left(U_{i+1,j}+U_{i-1,j}\right)-\frac{a\Delta t}{2\Delta x}\left(U_{i+1,j}-U_{i-1,j}\right)$$

This method is not commonly used in practice because of its low accuracy. This method is convergent only if the following requirement is fulfilled:

$$\left|\frac{a\Delta t}{\Delta x}\right|\le 1$$

This stability restriction allows us to use time step $\Delta t=\mathrm{O}(\Delta x)$

Although the method is explicit

The basic reason is that advection equation involves only the first order derivative of u_x rather

than u_{xx} , so the difference equation involves 1/Δx rather than 1/Δx^2. Unlike the heat/diffusion equation, the advection equation is not stiff. This is a fundamental difference between hyperbolic equations (such as the advection equation) and parabolic equations (such as the diffusion equation). The hyperbolic equations are typically solved with explicit methods, while the efficient solution of parabolic equations generally requires implicit methods

Lax-Friedrichs Method

Using the fact that

$$\frac{1}{2}\left(U_{i+1,j}+U_{i-1,j}\right)=U_{i,j}+\frac{1}{2}\left(U_{i+1,j}-2U_{i,j}+U_{i-1,j}\right)$$

Then we obtain

$$U_{i,j+1}=U_{i,j}+\frac{1}{2}\left(U_{i+1,j}-2U_{i,j}+U_{i-1,j}\right)-\frac{a\Delta t}{2\Delta x}\left(U_{i+1,j}-U_{i-1,j}\right)$$

Assuming the $\frac{\Delta t}{\Delta x}$ is fixed.

$$\frac{U_{i,j+1}-U_{i,j}}{\Delta t}+a\left(\frac{U_{i+1,j}-U_{i-1,j}}{2\Delta x}\right)=\frac{\Delta x^2}{2\Delta t}\left(\frac{U_{i+1,j}-2U_{i,j}+U_{i-1,j}}{\Delta x^2}\right)$$

Which look

$$\frac{\partial u}{\partial t}+a\frac{\partial u}{\partial x}=D\frac{\partial^2 u}{\partial x^2}$$

Where $D=\frac{\Delta x^2}{2\Delta t}$

The above equations can be viewed as a forward Euler discretization of the system of ODEs

$$\frac{dU}{dt}=A_D U$$

Where

$$A_D=-\frac{a}{2\Delta x}\begin{bmatrix}0 & 1 & & & -1\\ -1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 0 & 1\\ 1 & & & -1 & 0\end{bmatrix}+\frac{D}{\Delta x^2}\begin{bmatrix}-2 & 1 & & & 1\\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1\\ 1 & & & 1 & -2\end{bmatrix}$$

The Method is stable provided that $\left|\frac{a\Delta t}{\Delta x}\right| \leq 1$

To obtain a system of equations with finite dimension, we must solve the equation on some bounded domain rather than solving the Cauchy problem. In a bounded domain, say, $0 \leq x \leq 1$ the advection equation can have a boundary condition specified on only one of the two boundaries.
If a>0, then we need a boundary condition at x = 0, say

$$u(0,t) = g_0(t)$$

which is the inflow boundary. The boundary at x = 1 is the outflow boundary and the solution at this boundary is completely determined by what is advecting to the right from the interior.
If a < 0, then we need a boundary condition at x = 1, which is the inflow boundary in this case.
If we consider the special case of periodic boundary conditions,

$$u(0,t) = u(1,t) \text{ for } t \geq 0$$

where these conditions say that whatever flows out at the outflow boundary flows back in at the inflow boundary.

To solves the advection equation

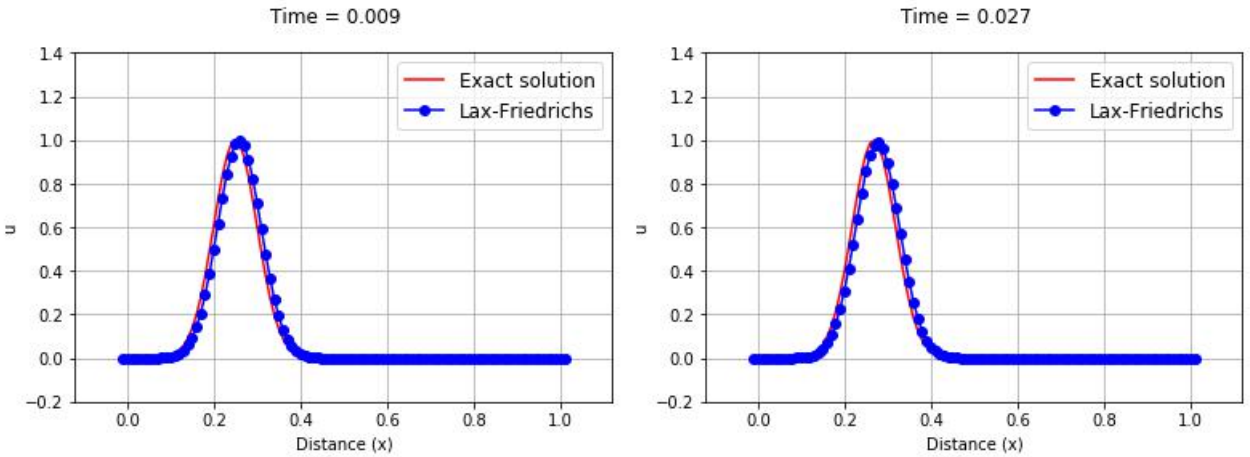
U_t + vU_x = 0

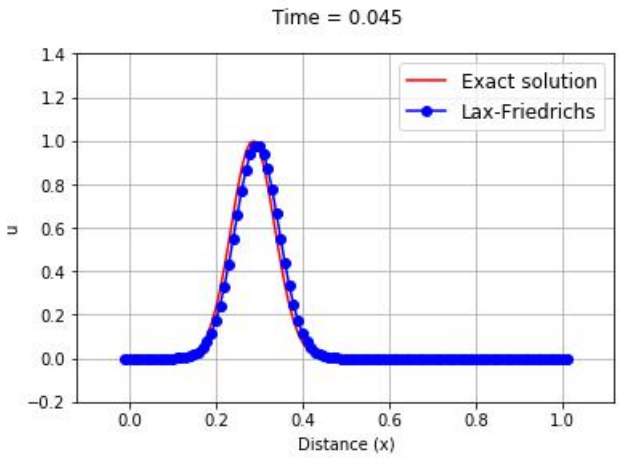
over the spatial domain of 0 <= x <= 1 that is discretized into 103 nodes, with dx=0.01, using the Lax-Friedrichs scheme for an initial profile of a Gaussian curve, defined by

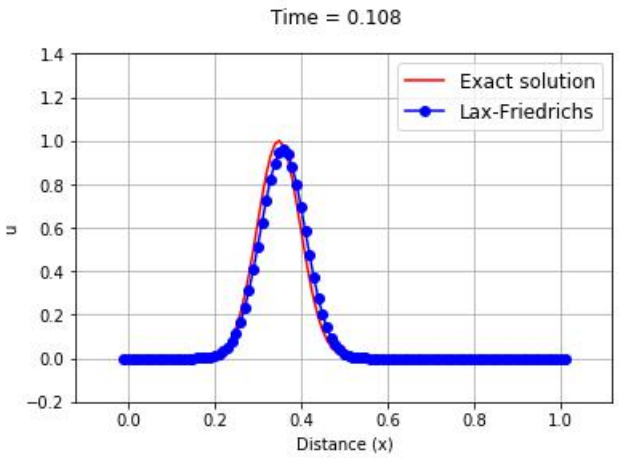
U(x,t) = exp(-200*(x-xc-v*t).^2)

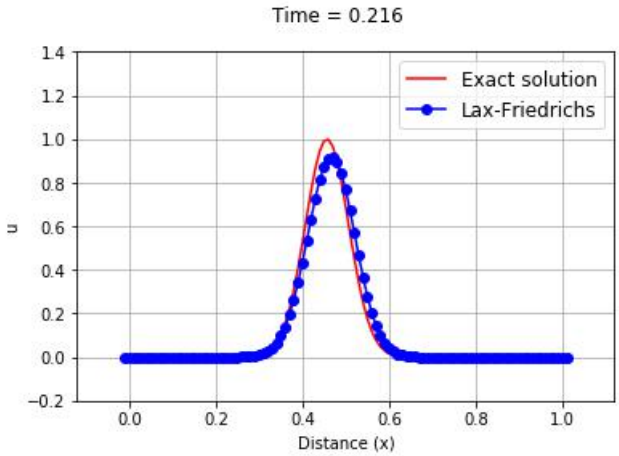
where xc=0.25 is the center of the curve at t=0.

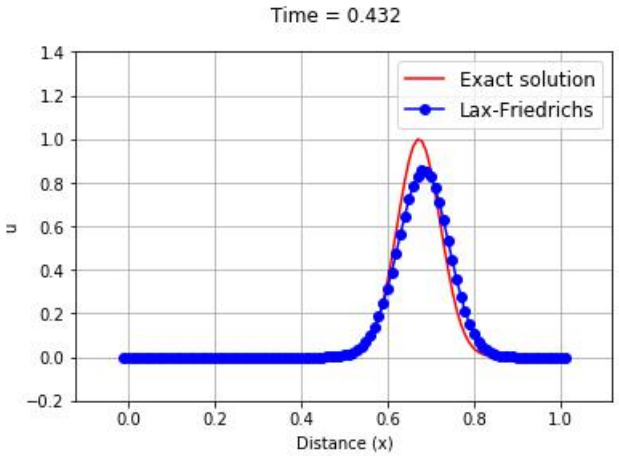
The periodic boundary conditions are applied either end of the domain.
The velocity is v=1. The solution is iterated until t=1.5 seconds.

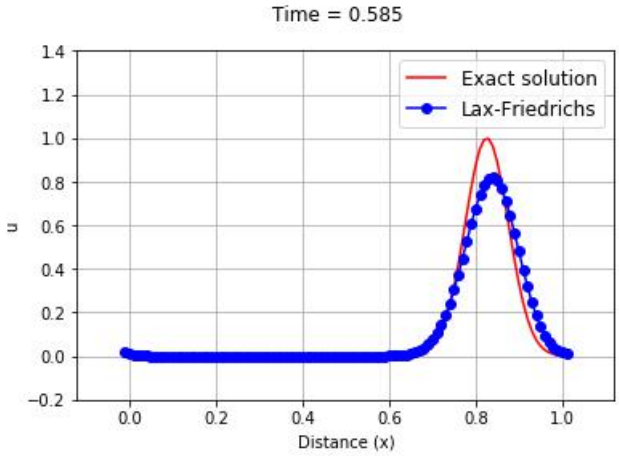


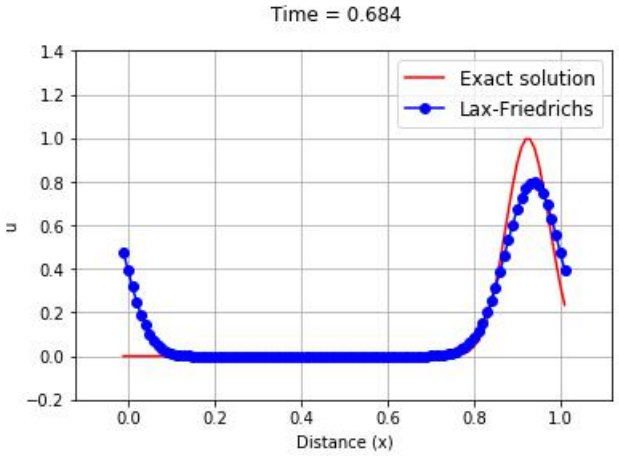

Time = 0.045
Exact solution
Lax-Friedrichs
u
Distance (x)

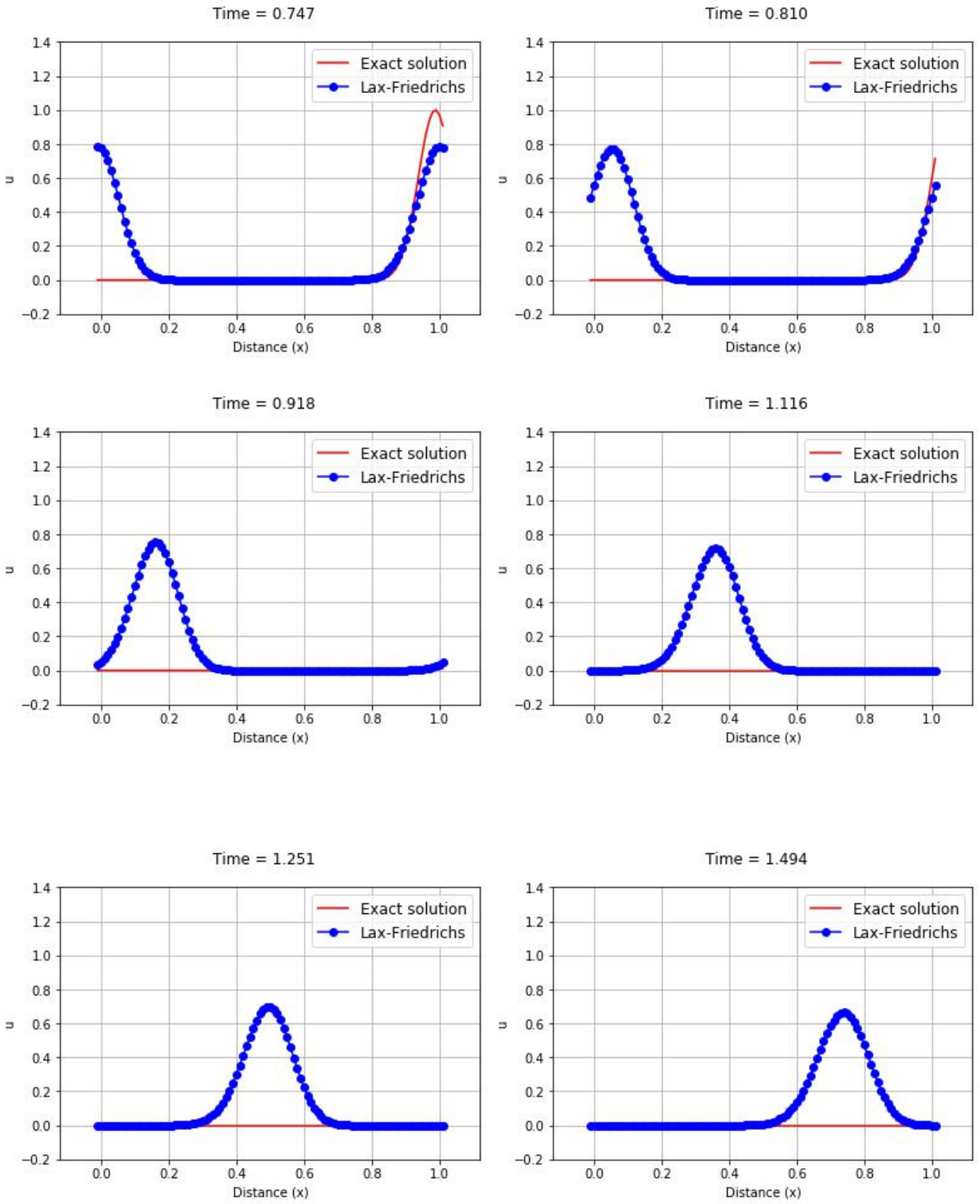

Time = 0.108
Exact solution
Lax-Friedrichs
u
Distance (x)


Time = 0.216
Exact solution
Lax-Friedrichs
u
Distance (x)


Time = 0.432
Exact solution
Lax-Friedrichs
u
Distance (x)


Time = 0.585
Exact solution
Lax-Friedrichs
u
Distance (x)


Time = 0.684
Exact solution
Lax-Friedrichs
u
Distance (x)



Code

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.
4.
5.  class LaxFriedrichs:
```

```
6.
7.      def __init__(self, N, tmax):
8.          self.N = N # number of nodes
9.          self.tmax = tmax
10.         self.xmin = 0
11.         self.xmax = 1
12.         self.dt = 0.009 # timestep
13.         self.v = 1 # velocity
14.         self.xc = 0.25
15.         self.initializeDomain()
16.         self.initializeU()
17.         self.initializeParams()
18.
19.
20.     def initializeDomain(self):
21.         self.dx = (self.xmax - self.xmin)/self.N
22.         self.x = np.arange(self.xmin-self.dx, self.xmax+(2*self.dx),
   self.dx)
23.
24.
25.     def initializeU(self):
26.         u0 = np.exp(-200*(self.x-self.xc)**2)
27.         self.u = u0.copy()
28.         self.unp1 = u0.copy()
29.
30.
31.     def initializeParams(self):
32.         self.nsteps = round(self.tmax/self.dt)
33.         self.alpha = self.v*self.dt/(2*self.dx)
34.
35.
36.     def solve_and_plot(self):
37.         tc = 0
38.
39.         for i in range(self.nsteps):
40.             plt.clf()
41.
42.             # The Lax-Friedrichs scheme, Eq. (18.17)
43.             for j in range(self.N+2):
44.                 self.unp1[j] = self.u[j] - self.alpha*(self.u[j+1] -
   self.u[j-1]) + \
45.                 (1/2)*(self.u[j+1] - 2*self.u[j] + self.u[j-1])
46.
47.             self.u = self.unp1.copy()
48.
```

```
49.             # Periodic boundary conditions
50.             self.u[0] = self.u[self.N+1]
51.             self.u[self.N+2] = self.u[1]
52.
53.             uexact = np.exp(-200*(self.x-self.xc-self.v*tc)**2)
54.
55.             plt.plot(self.x, uexact, 'r', label="Exact solution")
56.             plt.plot(self.x, self.u, 'bo-', label="Lax-Friedrichs")
57.             plt.axis((self.xmin-0.12, self.xmax+0.12, -0.2, 1.4))
58.             plt.grid(True)
59.             plt.xlabel("Distance (x)")
60.             plt.ylabel("u")
61.             plt.legend(loc=1, fontsize=12)
62.             plt.suptitle("Time = %1.3f" % (tc+self.dt))
63.             plt.pause(0.01)
64.             tc += self.dt
65.
66.
67. def main():
68.     sim = LaxFriedrichs(100, 1.5)
69.     sim.solve_and_plot()
70.     plt.show()
71.
72.
73. if __name__ == "__main__":
74.     main()
```

Lax-Wendroff Method

The Lax-Wendroff method is given by

$$U_{i,j+1} = U_{i,j} - \frac{a\Delta t}{2\Delta x}\left(U_{i+1,j} - U_{i-1,j}\right) + \frac{a^2\Delta t^2}{2\Delta x^2}\left(U_{i+1,j} - 2U_{i,j} + U_{i-1,j}\right)$$

This code solves the advection equation

U_t + vU_x = 0

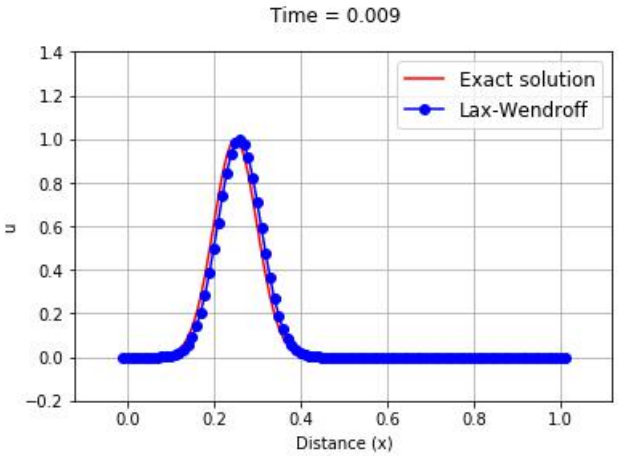
over the spatial domain of 0 <= x <= 1 that is discretized into 103 nodes, with dx=0.01, using the Lax-Wendroff scheme for an initial profile of a Gaussian curve, defined by

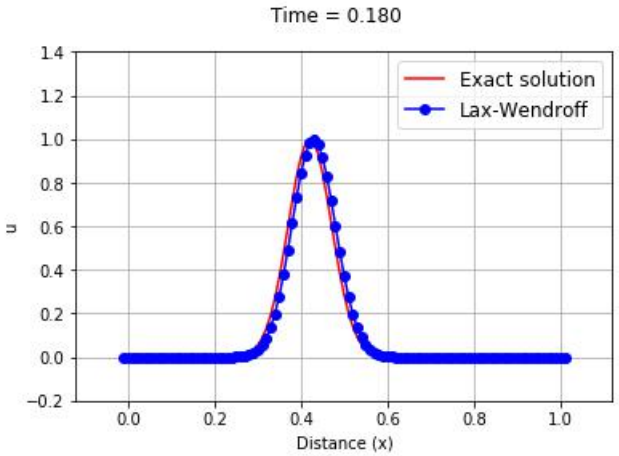
U(x,t) = exp(-200*(x-xc-v*t).^2)

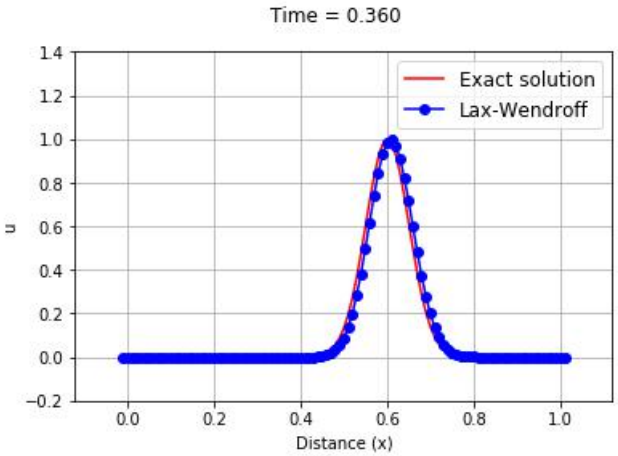
where xc=0.25 is the center of the curve at t=0.

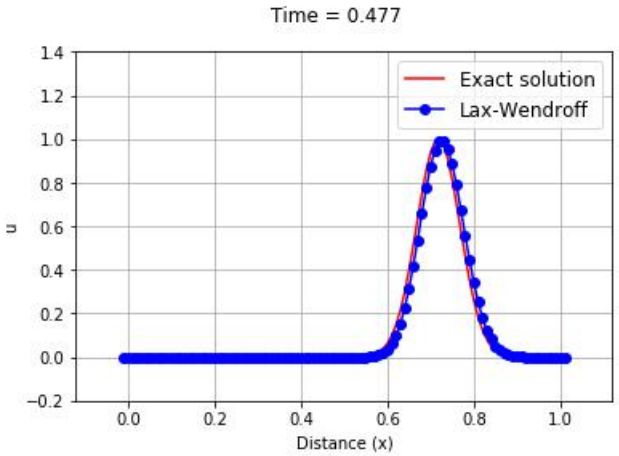
The periodic boundary conditions are applied either end of the domain.

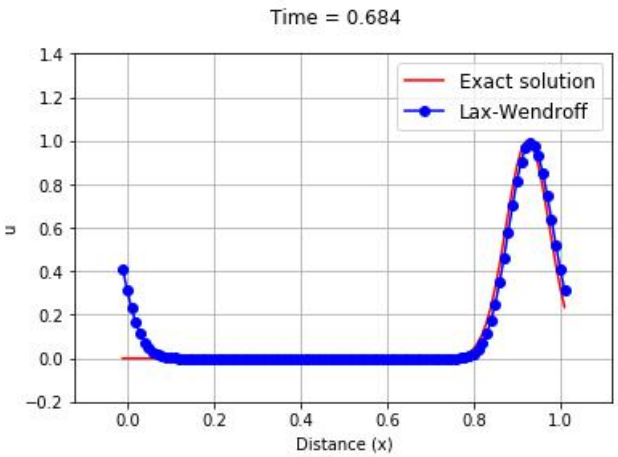
The velocity is v=1. The solution is iterated until t=1.5 seconds.

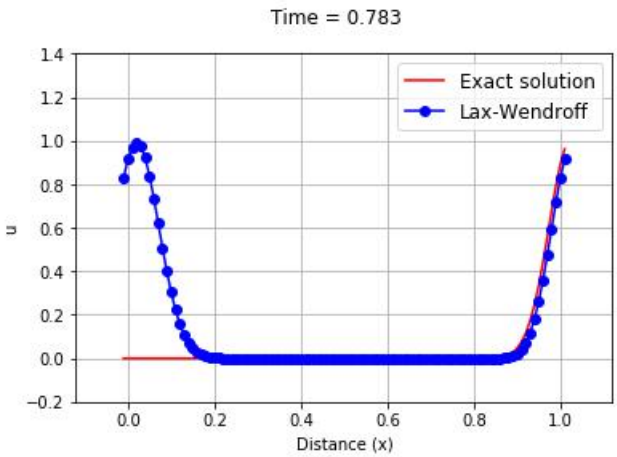


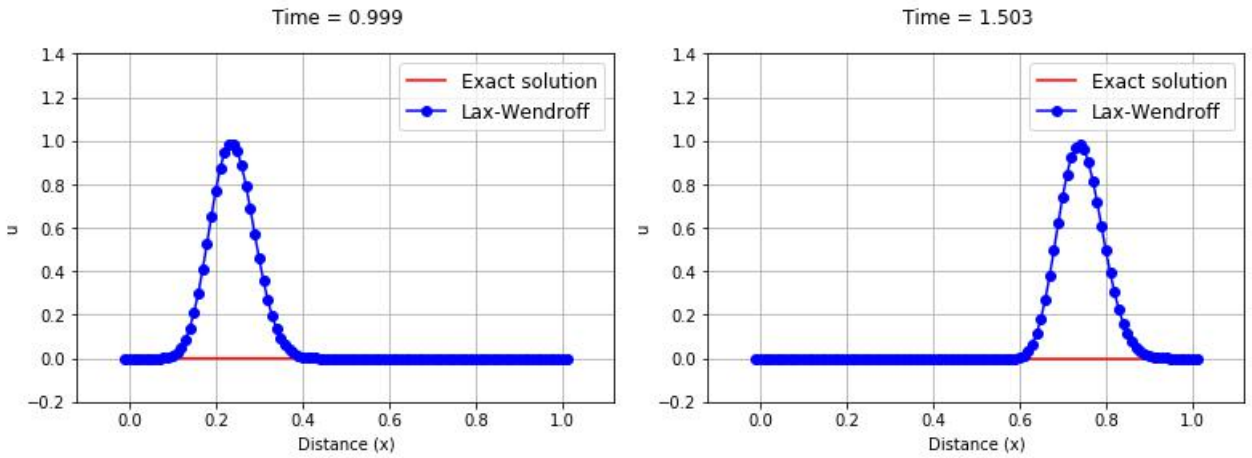












Code

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.
4.
5.  class LaxWendroff:
6.
7.      def __init__(self, N, tmax):
8.          self.N = N # number of nodes
9.          self.tmax = tmax
10.         self.xmin = 0
11.         self.xmax = 1
12.         self.dt = 0.009 # timestep
13.         self.v = 1 # velocity
14.         self.xc = 0.25
15.         self.initializeDomain()
16.         self.initializeU()
17.         self.initializeParams()
18.
19.
20.     def initializeDomain(self):
21.         self.dx = (self.xmax - self.xmin)/self.N
22.         self.x = np.arange(self.xmin-self.dx, self.xmax+(2*self.dx),
    self.dx)
23.
24.
25.     def initializeU(self):
26.         u0 = np.exp(-200*(self.x-self.xc)**2)
27.         self.u = u0.copy()
28.         self.unp1 = u0.copy()
29.
30.
31.     def initializeParams(self):
```

```
        self.nsteps = round(self.tmax/self.dt)
        self.alpha = self.v*self.dt/(2*self.dx)


    def solve_and_plot(self):
        tc = 0

        for i in range(self.nsteps):
            plt.clf()

            # The Lax-Wendroff scheme, Eq. (18.20)
            for j in range(self.N+2):
                self.unp1[j]          =          self.u[j]          + (self.v**2*self.dt**2/(2*self.dx**2))*(self.u[j+1]-2*self.u[j]+self.u[j-1]) \
                - self.alpha*(self.u[j+1]-self.u[j-1])

            self.u = self.unp1.copy()

            # Periodic boundary conditions
            self.u[0] = self.u[self.N+1]
            self.u[self.N+2] = self.u[1]

            uexact = np.exp(-200*(self.x-self.xc-self.v*tc)**2)

            plt.plot(self.x, uexact, 'r', label="Exact solution")
            plt.plot(self.x, self.u, 'bo-', label="Lax-Wendroff")
            plt.axis((self.xmin-0.12, self.xmax+0.12, -0.2, 1.4))
            plt.grid(True)
            plt.xlabel("Distance (x)")
            plt.ylabel("u")
            plt.legend(loc=1, fontsize=12)
            plt.suptitle("Time = %1.3f" % (tc+self.dt))
            plt.pause(0.01)
            tc += self.dt


def main():
    sim = LaxWendroff(100, 1.5)
    sim.solve_and_plot()
    plt.show()


if __name__ == "__main__":
    main()
```

Upwind Method

The upwind methods with forward differencing in time and backward and forward in space are given by

$$U_{i,j+1} = U_{i,j} - \frac{a\Delta t}{\Delta x}\left(U_{i,j} - U_{i-1,j}\right)$$

The method is stable only if

$$0 \le \frac{a\Delta t}{\Delta x} \le 1$$

which can only be used if a > 0
Or

$$U_{i,j+1} = U_{i,j} - \frac{a\Delta t}{\Delta x}\left(U_{i+1,j} - U_{i,j}\right)$$

The method is stable only if

$$-1 \le \frac{a\Delta t}{\Delta x} \le 0$$

which can be used only if a < 0
These methods are first order accurate in both space and time. If a > 0 the solution moves to the right, while if a < 0 the solution moves to the left.

To solve the advection equation

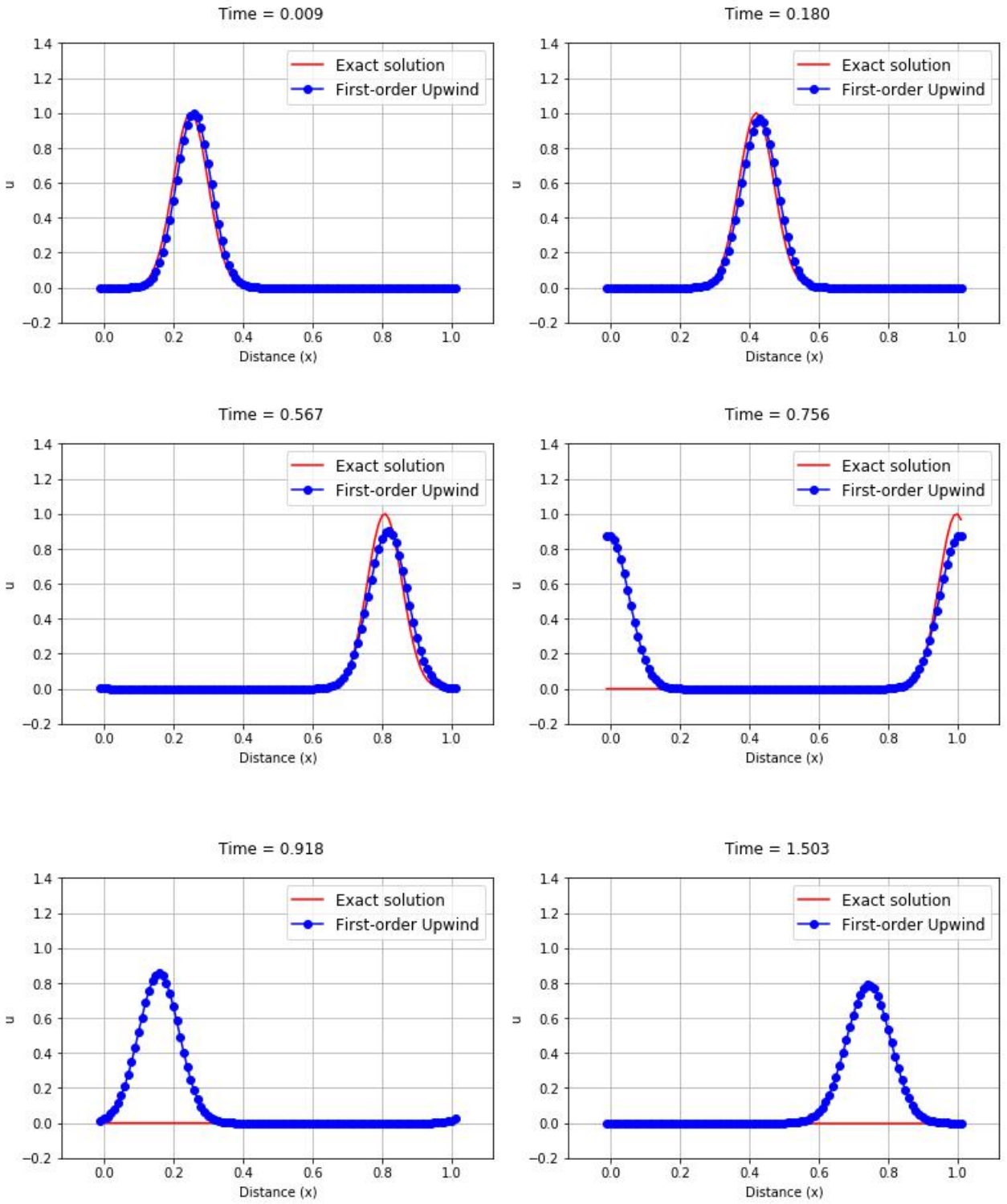
U_t + vU_x = 0

over the spatial domain of 0 <= x <= 1 that is discretized into 103 nodes, with dx=0.01, using the First-Order Upwind (FOU) scheme with Forward difference in Eq. (18.21) for an initial profile of a Gaussian curve, defined by

U(x,t) = exp(-200*(x-xc-v*t).^2)

where xc=0.25 is the center of the curve at t=0.

The periodic boundary conditions are applied either end of the domain. The velocity is v=1. The solution is iterated until t=1.5 seconds.
Results




```
1.  import numpy as np
1.  import matplotlib.pyplot as plt
2.
3.
4.  class UpwindMethod1:
5.
6.      def __init__(self, N, tmax):
```

```
7.          self.N = N # number of nodes
8.          self.tmax = tmax
9.          self.xmin = 0
10.         self.xmax = 1
11.         self.dt = 0.009 # timestep
12.         self.v = 1 # velocity
13.         self.xc = 0.25
14.         self.initializeDomain()
15.         self.initializeU()
16.         self.initializeParams()
17.
18.
19.     def initializeDomain(self):
20.         self.dx = (self.xmax - self.xmin)/self.N
21.         self.x = np.arange(self.xmin-self.dx, self.xmax+(2*self.dx),
   self.dx)
22.
23.
24.     def initializeU(self):
25.         u0 = np.exp(-200*(self.x-self.xc)**2)
26.         self.u = u0.copy()
27.         self.unp1 = u0.copy()
28.
29.
30.     def initializeParams(self):
31.         self.nsteps = round(self.tmax/self.dt)
32.
33.
34.     def solve_and_plot(self):
35.         tc = 0
36.
37.         for i in range(self.nsteps):
38.             plt.clf()
39.
40.             # The FOU scheme, Eq. (18.21)
41.             for j in range(self.N+2):
42.                 self.unp1[j]          =          self.u[j]          -
   (self.v*self.dt/(self.dx))*(self.u[j]-self.u[j-1])
43.
44.             self.u = self.unp1.copy()
45.
46.             # Periodic boundary conditions
47.             self.u[0] = self.u[self.N+1]
48.             self.u[self.N+2] = self.u[1]
49.
```

```
            uexact = np.exp(-200*(self.x-self.xc-self.v*tc)**2)

            plt.plot(self.x, uexact, 'r', label="Exact solution")
            plt.plot(self.x,   self.u,   'bo-',   label="First-order
   Upwind")
            plt.axis((self.xmin-0.12, self.xmax+0.12, -0.2, 1.4))
            plt.grid(True)
            plt.xlabel("Distance (x)")
            plt.ylabel("u")
            plt.legend(loc=1, fontsize=12)
            plt.suptitle("Time = %1.3f" % (tc+self.dt))
            plt.pause(0.01)
            tc += self.dt


def main():
    sim = UpwindMethod1(100, 1.5)
    sim.solve_and_plot()
    plt.show()


if __name__ == "__main__":
    main()
```

Beam-Warming Method

The Beam-Warming method is second order accurate. For a > 0, the method takes the form

$$U_{i,j+1} = U_{i,j} - \frac{a\Delta t}{2\Delta x}\left(3U_{i,j} - 4U_{i-1,j} + U_{i-2,j}\right) + \frac{a^2\Delta t^2}{2\Delta x^2}\left(U_{i,j} - 2U_{i-1,j} + U_{i-2,j}\right)$$

For a < 0, the method takes the form

$$U_{i,j+1} = U_{i,j} - \frac{a\Delta t}{2\Delta x}\left(-3U_{i,j} + 4U_{i-1,j} - U_{i-2,j}\right) + \frac{a^2\Delta t^2}{2\Delta x^2}\left(U_{i,j} - 2U_{i-1,j} + U_{i-2,j}\right)$$

Solve the advection equation

U_t + vU_x = 0

over the spatial domain of 0 <= x <= 1 that is discretized into 103 nodes, with dx=0.01, using the Beam-Warming scheme given by Eq. (18.25) with a<0, for an initial profile of a Gaussian curve, defined by

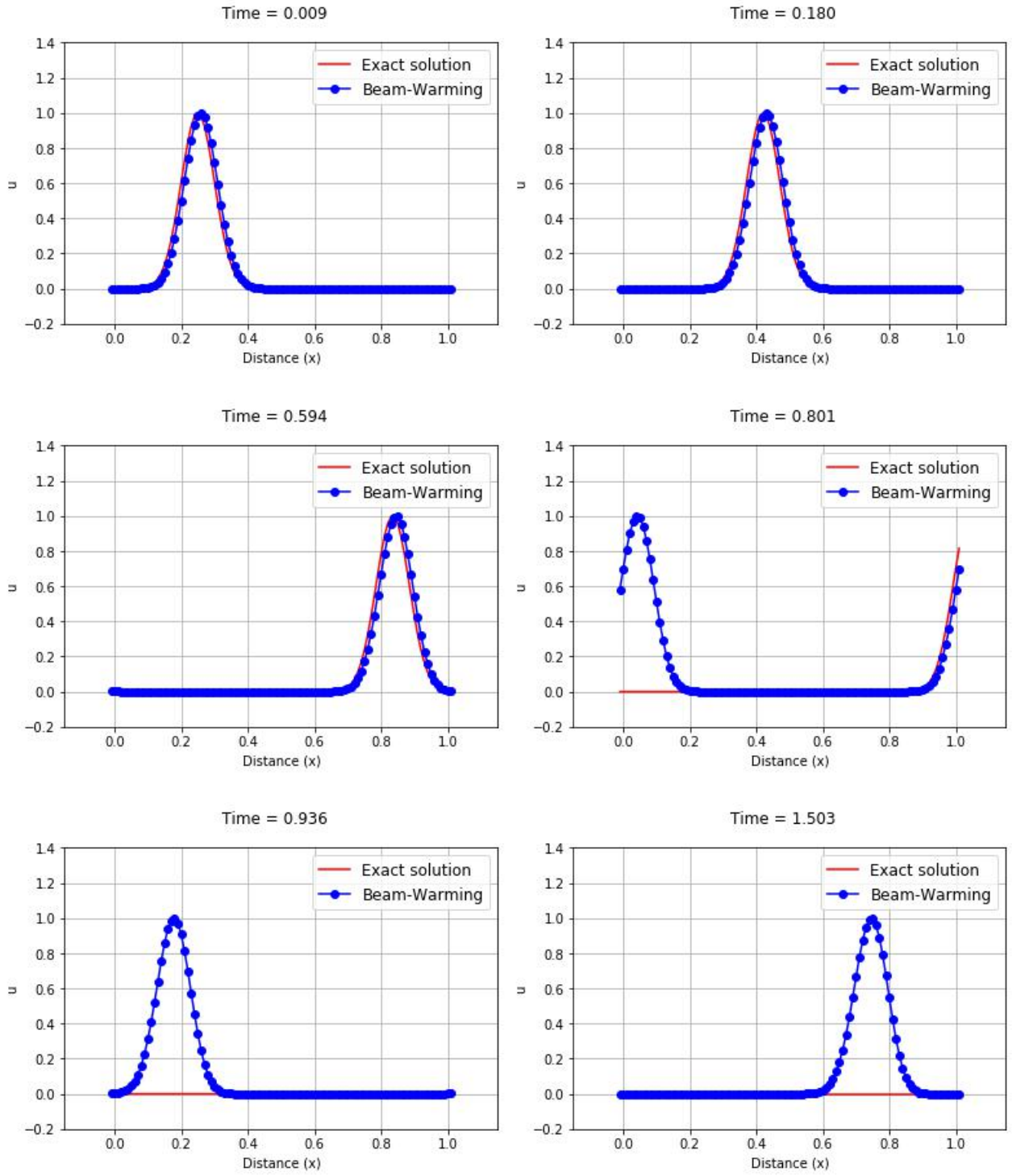
U(x,t) = exp(-200*(x-xc-v*t).^2)

where xc=0.25 is the center of the curve at t=0.

Periodic boundary conditions are applied at both ends of

the domain.

The velocity is v=1. The solution is iterated until t=1.5 seconds.



Code

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.
4.
5.  class beamWarming1:
```

```
6.
7.      def __init__(self, N, tmax):
8.          self.N = N # number of nodes
9.          self.tmax = tmax
10.         self.xmin = 0
11.         self.xmax = 1
12.         self.dt = 0.009 # timestep
13.         self.v = 1 # velocity
14.         self.xc = 0.25
15.         self.initializeDomain()
16.         self.initializeU()
17.         self.initializeParams()
18.
19.
20.     def initializeDomain(self):
21.         self.dx = (self.xmax - self.xmin)/self.N
22.         self.x = np.arange(self.xmin-self.dx, self.xmax+(2*self.dx),
   self.dx)
23.
24.
25.     def initializeU(self):
26.         u0 = np.exp(-200*(self.x-self.xc)**2)
27.         self.u = u0.copy()
28.         self.unp1 = u0.copy()
29.
30.
31.     def initializeParams(self):
32.         self.nsteps = round(self.tmax/self.dt)
33.         self.alpha1 = self.v*self.dt/(2*self.dx)
34.         self.alpha2 = self.v**2*self.dt**2/(2*self.dx**2)
35.
36.
37.     def solve_and_plot(self):
38.         tc = 0
39.
40.         for i in range(self.nsteps):
41.             plt.clf()
42.
43.             # The Beam-Warming scheme, Eq. (18.25)
44.             for j in range(self.N+3):
45.                 self.unp1[j] = self.u[j] - self.alpha1*(3*self.u[j] -
   4*self.u[j-1] + self.u[j-2]) + \
46.                 self.alpha2*(self.u[j] - 2*self.u[j-1] + self.u[j-2])
47.
48.             self.u = self.unp1.copy()
```

```
49.
50.             # Periodic boundary conditions
51.             self.u[0] = self.u[self.N+1]
52.             self.u[1] = self.u[self.N+2]
53.
54.             uexact = np.exp(-200*(self.x-self.xc-self.v*tc)**2)
55.
56.             plt.plot(self.x, uexact, 'r', label="Exact solution")
57.             plt.plot(self.x, self.u, 'bo-', label="Beam-Warming")
58.             plt.axis((self.xmin-0.15, self.xmax+0.15, -0.2, 1.4))
59.             plt.grid(True)
60.             plt.xlabel("Distance (x)")
61.             plt.ylabel("u")
62.             plt.legend(loc=1, fontsize=12)
63.             plt.suptitle("Time = %1.3f" % (tc+self.dt))
64.             plt.pause(0.01)
65.             tc += self.dt
66.
67.
68. def main():
69.     sim = beamWarming1(100, 1.5)
70.     sim.solve_and_plot()
71.     plt.show()
72.
73. if __name__ == "__main__":
74.     main()
```

CIP Method using Fortran

To solve the advection term, the high-order Godunov scheme known as the Constrained Interpolation Profile Scheme (CIP) proposed by Takewakiet al.(1985) is used.
Yabe et.al (2001) explained the strategy of CIP method by using an advection equation.

$$\frac{\partial f}{\partial t} + u\frac{\partial f}{\partial x} = 0 \qquad (3.30)$$

As we know, using the first order upwind scheme the numerical diffusion arises when we construct the profile by the linear interpolation. If we use a quadratic polynomial for the interpolation, the model suffers from overshooting, which is called the Lax-Wendroff scheme or the Leith scheme.
The accuracy declines, because we neglect the behavior of the solution inside of the grid cell and merely follow the smoothness of the solution. So the CIP concept is to develop a method incorporating the real solution into the profile within a grid cell.
The profile is as shown below. If we differentiate the above equation with spatial variable x,we get

$$\frac{\partial g}{\partial t} + u\frac{\partial g}{\partial x} = -\frac{\partial u}{\partial x}g \qquad (3.31)$$

where the g represents the spatial derivative of $\partial f / \partial x$. if two values of f and g are given at two grids points, the profile between these points can be interpolated by the cubic polynomial $F(x) = ax^3 + bx^2 + cx + d$. thus , the profile at the $n+1$ step can be obtained by shifting the profile by $u\Delta t$ so that

$$f^{n+1} = F(x - u\Delta t) \tag{3.32}$$

$$g^{n+1} = dF(x - u\Delta t)/dx \tag{3.33}$$

$$a_i = \frac{g_i + g_{iup}}{D^2} + \frac{2(f_i - f_{iup})}{D^3} \tag{3.34}$$

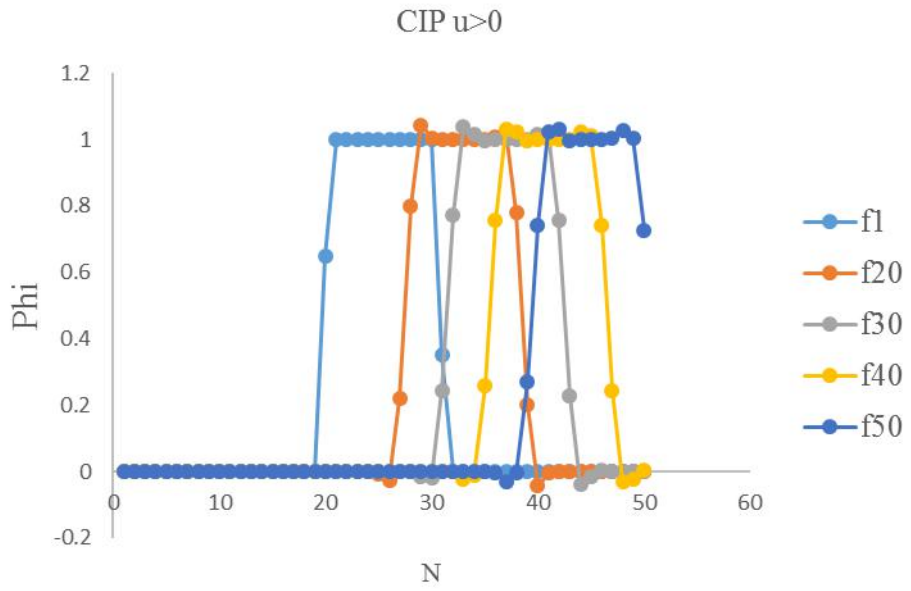
$$b_i = \frac{3(f_{iup} - f_i)}{D^2} - \frac{2g_i + g_{iup}}{D} \tag{3.35}$$

$$f_i^{n+1} = a_i\xi^3 + b_i\xi^2 + g_i^n\xi + f_i^n \tag{3.36}$$

$$g_i^{n+1} = 3a_i\xi^2 + 2b_i\xi + g_i^n \tag{3.37}$$

where we define ξ = -uΔt . $D = -\Delta x, iup = i - 1$ for $u \geq 0$ and $D = \Delta x$, $iup = i + 1$ for $u < 0$.

Initial Condition 20<x<30, phi=1
dx = 1.0m ,dt = 0.2s,u = 2m/s, CFL= 0.4.




```
1.  ! A fortran95 program for G95
2.  ! By GWT
3.  program main
4.    implicit none
```

```
5.    integer, parameter::imax = 100
6.    integer, parameter::nmax =50
7.    integer i,j,n
8.    real
   ai,bi,ci,di,fi(0:nmax+1),fiup(0:nmax+1),gi(0:nmax+1),giup(0:nmax+1),
   fnew(0:nmax+1),gnew(0:nmax+1)
9.    real dx,dt,xmax,x0,kai,D,temp(0:4),u,gzi,a,b
10.   dx = 1.0
11.   dt = 0.2
12.   u = 2
13.   kai = u*dt/dx
14.   D = -dx
15.   do n = 0,nmax,1
16.     fi(n) =0
17.     gi(n) = 0
18.     if (n.ge.20 .and. n.le.30) then
19.         fi(n) = 1.0
20.     end if
21.   end do
22.   open(1,file='test.txt',status="unknown")
23.   do i =1 ,imax,1
24.     do n=1, nmax,1
25.          !call cip1d(fi(n),fi(n-1),gi,gi(n-1),-dx*kai,D,temp)
26.     ai = (gi(n)+gi(n-1))/(D*D)+2.0*(fi(n)-fi(n-1))/(D*D*D)
27.     bi = 3 *(fi(n-1)-fi(n))/(D*D)- (2*gi(n)+gi(n-1))/D
28.     fnew(n) = ((ai*(-dx*kai)+bi)*(-dx*kai)+gi(n))*(-dx*kai)+fi(n)
29.     gnew(n)= (3.0*ai*(-dx*kai)+2*bi)*(-dx*kai)+gi(n)
30.
31.     end do
32.
33.     do n=1,nmax,1
34.         fi(n) = fnew(n)
35.         gi(n) = gnew(n)
36.         if(mod(nmax,2)==0) write (1,*) 'n,i,fnew(n),', n,i,fnew(n)
37.     end do
38.   end do
39.
40.
41.
42. end
```

Diffusion Equation

The alternating-direction implicit, or ADI, scheme provides a means for solving parabolic equations in 2-spatial dimensions using tri-diagonal matrices.
For the first step, the diffusion equation

$$u_t = D\nabla^2 u$$

is approximated by

$$\frac{U_{i,j}^{k+1/2} - U_{i,j}^{k}}{\Delta t/2} = D\left(\frac{U_{i+1,j}^{k} - 2U_{i,j}^{k} + U_{i-1,j}^{k}}{\Delta x^2} + \frac{U_{i,j+1}^{k+1/2} - 2U_{i,j}^{k+1/2} + U_{i,j-1}^{k+1/2}}{\Delta y^2}\right)$$

For the case $\Delta x = \Delta y = h$ and letting $r = \frac{\Delta t D}{2h^2}$

The eq can be expressed as

$$-rU_{i,j-1}^{k+1/2} + (1+2r)U_{i,j}^{k+1/2} - rU_{i,j+1}^{k+1/2} = rU_{i-1,j}^{k} + (1-2r)U_{i,j}^{k} + U_{i+1,j}^{k}$$

which is explicit.

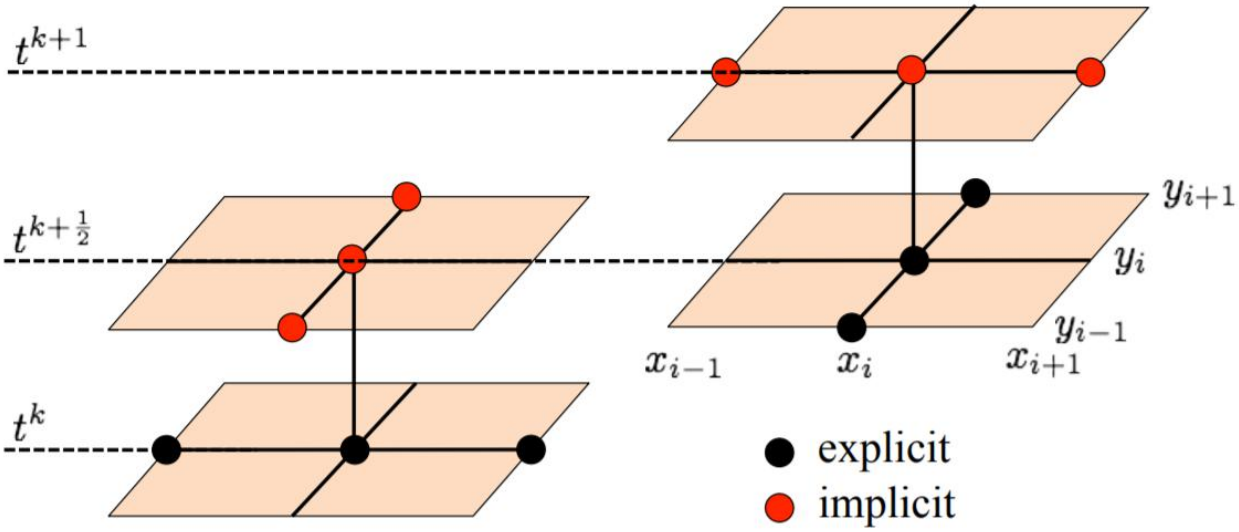
For the second step from $t^{k+1/2}$ to t^{k+1} equation is approximated by

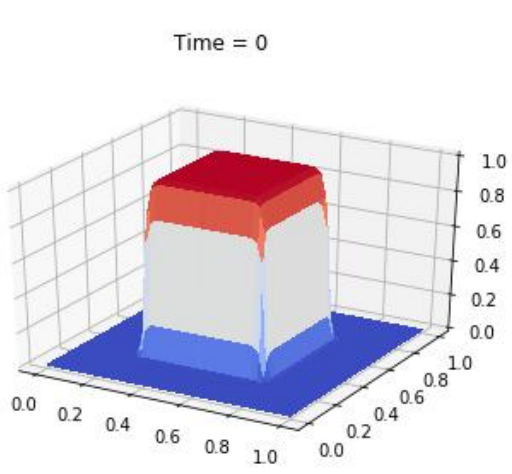
$$\frac{U_{i,j}^{k+1} - U_{i,j}^{k+1/2}}{\Delta t/2} = D\left(\frac{U_{i+1,j}^{k+1} - 2U_{i,j}^{k+1} + U_{i-1,j}^{k+1}}{h^2} + \frac{U_{i,j+1}^{k+1/2} - 2U_{i,j}^{k+1/2} + U_{i,j-1}^{k+1/2}}{h^2}\right)$$

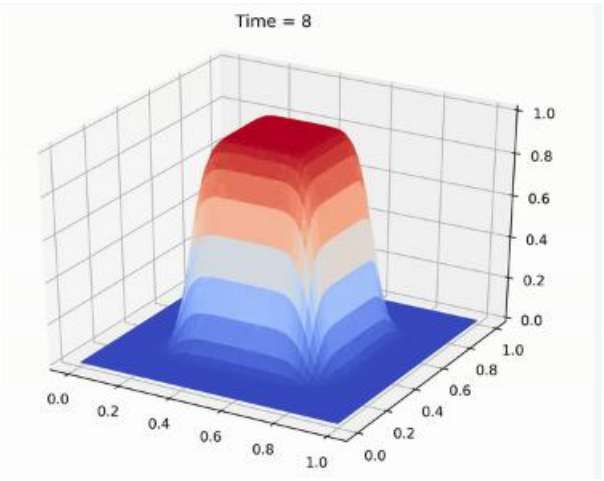
Rearranging Gives

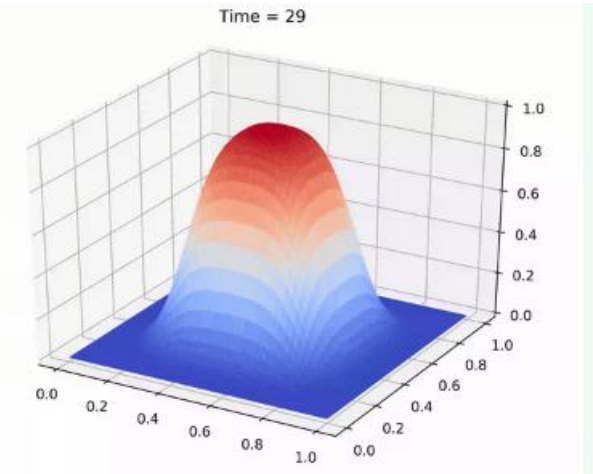
$$-rU_{i-1,j}^{k+1} + (1+2r)U_{i,j}^{k+1} - rU_{i+1,j}^{k+1} = rU_{i,j-1}^{k+1/2} + (1-2r)U_{i,j}^{k+1/2} + U_{i,j+1}^{k+1/2}$$

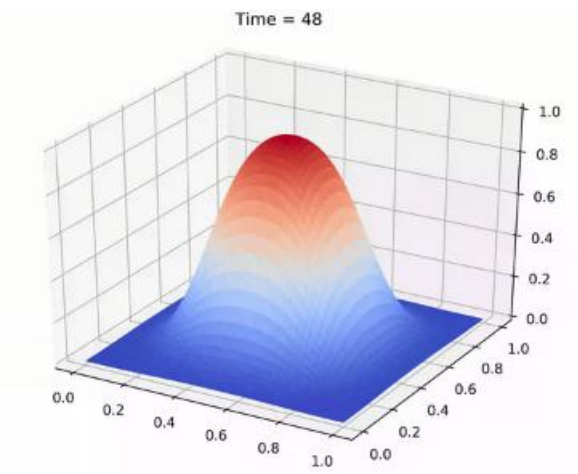
which is implicit.
When writing for a 2-dimensional grid, the equation results in a tri-diagonal system

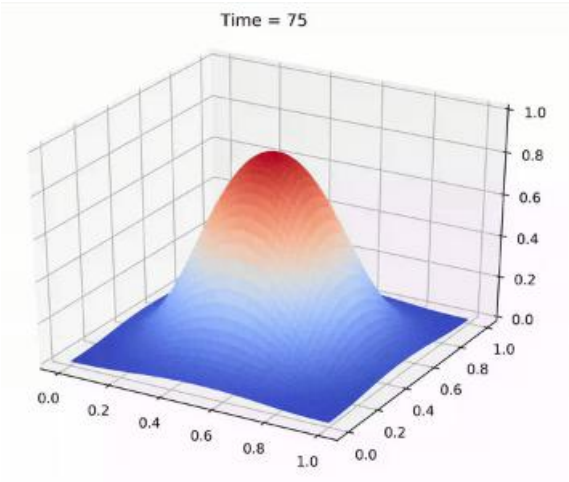

t^{k+1}
$t^{k+\frac{1}{2}}$
t^k
y_{i+1}
y_i
y_{i-1}
x_{i-1}
x_i
x_{i+1}
explicit
implicit

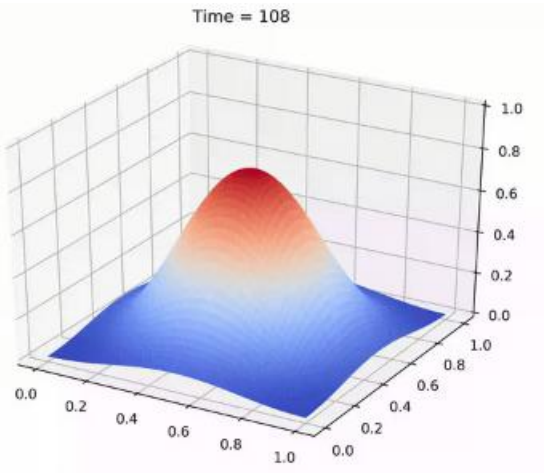

Time = 0


Time = 8


Time = 29


Time = 48





Code

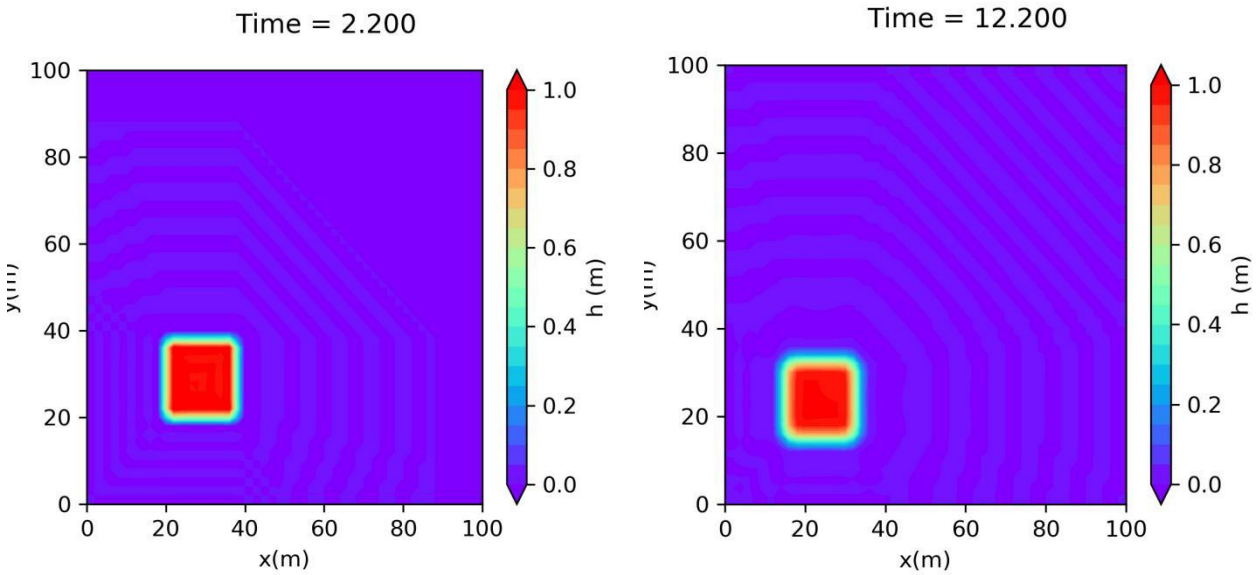
```
1.  import numpy as np
2.  from scipy import sparse
3.  import matplotlib.pyplot as plt
4.  from mpl_toolkits.mplot3d import Axes3D
5.  from matplotlib import cm
6.
7.  class ADImethod:
8.
9.      def __init__(self, M, maxt, D):
10.         self.M = M
11.         self.x0 = 0
12.         self.xf = 1
13.         self.y0 = 0
14.         self.yf = 1
15.         self.maxt = maxt
16.         self.dt = 0.01
17.         self.D = D
18.         self.h = 1/self.M
19.         self.r = self.D*self.dt/(2*self.h**2)
20.         self.generateGrid()
21.         self.lhsMatrixA()
22.         self.rhsMatrixA()
23.
24.
25.     def generateGrid(self):
26.         self.X,  self.Y  =  np.meshgrid(np.linspace(self.x0,  self.xf,
   self.M), np.linspace(self.y0, self.yf,self. M))
27.         ic01 = np.logical_and(self.X >= 1/4, self.X <= 3/4)
28.         ic02 = np.logical_and(self.Y >= 1/4, self.Y <= 3/4)
29.         ic0 = np.multiply(ic01, ic02)
30.         self.U = ic0*1
31.
```

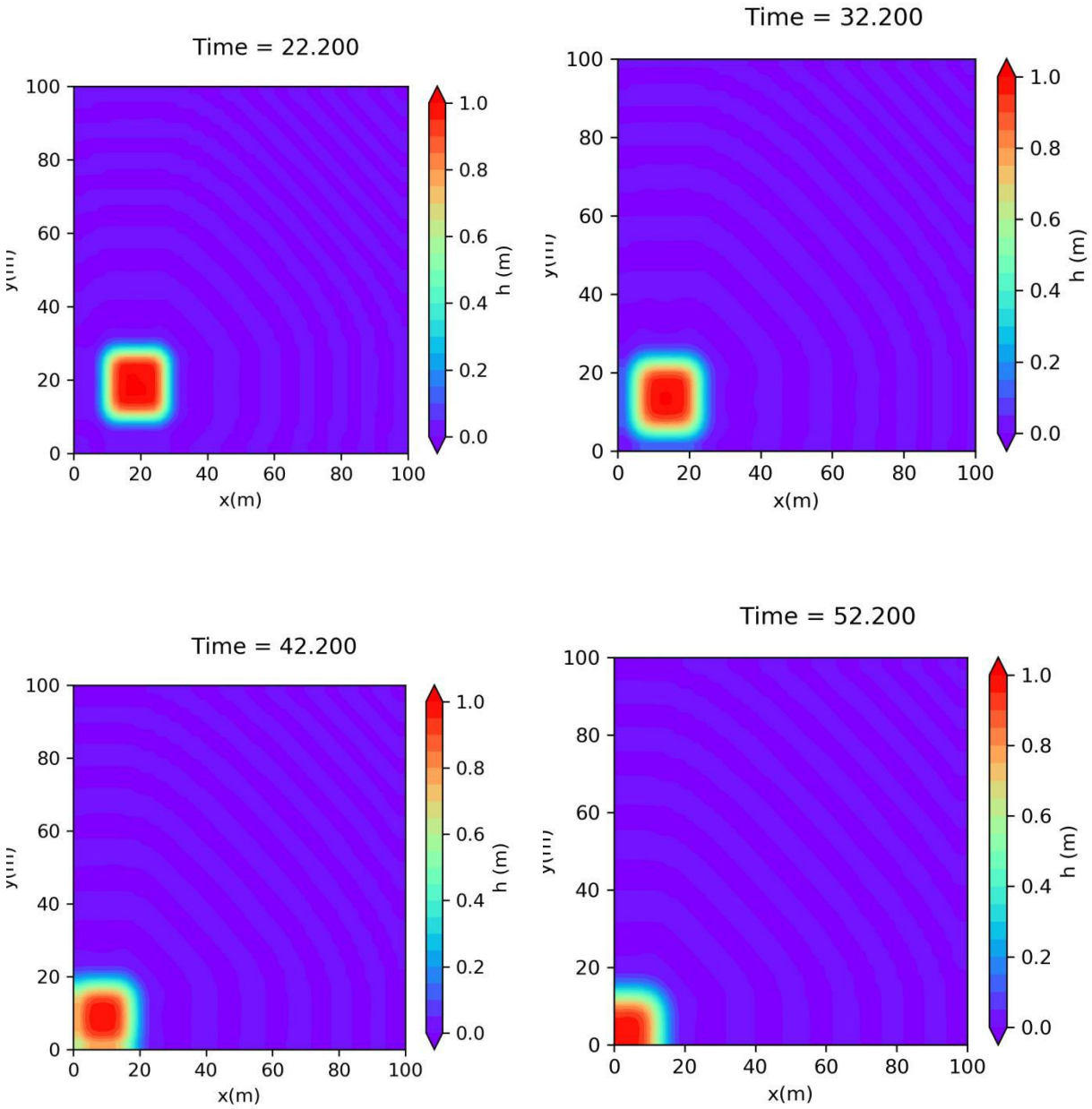
```
32.
33.     def lhsMatrixA(self):
34.         maindiag = (1+2*self.r)*np.ones((1, self.M))
35.         offdiag = -self.r*np.ones((1, self.M-1))
36.         a = maindiag.shape[1]
37.         diagonals = [maindiag, offdiag, offdiag]
38.         Lx = sparse.diags(diagonals, [0, -1, 1], shape=(a, a)).toarray()
39.         Ix = sparse.identity(self.M).toarray()
40.         self.A = sparse.kron(Ix, Lx).toarray()
41.
42.         pos1 = np.arange(0,self.M**2,self.M)
43.
44.         for i in range(len(pos1)):
45.             self.A[pos1[i], pos1[i]] = 1 + self.r
46.
47.         pos2 = np.arange(self.M-1, self.M**2, self.M)
48.
49.         for j in range(len(pos2)):
50.             self.A[pos2[j], pos2[j]] = 1 + self.r
51.
52.
53.     def rhsMatrixA(self):
54.         maindiag = (1-self.r)*np.ones((1, self.M))
55.         offdiag = self.r*np.ones((1, self.M-1))
56.         a = maindiag.shape[1]
57.         diagonals = [maindiag, offdiag, offdiag]
58.         Rx = sparse.diags(diagonals, [0, -1, 1], shape=(a, a)).toarray()
59.         Ix = sparse.identity(self.M).toarray()
60.         self.A_rhs = sparse.kron(Rx, Ix).toarray()
61.
62.         pos3 = np.arange(self.M, self.M**2-self.M)
63.
64.         for k in range(len(pos3)):
65.             self.A_rhs[pos3[k], pos3[k]] = 1 - 2*self.r
66.
67.
68.     def solve_and_plot(self):
69.         fig = plt.figure()
70.         ax = fig.gca(projection='3d')
71.         ax.set_zlim(0, 1)
72.         tc = 0
73.         nstep = round(self.maxt/self.h)
74.         while tc < nstep:
75.             b1 = np.flipud(self.U).reshape(self.M**2, 1)
76.             sol = np.linalg.solve(self.A, np.matmul(self.A_rhs, b1))
```

```
            self.U = np.flipud(sol).reshape(self.M, self.M)

            b2 = np.flipud(self.U).reshape(self.M**2, 1)
            sol = np.linalg.solve(self.A, np.matmul(self.A_rhs, b2))
            self.U = np.flipud(sol).reshape(self.M, self.M)
            tc += 1

        ax.plot_surface(self.X, self.Y, self.U, cmap=cm.coolwarm,
                        linewidth=0, antialiased=False)
        plt.tight_layout()
        plt.show()

def main():
    simulator = ADImethod(20, 2.5, 0.01)
    simulator.solve_and_plot()

if __name__ == "__main__":
    main()
```

Advection-Diffusion Equation

CIP for advection equation and Central difference method for diffusion equation





Code

```
"""

@author: Wentao Gong
"""
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
import csv
import os
import glob
import re


```

```
xmax  = 100       # length of river reach (m)
ymax  = 100

nx = 50    # number of grid
ny = 50
dx = xmax/float(nx)    # size of grid
dy = ymax/float(ny)    # size of grid
dt = 0.2    # computational time step (sec)
u  = -0.5    # advection velocity
v  = -0.5
D  = 0.1   # diffusion coefficient
h0 = 1.
i_cip = 1

dx2 = dx*dx
dx3 = dx2*dx
dy2 = dy*dy
dy3 = dy2*dy

x    = np.zeros((ny+1,nx+1))
y    = np.zeros((ny+1,nx+1))
h    = np.zeros((ny+1,nx+1))
wh   = np.zeros((ny+1,nx+1))
ghx  = np.zeros((ny+1,nx+1))
wghx = np.zeros((ny+1,nx+1))
ghy  = np.zeros((ny+1,nx+1))
wghy = np.zeros((ny+1,nx+1))
for j in np.arange(ny+1):
    x[j][0] = 0.

    for i in np.arange(1,nx+1):
        x[j][i] = x[j][i-1]+dx

for i in np.arange(nx+1):
    y[0][i] = 0.

    for j in np.arange(1,ny+1):
        y[j][i] = y[j-1][i]+dy

for j in np.arange(ny+1):
    for i in np.arange(nx+1):
        if 0.2*nx < i < 0.4*nx and 0.2*ny < j < 0.4*ny:
            h[j][i] = h0
        else:
            h[j][i]  = 0
```

```
59.
60. for j in np.arange(1,ny):
61.     for i in np.arange(1,nx):
62.         ghx[j][i] = (h[j][i+1]-h[j][i-1])*0.5/dx
63.         ghy[j][i] = (h[j+1][i]-h[j-1][i])*0.5/dy
64.
65. for j in np.arange(1,ny):
66.     ghx[j][ 0] = ghx[j][   1]
67.     ghx[j][ny] = ghx[j][ny-1]
68.     ghy[j][ 0] = ghy[j][   1]
69.     ghy[j][ny] = ghy[j][ny-1]
70.
71. for i in np.arange(nx+1):
72.     ghx[ 0][i] = ghx[   1][i]
73.     ghx[ny][i] = ghx[ny-1][i]
74.     ghy[ 0][i] = ghy[   1][i]
75.     ghy[ny][i] = ghy[ny-1][i]
76.
77.
78. time = 0.
79. optime = 0.
80. etime = 150.
81. tuk = 2.
82.
83. n = 0
84. while time<etime:
85.    for j in np.arange(1,ny):
86.         for i in np.arange(1,nx):
87.             wh[j][i] = h[j][i]+D*dt*( (h[j][i+1]-2.0*h[j][i]+h[j][
    i-1])/dx**2.0+(h[j+1][i]-2.0*h[j][i]+h[j-1][i])/dy**2.0 )
88.
89.     for j in np.arange(1,ny):
90.         wh[j][ 0] = wh[j][   1]
91.         wh[j][ny] = wh[j][ny-1]
92.
93.     for i in np.arange(nx+1):
94.         wh[ 0][i] = wh[   1][i]
95.         wh[ny][i] = wh[ny-1][i]
96.
97.     for j in np.arange(1,ny):
98.         for i in np.arange(1,nx):
99.             wghx[j][i] = ghx[j][i]+(-wh[j][i-1]+wh[j][i+1]+h[j][i-
    1]-h[j][i+1])*0.5/dx
100.                wghy[j][i] = ghy[j][i]+(-wh[j-1][i]+wh[j+1][i]+h[j-
    1][i]-h[j+1][i])*0.5/dy
```

```
101.
102.       for j in np.arange(1,ny):
103.           wghx[j][ 0] = wghx[j][   1]
104.           wghx[j][ny] = wghx[j][ny-1]
105.           wghy[j][ 0] = wghy[j][   1]
106.           wghy[j][ny] = wghy[j][ny-1]
107.
108.       for i in np.arange(nx+1):
109.           wghx[ 0][i] = wghx[   1][i]
110.           wghx[ny][i] = wghx[ny-1][i]
111.           wghy[ 0][i] = wghy[   1][i]
112.           wghy[ny][i] = wghy[ny-1][i]
113.
114.       if i_cip==0:
115.
116.           for j in np.arange(1,ny):
117.               for i in np.arange(1,nx):
118.                   udhdx = ((u+abs(u))*(wh[j][i]-wh[j][i-1]) + (u-
   abs(u))*(wh[j][i+1]-wh[j][i]))*0.5/dx
119.                   vdhdy = ((v+abs(v))*(wh[j][i]-wh[j-1][i]) + (v-
   abs(v))*(wh[j+1][i]-wh[j][i]))*0.5/dy
120.
121.                   h[j][i] = wh[j][i] - (udhdx+vdhdy)*dt
122.
123.       else:
124.   for j in np.arange(1,ny):
125.               for i in np.arange(1,nx):
126.
127.                   xx = -u*dt
128.                   yy = -v*dt
129.                   isn = int(np.sign(u))
130.                   jsn = int(np.sign(v))
131.                   fis = float(isn)
132.                   fjs = float(jsn)
133.                   im1 = i-isn
134.                   jm1 = j-jsn
135.
136.                   a1   = ((wghx[j][im1]+wghx[j][i])*dx*fis-2.0*(wh
   [j][i]-wh[j][im1]))/(dx3*fis)
137.                   e1   = (3.0*(wh[j][im1]-wh[j][i])+(wghx[j][im1]+
   2.0*wghx[j][i])*dx*fis)/dx2
138.                   b1   = ((wghy[jm1][i]+wghy[j][i])*dy*fjs-2.0*(wh
   [j][i]-wh[jm1][i]))/(dy3*fjs)
139.                   f1   = (3.0*(wh[jm1][i]-wh[j][i])+(wghy[jm1][i]+
   2.0*wghy[j][i])*dy*fjs)/dy2
```

```
                    tmp  = wh[j][i]-wh[jm1][i]-wh[j][im1]+wh[jm1][im
  1]
                    tmq  = wghy[j][im1]-wghy[j][i]
                    d1   = (-tmp-tmq*dy*fjs)/(dx*dy2*fis)
                    c1   = (-tmp-(wghx[jm1][i]-wghx[j][i])*dx*fis)/(
  dx2*dy*fjs)
                    g1   = (-tmq+c1*dx2)/(dx*fis)

                    h[j][i]   = ((a1*xx+c1*yy+e1)*xx+g1*yy+wghx[j][i]
  )*xx+((b1*yy+d1*xx+f1)*yy+wghy[j][i])*yy+wh[j][i]
                    ghx[j][i] = (3.0*a1*xx+2.0*(c1*yy+e1))*xx+(d1*yy
  +g1)*yy+wghx[j][i]
                    ghy[j][i] = (3.0*b1*yy+2.0*(d1*xx+f1))*yy+(c1*xx
  +g1)*xx+wghy[j][i]


        for j in np.arange(1,ny):
            h[j][ 0] = h[j][   1]
            h[j][ny] = h[j][ny-1]

        for i in np.arange(nx+1):
            h[ 0][i] = h[   1][i]
            h[ny][i] = h[ny-1][i]

        for j in np.arange(1,ny):
            ghx[j][ 0] = ghx[j][   1]
            ghx[j][ny] = ghx[j][ny-1]
            ghy[j][ 0] = ghy[j][   1]
            ghy[j][ny] = ghy[j][ny-1]

        for i in np.arange(nx+1):
            ghx[ 0][i] = ghx[   1][i]
            ghx[ny][i] = ghx[ny-1][i]
            ghy[ 0][i] = ghy[   1][i]
            ghy[ny][i] = ghy[ny-1][i]
    if optime>tuk:
            print("Time= ", time, "sec")

            plt.figure(figsize=(4,4*ymax/xmax*0.9))
            plt.xlabel('x(m)')
            plt.ylabel('y(m)')
            plt.axis([0,xmax,0.,ymax])

            levels = np.linspace(0,h0,21)
            labels = np.linspace(0,h0,6)
```

```
180.
181.            plt.contourf(x, y, h, levels, cmap=cm.rainbow, extend="b
  oth")
182.            plt.colorbar(ticks=labels, label='h (m)')
183.
184.            nnn = str(n)
185.
186.            plt.savefig('Figure' + nnn.zfill(4) +".jpg", dpi=300)
187.            plt.close()
188.
189.            n += 1
190.
191.            optime = optime-tuk
192.
193.        optime+=dt
194.        time+=dt
```

Navier_Stokes Equations

Cavity Flow with Navier–Stokes

The momentum equation in vector form for a velocity field $\vec{v}$ is:

$$\frac{\partial v}{\partial t} + (v \cdot \nabla)\vec{v} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \vec{v}$$

This represents three scalar equations, one for each velocity component (u, v, w)

But we will solve it in two dimensions, so there will be two scalar equations.
two equations for the velocity components (u,v) and one equation for pressure:

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = -\rho\left(\frac{\partial u}{\partial x}\frac{\partial u}{\partial x} + 2\frac{\partial u}{\partial y}\frac{\partial v}{\partial x} + \frac{\partial v}{\partial y}\frac{\partial v}{\partial y}\right)$$

Discretized equations

First, let's discretize the u-momentum equation, as follows:

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} + u_{i,j}^{n}\frac{u_{i,j}^{n} - u_{i-1,j}^{n}}{\Delta x} + v_{i,j}^{n}\frac{v_{i,j}^{n} - v_{i,j-1}^{n}}{\Delta y} =$$

$$-\frac{1}{\rho}\frac{p_{i+1,j}^{n} - p_{i-1,j}^{n}}{2\Delta x} + \nu\left(\frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} + \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^2}\right)$$

Similarly for the v -momentum equation:

$$\frac{v_{i,j}^{n+1}-v_{i,j}^{n}}{\Delta t}+u_{i,j}^{n}\frac{v_{i,j}^{n}-v_{i-1,j}^{n}}{\Delta x}+v_{i,j}^{n}\frac{v_{i,j}^{n}-v_{i,j-1}^{n}}{\Delta y}=$$
$$-\frac{1}{\rho}\frac{p_{i,j+1}^{n}-p_{i,j-1}^{n}}{2\Delta y}+\nu\left(\frac{v_{i+1,j}^{n}-2v_{i,j}^{n}+u_{i-1,j}^{n}}{\Delta x^{2}}+\frac{v_{i,j+1}^{n}-2v_{i,j}^{n}+v_{i,j-1}^{n}}{\Delta y^{2}}\right)$$

Finally, the discretized pressure-Poisson equation can be written thus:

$$\frac{p_{i+1,j}^{n}-2p_{i,j}^{n}+p_{i-1,j}^{n}}{\Delta x^{2}}+\frac{p_{i,j+1}^{n}-2p_{i,j}^{n}+p_{i,j-1}^{n}}{\Delta y^{2}}=$$
$$\rho\left[\frac{1}{\Delta t}\left(\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}+\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y}\right)\right.$$
$$\left.-\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}-2\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}\frac{u_{i,j+1}-u_{i,j+1}}{2\Delta y}-\frac{u_{i,j+1}-u_{i,j+1}}{2\Delta y}\frac{u_{i,j+1}-u_{i,j+1}}{2\Delta y}\right]$$

let's rearrange the equations in the way that the iterations need to proceed in the code. First, the momentum equations for the velocity at the next time step.
The momentum equation in the u direction:

$$\begin{aligned}u_{i,\,j}^{n+1}=u_{i,\,j}^{n}&-u_{i,\,j}^{n}\frac{\Delta t}{\Delta x}\left(u_{i,\,j}^{n}-u_{i-1,\,j}^{n}\right)-v_{i,\,j}^{n}\frac{\Delta t}{\Delta y}\left(u_{i,\,j}^{n}-u_{i,\,j-1}^{n}\right)\\&-\frac{\Delta t}{\rho 2\Delta x}\left(p_{i+1,\,j}^{n}-p_{i-1,\,j}^{n}\right)\\&+\nu\left(\frac{\Delta t}{\Delta x^{2}}\left(u_{i+1,\,j}^{n}-2u_{i,\,j}^{n}+u_{i-1,\,j}^{n}\right)+\frac{\Delta t}{\Delta y^{2}}\left(u_{i,\,j+1}^{n}-2u_{i,\,j}^{n}+u_{i,\,j-1}^{n}\right)\right)\end{aligned}$$

The momentum equation in the y direction:

$$\begin{aligned}v_{i,\,j}^{n+1}=v_{i,\,j}^{n}&-u_{i,\,j}^{n}\frac{\Delta t}{\Delta x}\left(v_{i,\,j}^{n}-v_{i-1,\,j}^{n}\right)-v_{i,\,j}^{n}\frac{\Delta t}{\Delta y}\left(v_{i,\,j}^{n}-v_{i,\,j-1}^{n}\right)\\&-\frac{\Delta t}{\rho 2\Delta y}\left(p_{i,\,j+1}^{n}-p_{i,\,j-1}^{n}\right)\\&+\nu\left(\frac{\Delta t}{\Delta x^{2}}\left(v_{i+1,\,j}^{n}-2v_{i,\,j}^{n}+v_{i-1,\,j}^{n}\right)+\frac{\Delta t}{\Delta y^{2}}\left(v_{i,\,j+1}^{n}-2v_{i,\,j}^{n}+v_{i,\,j-1}^{n}\right)\right)\end{aligned}$$

we rearrange the pressure-Poisson equation:

$$\begin{aligned}p_{i,\,j}^{n}=&\frac{\left(p_{i+1,\,j}^{n}+p_{i-1,\,j}^{n}\right)\Delta y^{2}+\left(p_{i,\,j+1}^{n}+p_{i,\,j-1}^{n}\right)\Delta x^{2}}{2\left(\Delta x^{2}+\Delta y^{2}\right)}\\&-\frac{\rho\Delta x^{2}\Delta y^{2}}{2\left(\Delta x^{2}+\Delta y^{2}\right)}\\&\times\left[\frac{1}{\Delta t}\left(\frac{u_{i+1,\,j}-u_{i-1,\,j}}{2\Delta x}+\frac{v_{i,\,j+1}-v_{i,\,j-1}}{2\Delta y}\right)-\frac{u_{i+1,\,j}-u_{i-1,\,j}}{2\Delta x}\frac{u_{i+1,\,j}-u_{i-1,\,j}}{2\Delta x}-2\frac{u_{i,\,j+1}-u_{i,\,j-1}}{2\Delta y}\frac{v_{i+1,\,j}-v_{i-1,\,j}}{2\Delta x}-\frac{v_{i,\,j+1}-v_{i,\,j-1}}{2\Delta y}\frac{v_{i,\,j+1}-v_{i,\,j-1}}{2\Delta y}\right]\end{aligned}$$

The initial condition is u,v,p=0 everywhere, and the boundary conditions are:

$u = 1$ at $y = 2$

$u, v = 0$ on the on the boundaries

$$\frac{\partial p}{\partial y} = 0 \text{ at } y = 0$$

P=0 at $y = 2$

$$\frac{\partial p}{\partial x} = 0 \text{ at } x = 0,2$$

Implementing Cavity Flow

```
import numpy as np
#%matplotlib inline
#%config InlineBackend.figure_format = 'svg'
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib import cm
plt.rcParams['animation.html'] = 'html5'
nx = 41
ny = 41
nt = 500
nit = 50
c = 1
dx = 2 / (nx - 1)
dy = 2 / (ny - 1)
x = numpy.linspace(0, 2, nx)
y = numpy.linspace(0, 2, ny)
X, Y = numpy.meshgrid(x, y)

rho = 1
nu = .1
dt = .001

u = numpy.zeros((ny, nx))
v = numpy.zeros((ny, nx))
p = numpy.zeros((ny, nx))
b = numpy.zeros((ny, nx))
```

The function build_up_b below represents the contents of the square brackets, so that the entirety of the PPE is slightly more manageable.

```
def build_up_b(b, rho, dt, u, v, dx, dy):
```

```
28.         b[1:-1, 1:-1] = (rho * (1 / dt *
29.                                 ((u[1:-1, 2:] - u[1:-1, 0:-2]) /
30.                                   (2 * dx) + (v[2:, 1:-1] - v[0:-2, 1:-1]) / (2 * dy)) -
31.                                 ((u[1:-1, 2:] - u[1:-1, 0:-2]) / (2 * dx))**2 -
32.                                    2 * ((u[2:, 1:-1] - u[0:-2, 1:-1]) / (2 * dy) *
33.                                          (v[1:-1, 2:] - v[1:-1, 0:-2]) / (2 * dx))-
34.                                         ((v[2:, 1:-1] - v[0:-2, 1:-1]) / (2 * dy))**2))
35.
36.         return b
```

The function pressure_poisson is also defined to help segregate the different rounds of calculations. Note the presence of the pseudo-time variable nit. This sub-iteration in the Poisson calculation helps ensure a divergence-free field.

```
37.   def pressure_poisson(p, dx, dy, b):
38.         pn = numpy.empty_like(p)
39.         pn = p.copy()
40.
41.         for q in range(nit):
42.               pn = p.copy()
43.               p[1:-1, 1:-1] = (((pn[1:-1, 2:] + pn[1:-1, 0:-2]) * dy**2 +
44.                                       (pn[2:, 1:-1] + pn[0:-2, 1:-1]) * dx**2) /
45.                                       (2 * (dx**2 + dy**2)) -
46.                                       dx**2 * dy**2 / (2 * (dx**2 + dy**2)) *
47.                                       b[1:-1,1:-1])
48.
49.               p[:, -1] = p[:, -2] # dp/dx = 0 at x = 2
50.               p[0, :] = p[1, :]     # dp/dy = 0 at y = 0
51.               p[:, 0] = p[:, 1]     # dp/dx = 0 at x = 0
52.               p[-1, :] = 0             # p = 0 at y = 2
53.
54.
         return p
```

Finally, the rest of the cavity flow equations are wrapped inside the function cavity_flow, allowing us to easily plot the results of the cavity flow solver for different lengths of time.

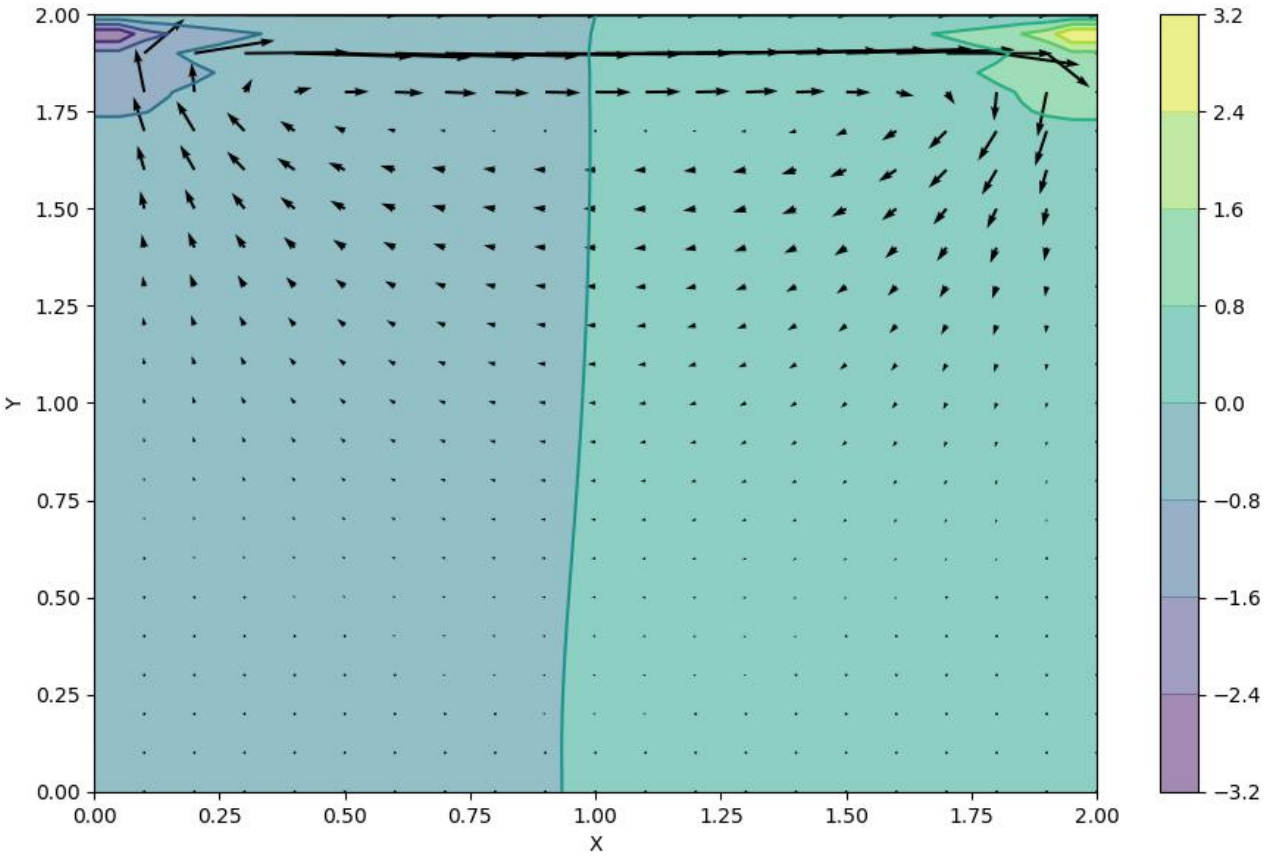
```
55.   def cavity_flow(nt, u, v, dt, dx, dy, p, rho, nu):
56.         un = numpy.empty_like(u)
57.         vn = numpy.empty_like(v)
58.         b = numpy.zeros((ny, nx))
59.
60.         for n in range(nt):
61.               un = u.copy()
62.               vn = v.copy()
63.
64.               b = build_up_b(b, rho, dt, u, v, dx, dy)
65.               p = pressure_poisson(p, dx, dy, b)
66.
```

```
        u[1:-1, 1:-1] = (un[1:-1, 1:-1]-
                         un[1:-1, 1:-1] * dt / dx *
                        (un[1:-1, 1:-1] - un[1:-1, 0:-2]) -
                         vn[1:-1, 1:-1] * dt / dy *
                        (un[1:-1, 1:-1] - un[0:-2, 1:-1]) -
                         dt / (2 * rho * dx) * (p[1:-1, 2:] - p[1:-1, 0:-2]) +
                         nu * (dt / dx**2 *
                        (un[1:-1, 2:] - 2 * un[1:-1, 1:-1] + un[1:-1, 0:-2]) +
                         dt / dy**2 *
                        (un[2:, 1:-1] - 2 * un[1:-1, 1:-1] + un[0:-2, 1:-1])))

        v[1:-1,1:-1] = (vn[1:-1, 1:-1] -
                         un[1:-1, 1:-1] * dt / dx *
                        (vn[1:-1, 1:-1] - vn[1:-1, 0:-2]) -
                         vn[1:-1, 1:-1] * dt / dy *
                        (vn[1:-1, 1:-1] - vn[0:-2, 1:-1]) -
                         dt / (2 * rho * dy) * (p[2:, 1:-1] - p[0:-2, 1:-1]) +
                         nu * (dt / dx**2 *
                        (vn[1:-1, 2:] - 2 * vn[1:-1, 1:-1] + vn[1:-1, 0:-2]) +
                         dt / dy**2 *
                        (vn[2:, 1:-1] - 2 * vn[1:-1, 1:-1] + vn[0:-2, 1:-1])))

        u[0, :]  = 0
        u[:, 0]  = 0
        u[:, -1] = 0
        u[-1, :] = 1    # set velocity on cavity lid equal to 1
        v[0, :]  = 0
        v[-1, :] = 0
        v[:, 0]  = 0
        v[:, -1] = 0


    return u, v, p
```

Let's start with nt = 100 and see what the solver gives us:

```
u = numpy.zeros((ny, nx))
v = numpy.zeros((ny, nx))
p = numpy.zeros((ny, nx))
b = numpy.zeros((ny, nx))
nt = 100

u, v, p = cavity_flow(nt, u, v, dt, dx, dy, p, rho, nu)
fig = pyplot.figure(figsize=(11,7), dpi=100)
# plotting the pressure field as a contour
pyplot.contourf(X, Y, p, alpha=0.5, cmap=cm.viridis)
```

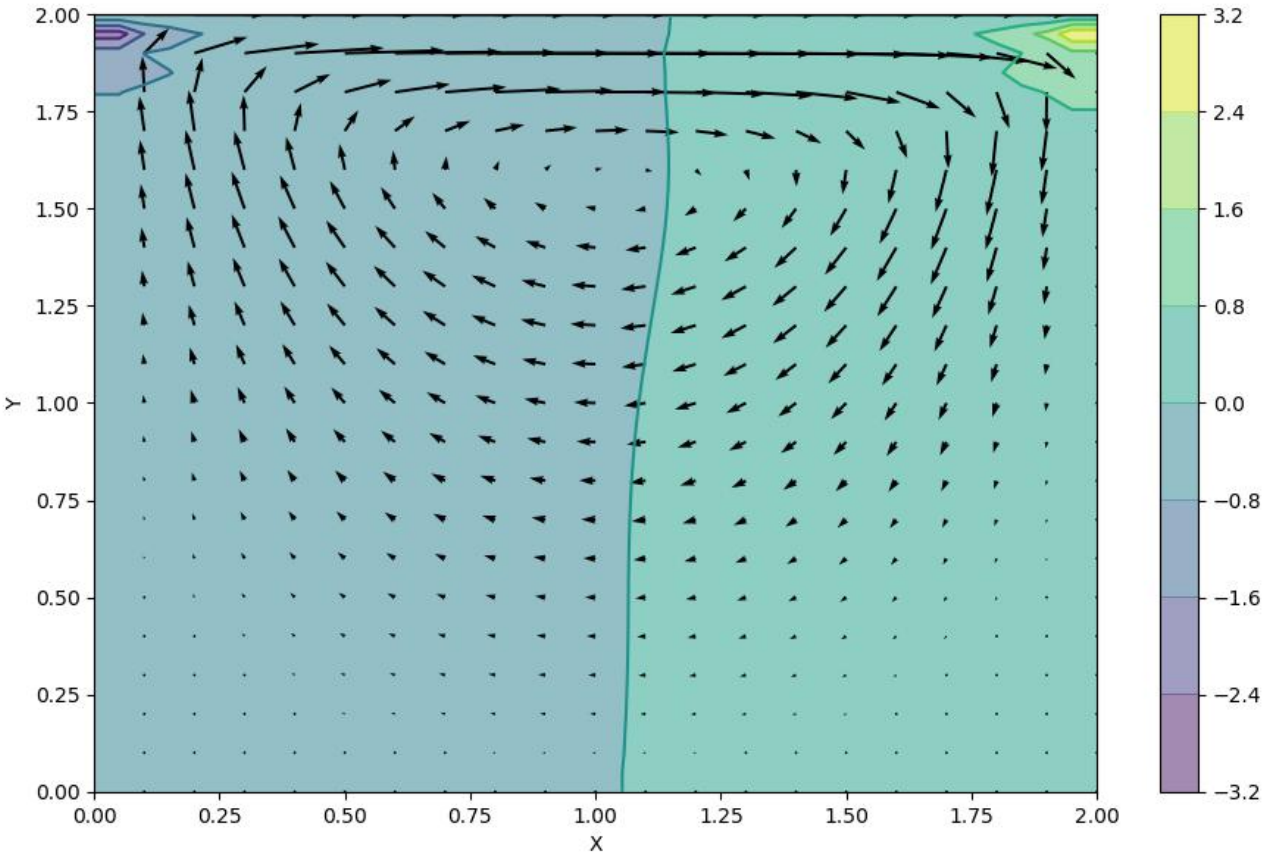
```
pyplot.colorbar()
# plotting the pressure field outlines
pyplot.contour(X, Y, p, cmap=cm.viridis)
# plotting velocity field
pyplot.quiver(X[::2, ::2], Y[::2, ::2], u[::2, ::2], v[::2, ::2])
pyplot.xlabel('X')
pyplot.ylabel('Y');
```



You can see that two distinct pressure zones are forming and that the spiral pattern expected from lid-driven cavity flow is beginning to form. Experiment with different values of nt to see how long the system takes to stabilize.

```
u = numpy.zeros((ny, nx))
v = numpy.zeros((ny, nx))
p = numpy.zeros((ny, nx))
b = numpy.zeros((ny, nx))
nt = 700
u, v, p = cavity_flow(nt, u, v, dt, dx, dy, p, rho, nu)
fig = pyplot.figure(figsize=(11,7), dpi=100)
# plotting the pressure field as a contour
pyplot.contourf(X, Y, p, alpha=0.5, cmap=cm.viridis)
```

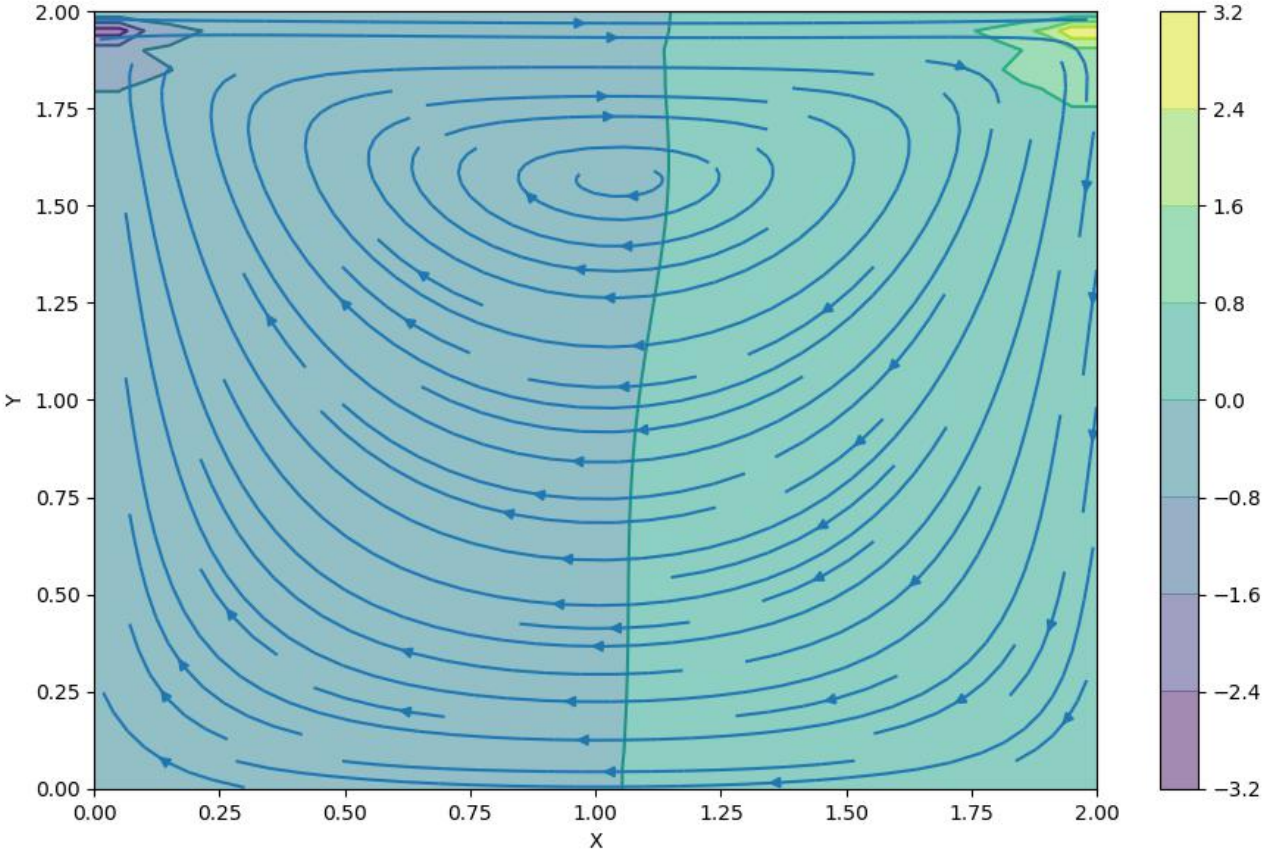
```
pyplot.colorbar()
# plotting the pressure field outlines
pyplot.contour(X, Y, p, cmap=cm.viridis)
# plotting velocity field
pyplot.quiver(X[::2, ::2], Y[::2, ::2], u[::2, ::2], v[::2, ::2])
pyplot.xlabel('X')
pyplot.ylabel('Y');
```



The quiver plot shows the magnitude of the velocity at the discrete points in the mesh grid we created. (We're actually only showing half of the points because otherwise it's a bit of a mess. The X[::2, ::2] syntax above is a convenient way to ask for every other point.)

Another way to visualize the flow in the cavity is to use a streamplot:

```
fig = pyplot.figure(figsize=(11, 7), dpi=100)
pyplot.contourf(X, Y, p, alpha=0.5, cmap=cm.viridis)
pyplot.colorbar()
pyplot.contour(X, Y, p, cmap=cm.viridis)
pyplot.streamplot(X, Y, u, v)
pyplot.xlabel('X')
pyplot.ylabel('Y');
```



Channel Flow with Navier–Stokes

add a source term to the u -momentum equation

Discretized equations

The u -momentum equation:

$$\frac{u_{i,\,j}^{n+1}-u_{i,\,j}^{n}}{\Delta t}+u_{i,\,j}^{n}\frac{u_{i,\,j}^{n}-u_{i-1,\,j}^{n}}{\Delta x}+v_{i,\,j}^{n}\frac{u_{i,\,j}^{n}-u_{i,\,j-1}^{n}}{\Delta y}=$$
$$-\frac{1}{\rho}\frac{p_{i+1,\,j}^{n}-p_{i-1,\,j}^{n}}{2\Delta x}$$
$$+\nu\left(\frac{u_{i+1,\,j}^{n}-2u_{i,\,j}^{n}+u_{i-1,\,j}^{n}}{\Delta x^{2}}+\frac{u_{i,\,j+1}^{n}-2u_{i,\,j}^{n}+u_{i,\,j-1}^{n}}{\Delta y^{2}}\right)+F_{i,\,j}$$

The -momentum equation:

$$\frac{v_{i,\,j}^{n+1}-v_{i,\,j}^{n}}{\Delta t}+u_{i,\,j}^{n}\frac{v_{i,\,j}^{n}-v_{i-1,\,j}^{n}}{\Delta x}+v_{i,\,j}^{n}\frac{v_{i,\,j}^{n}-v_{i,\,j-1}^{n}}{\Delta y}=$$
$$-\frac{1}{\rho}\frac{p_{i,\,j+1}^{n}-p_{i,\,j-1}^{n}}{2\Delta y}$$
$$+\nu\left(\frac{v_{i+1,\,j}^{n}-2v_{i,\,j}^{n}+v_{i-1,\,j}^{n}}{\Delta x^{2}}+\frac{v_{i,\,j+1}^{n}-2v_{i,\,j}^{n}+v_{i,\,j-1}^{n}}{\Delta y^{2}}\right)$$

And the pressure equation:

$$\frac{p^n_{i+1,\,j}-2p^n_{i,\,j}+p^n_{i-1,\,j}}{\Delta x^2}+\frac{p^n_{i,\,j+1}-2p^n_{i,\,j}+p^n_{i,\,j-1}}{\Delta y^2}=$$
$$\rho\left[\frac{1}{\Delta t}\left(\frac{u_{i+1,\,j}-u_{i-1,\,j}}{2\Delta x}+\frac{v_{i,\,j+1}-v_{i,\,j-1}}{2\Delta y}\right)-\frac{u_{i+1,\,j}-u_{i-1,\,j}}{2\Delta x}\frac{u_{i+1,\,j}-u_{i-1,\,j}}{2\Delta x}-2\frac{u_{i,\,j+1}-u_{i,\,j-1}}{2\Delta y}\frac{v_{i+1,\,j}-v_{i-1,\,j}}{2\Delta x}-\frac{v_{i,\,j+1}-v_{i,\,j-1}}{2\Delta y}\frac{v_{i,\,j+1}-v_{i,\,j-1}}{2\Delta y}\right]$$

we need to re-arrange these equations to the form we need in the code to make the iterations proceed. For the u- and v momentum equations, we isolate the velocity at time step n+1:

$$\begin{aligned}v^{n+1}_{i,\,j}=v^n_{i,\,j}&-u^n_{i,\,j}\frac{\Delta t}{\Delta x}\left(v^n_{i,\,j}-v^n_{i-1,\,j}\right)-v^n_{i,\,j}\frac{\Delta t}{\Delta y}\left(v^n_{i,\,j}-v^n_{i,\,j-1}\right)\\&-\frac{\Delta t}{\rho 2\Delta y}\left(p^n_{i,\,j+1}-p^n_{i,\,j-1}\right)\\&+\nu\left[\frac{\Delta t}{\Delta x^2}\left(v^n_{i+1,\,j}-2v^n_{i,\,j}+v^n_{i-1,\,j}\right)+\frac{\Delta t}{\Delta y^2}\left(v^n_{i,\,j+1}-2v^n_{i,\,j}+v^n_{i,\,j-1}\right)\right]\end{aligned}$$

And for the pressure equation, we isolate the $p^n_{i,\,j}$ term to iterate in pseudo-time:

$$p^n_{i,\,j}=\frac{\left(p^n_{i+1,\,j}+p^n_{i-1,\,j}\right)\Delta y^2+\left(p^n_{i,\,j+1}+p^n_{i,\,j-1}\right)\Delta x^2}{2(\Delta x^2+\Delta y^2)}$$
$$-\frac{\rho\Delta x^2\Delta y^2}{2(\Delta x^2+\Delta y^2)}$$
$$\times\left[\frac{1}{\Delta t}\left(\frac{u_{i+1,\,j}-u_{i-1,\,j}}{2\Delta x}+\frac{v_{i,\,j+1}-v_{i,\,j-1}}{2\Delta y}\right)-\frac{u_{i+1,\,j}-u_{i-1,\,j}}{2\Delta x}\frac{u_{i+1,\,j}-u_{i-1,\,j}}{2\Delta x}-2\frac{u_{i,\,j+1}-u_{i,\,j-1}}{2\Delta y}\frac{v_{i+1,\,j}-v_{i-1,\,j}}{2\Delta x}-\frac{v_{i,\,j+1}-v_{i,\,j-1}}{2\Delta y}\frac{v_{i,\,j+1}-v_{i,\,j-1}}{2\Delta y}\right]$$

The initial condition is $u,\ v,\ p=0$ everywhere, and at the boundary conditions are:

$u,\ v,\ p$ are periodic on $x=0,\ 2$

$u,\ v=0$ at $y=0,\ 2$

$\frac{\partial p}{\partial y}=0$ at $y=0,\ 2$

$F=1$ everywhere.

One thing to note is that we have periodic boundary conditions throughout this grid, so we need to explicitly calculate the values at the leading and trailing edge of our u vector.

```
1.  import numpy
2.  from matplotlib import pyplot, cm
3.  from mpl_toolkits.mplot3d import Axes3D
4.
5.
```

```
def build_up_b(rho, dt, dx, dy, u, v):
    b = numpy.zeros_like(u)
    b[1:-1, 1:-1] = (rho * (1 / dt * ((u[1:-1, 2:] - u[1:-1, 0:-2]) / (2 * dx) +
                                      (v[2:, 1:-1] - v[0:-2, 1:-1]) / (2 * dy)) -
                            ((u[1:-1, 2:] - u[1:-1, 0:-2]) / (2 * dx))**2 -
                            2 * ((u[2:, 1:-1] - u[0:-2, 1:-1]) / (2 * dy) *
                                 (v[1:-1, 2:] - v[1:-1, 0:-2]) / (2 * dx))-
                            ((v[2:, 1:-1] - v[0:-2, 1:-1]) / (2 * dy))**2))

    # Periodic BC Pressure @ x = 2
    b[1:-1, -1] = (rho * (1 / dt * ((u[1:-1, 0] - u[1:-1,-2]) / (2 * dx) +
                                    (v[2:, -1] - v[0:-2, -1]) / (2 * dy)) -
                          ((u[1:-1, 0] - u[1:-1, -2]) / (2 * dx))**2 -
                          2 * ((u[2:, -1] - u[0:-2, -1]) / (2 * dy) *
                               (v[1:-1, 0] - v[1:-1, -2]) / (2 * dx)) -
                          ((v[2:, -1] - v[0:-2, -1]) / (2 * dy))**2))

    # Periodic BC Pressure @ x = 0
    b[1:-1, 0] = (rho * (1 / dt * ((u[1:-1, 1] - u[1:-1, -1]) / (2 * dx) +
                                   (v[2:, 0] - v[0:-2, 0]) / (2 * dy)) -
                         ((u[1:-1, 1] - u[1:-1, -1]) / (2 * dx))**2 -
                         2 * ((u[2:, 0] - u[0:-2, 0]) / (2 * dy) *
                              (v[1:-1, 1] - v[1:-1, -1]) / (2 * dx))-
                         ((v[2:, 0] - v[0:-2, 0]) / (2 * dy))**2))

    return b
```

We'll also define a Pressure Poisson iterative function,Once more, note that we have to include the periodic boundary conditions at the leading and trailing edge. We also have to specify the boundary conditions at the top and bottom of our grid.

```
def pressure_poisson_periodic(p, dx, dy):
    pn = numpy.empty_like(p)

    for q in range(nit):
        pn = p.copy()
        p[1:-1, 1:-1] = (((pn[1:-1, 2:] + pn[1:-1, 0:-2]) * dy**2 +
                          (pn[2:, 1:-1] + pn[0:-2, 1:-1]) * dx**2) /
                         (2 * (dx**2 + dy**2)) -
                         dx**2 * dy**2 / (2 * (dx**2 + dy**2)) * b[1:-1, 1:-1])

        # Periodic BC Pressure @ x = 2
        p[1:-1, -1] = (((pn[1:-1, 0] + pn[1:-1, -2])* dy**2 +
                        (pn[2:, -1] + pn[0:-2, -1]) * dx**2) /
```

```
                                (2 * (dx**2 + dy**2)) -
                                dx**2 * dy**2 / (2 * (dx**2 + dy**2)) * b[1:-1, -1])

        # Periodic BC Pressure @ x = 0
        p[1:-1, 0] = (((pn[1:-1, 1] + pn[1:-1, -1])* dy**2 +
                        (pn[2:, 0] + pn[0:-2, 0]) * dx**2) /
                       (2 * (dx**2 + dy**2)) -
                       dx**2 * dy**2 / (2 * (dx**2 + dy**2)) * b[1:-1, 0])

        # Wall boundary conditions, pressure
        p[-1, :] =p[-2, :]    # dp/dy = 0 at y = 2
        p[0, :] = p[1, :]    # dp/dy = 0 at y = 0

    return p
```

Now we have our familiar list of variables and initial conditions to declare before we start.

```
##variable declarations
nx = 41
ny = 41
nt = 10
nit = 50
c = 1
dx = 2 / (nx - 1)
dy = 2 / (ny - 1)
x = numpy.linspace(0, 2, nx)
y = numpy.linspace(0, 2, ny)
X, Y = numpy.meshgrid(x, y)


##physical variables
rho = 1
nu = .1
F = 1
dt = .01

#initial conditions
u = numpy.zeros((ny, nx))
un = numpy.zeros((ny, nx))

v = numpy.zeros((ny, nx))
vn = numpy.zeros((ny, nx))

p = numpy.ones((ny, nx))
pn = numpy.ones((ny, nx))
```

```
87.
88.  b = numpy.zeros((ny, nx))
```

We can either specify a number of timesteps nt and increment it until we're satisfied with the results, or we can tell our code to run until the difference between two consecutive iterations is very small.

We also have to manage 8 separate boundary conditions for each iteration.

```
89.  udiff = 1
90.  stepcount = 0
91.
92.  while udiff > .001:
93.      un = u.copy()
94.      vn = v.copy()
95.
96.      b = build_up_b(rho, dt, dx, dy, u, v)
97.      p = pressure_poisson_periodic(p, dx, dy)
98.
99.      u[1:-1, 1:-1] = (un[1:-1, 1:-1] -
100.                       un[1:-1, 1:-1] * dt / dx *
101.                      (un[1:-1, 1:-1] - un[1:-1, 0:-2]) -
102.                       vn[1:-1, 1:-1] * dt / dy *
103.                      (un[1:-1, 1:-1] - un[0:-2, 1:-1]) -
104.                       dt / (2 * rho * dx) *
105.                      (p[1:-1, 2:] - p[1:-1, 0:-2]) +
106.                       nu * (dt / dx**2 *
107.                      (un[1:-1, 2:] - 2 * un[1:-1, 1:-1] + un[1:-1, 0:-2]) +
108.                       dt / dy**2 *
109.                      (un[2:, 1:-1] - 2 * un[1:-1, 1:-1] + un[0:-2, 1:-1])) +
110.                       F * dt)
111.
112.     v[1:-1, 1:-1] = (vn[1:-1, 1:-1] -
113.                       un[1:-1, 1:-1] * dt / dx *
114.                      (vn[1:-1, 1:-1] - vn[1:-1, 0:-2]) -
115.                       vn[1:-1, 1:-1] * dt / dy *
116.                      (vn[1:-1, 1:-1] - vn[0:-2, 1:-1]) -
117.                       dt / (2 * rho * dy) *
118.                      (p[2:, 1:-1] - p[0:-2, 1:-1]) +
119.                       nu * (dt / dx**2 *
120.                      (vn[1:-1, 2:] - 2 * vn[1:-1, 1:-1] + vn[1:-1, 0:-2]) +
121.                       dt / dy**2 *
122.                      (vn[2:, 1:-1] - 2 * vn[1:-1, 1:-1] + vn[0:-2, 1:-1])))
123.
124.     # Periodic BC u @ x = 2
125.     u[1:-1, -1] = (un[1:-1, -1] - un[1:-1, -1] * dt / dx *
```

```
                    (un[1:-1, -1] - un[1:-1, -2]) -
                     vn[1:-1, -1] * dt / dy *
                    (un[1:-1, -1] - un[0:-2, -1]) -
                     dt / (2 * rho * dx) *
                    (p[1:-1, 0] - p[1:-1, -2]) +
                     nu * (dt / dx**2 *
                    (un[1:-1, 0] - 2 * un[1:-1,-1] + un[1:-1, -2]) +
                     dt / dy**2 *
                    (un[2:, -1] - 2 * un[1:-1, -1] + un[0:-2, -1])) + F * dt)

    # Periodic BC u @ x = 0
    u[1:-1, 0] = (un[1:-1, 0] - un[1:-1, 0] * dt / dx *
                    (un[1:-1, 0] - un[1:-1, -1]) -
                     vn[1:-1, 0] * dt / dy *
                    (un[1:-1, 0] - un[0:-2, 0]) -
                     dt / (2 * rho * dx) *
                    (p[1:-1, 1] - p[1:-1, -1]) +
                     nu * (dt / dx**2 *
                    (un[1:-1, 1] - 2 * un[1:-1, 0] + un[1:-1, -1]) +
                     dt / dy**2 *
                    (un[2:, 0] - 2 * un[1:-1, 0] + un[0:-2, 0])) + F * dt)

    # Periodic BC v @ x = 2
    v[1:-1, -1] = (vn[1:-1, -1] - un[1:-1, -1] * dt / dx *
                    (vn[1:-1, -1] - vn[1:-1, -2]) -
                     vn[1:-1, -1] * dt / dy *
                    (vn[1:-1, -1] - vn[0:-2, -1]) -
                     dt / (2 * rho * dy) *
                    (p[2:, -1] - p[0:-2, -1]) +
                     nu * (dt / dx**2 *
                    (vn[1:-1, 0] - 2 * vn[1:-1, -1] + vn[1:-1, -2]) +
                     dt / dy**2 *
                    (vn[2:, -1] - 2 * vn[1:-1, -1] + vn[0:-2, -1])))

    # Periodic BC v @ x = 0
    v[1:-1, 0] = (vn[1:-1, 0] - un[1:-1, 0] * dt / dx *
                    (vn[1:-1, 0] - vn[1:-1, -1]) -
                     vn[1:-1, 0] * dt / dy *
                    (vn[1:-1, 0] - vn[0:-2, 0]) -
                     dt / (2 * rho * dy) *
                    (p[2:, 0] - p[0:-2, 0]) +
                     nu * (dt / dx**2 *
                    (vn[1:-1, 1] - 2 * vn[1:-1, 0] + vn[1:-1, -1]) +
                     dt / dy**2 *
                    (vn[2:, 0] - 2 * vn[1:-1, 0] + vn[0:-2, 0])))
```

```
171.
172.
173.        # Wall BC: u,v = 0 @ y = 0,2
174.        u[0, :] = 0
175.        u[-1, :] = 0
176.        v[0, :] = 0
177.        v[-1, :]=0
178.
179.        udiff = (numpy.sum(u) - numpy.sum(un)) / numpy.sum(u)
180.        stepcount += 1
```

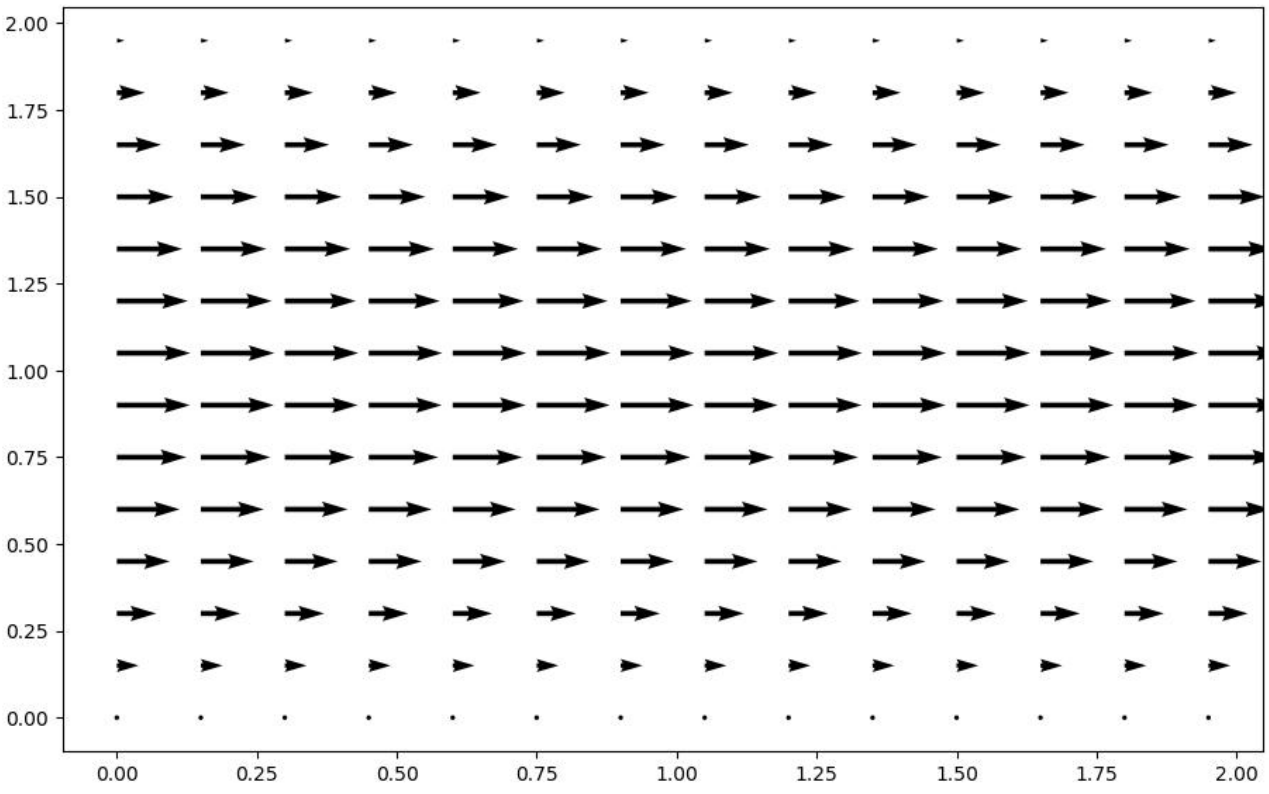
You can see that we've also included a variable stepcount to see how many iterations our loop went through before our stop condition was met.

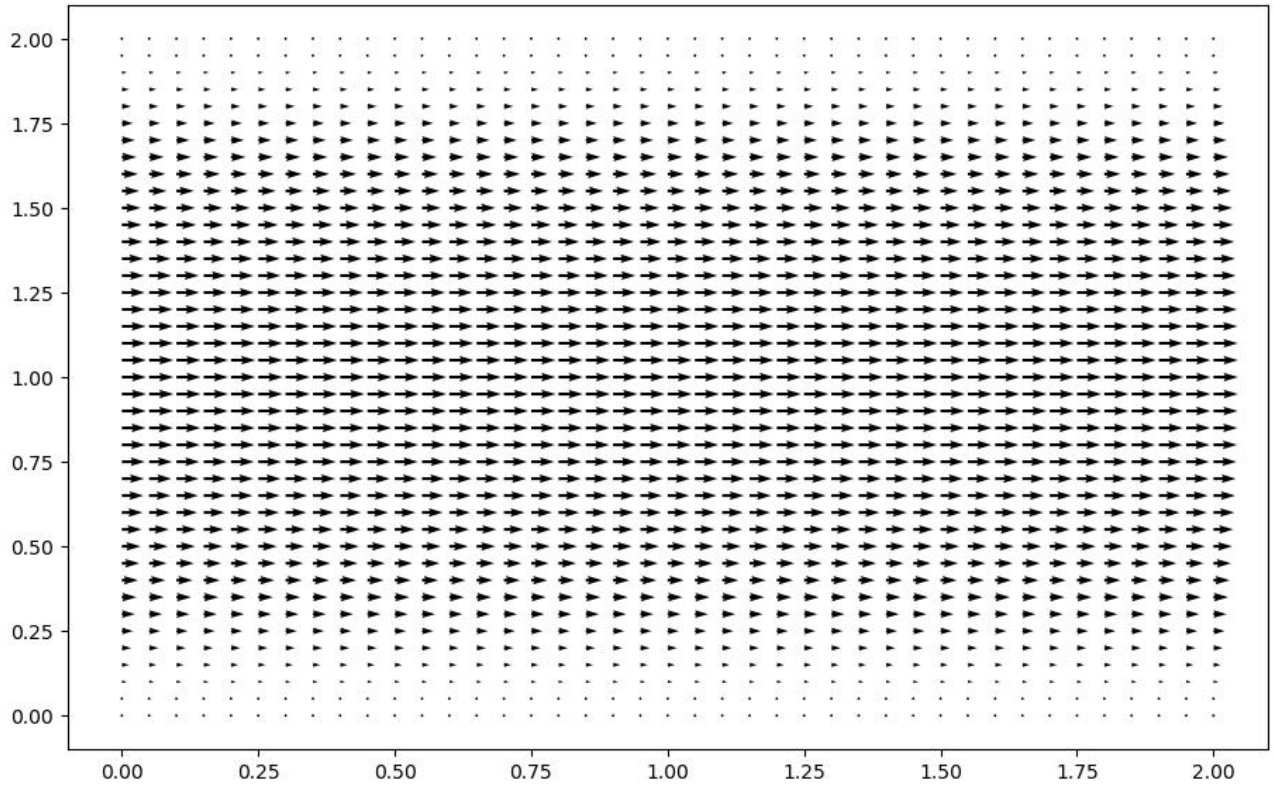
```
181. print(stepcount)
182. 499
```

If you want to see how the number of iterations increases as our udiff condition gets smaller and smaller, try defining a function to perform the while loop written above that takes an input udiff and outputs the number of iterations that the function runs.

```
183. fig = pyplot.figure(figsize = (11,7), dpi=100)
184. pyplot.quiver(X[::3, ::3], Y[::3, ::3], u[::3, ::3], v[::3, ::3]);
185.
186. fig = pyplot.figure(figsize = (11,7), dpi=100)
187. pyplot.quiver(X, Y, u, v);
```

The structures in the quiver command that look like [::3, ::3] are useful when dealing with large amounts of data that you want to visualize. The one used above tells matplotlib to only plot every 3rd data point. If we leave it out, you can see that the results can appear a little crowded.





Wave Equation

To solve the 2-dimensional wave equation,

$$\frac{\partial^2 u}{\partial t^2} = D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

using:

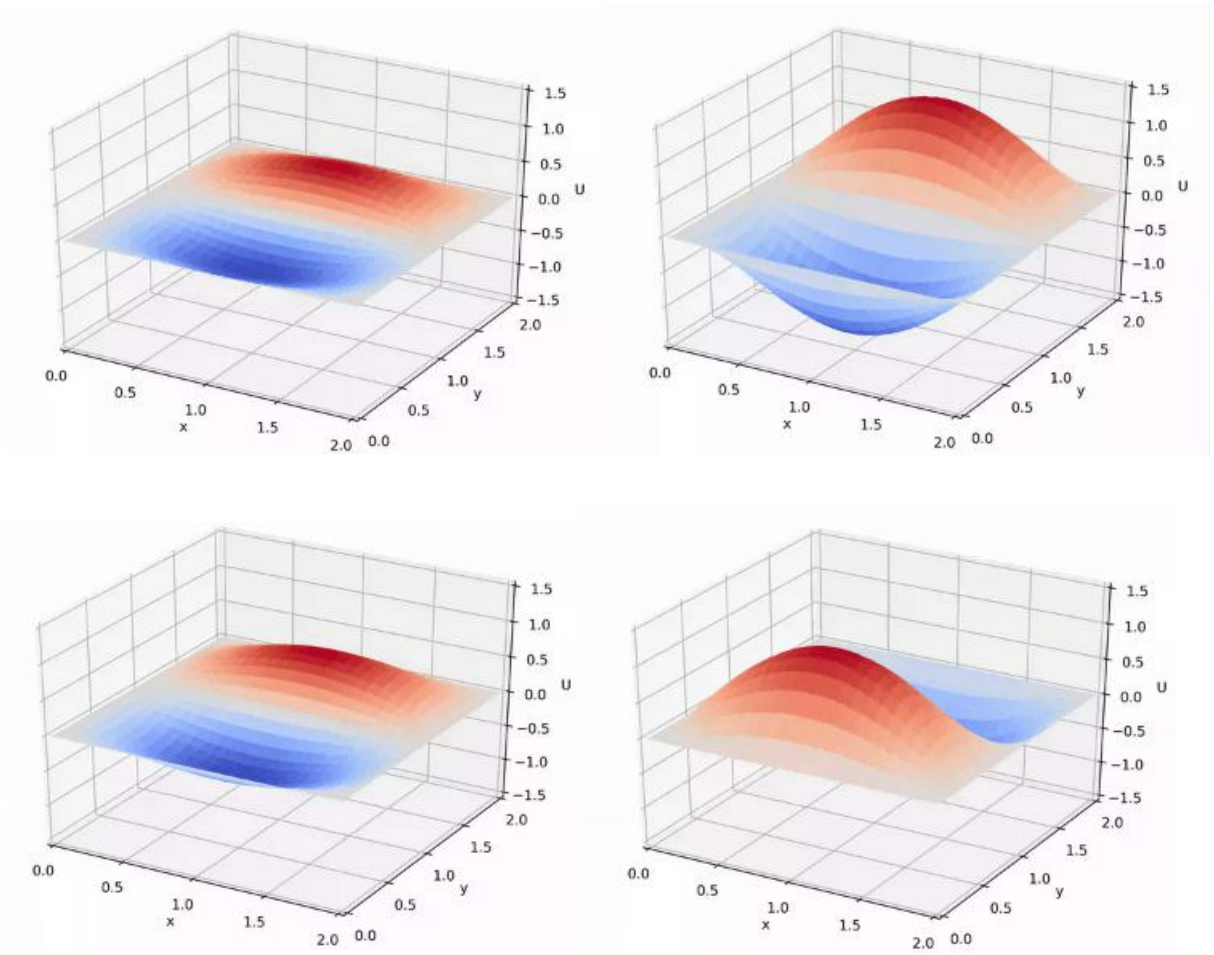
The finite difference method, by applying the three-point central difference approximation for the time and space discretization. This way of approximation leads to an explicit central difference method, where it requires

$$r = \frac{4D\Delta t^2}{\Delta x^2 + \Delta y^2} < 1$$

to guarantee stability. The following example is a solution of the wave equation on a [0,2]×[0,2][0,2]×[0,2] domain, with diffusion coefficient D=0.25, initial condition u(x,y,0)=0.1sin(πx)sin(πy/2), initial velocity ∂u(x,y,0)/∂t=0, and Dirichlet boundary condition u(0,y,t)=u(2,y,t)=u(x,0,t)=u(x,2,t)=0

For the requirement of r<1r<1, we use Python assert statement, so that the program will not execute and raise an error if the requirement is not fulfilled. In the code below, the assertion is applied in

the initialization function.



Code

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.  from mpl_toolkits.mplot3d import Axes3D
4.  from matplotlib import cm
5.
6.
7.  class WaveEquationFD:
8.
9.      def __init__(self, N, D, Mx, My):
10.         self.N = N
11.         self.D = D
12.         self.Mx = Mx
13.         self.My = My
14.         self.tend = 6
15.         self.xmin = 0
16.         self.xmax = 2
17.         self.ymin = 0
```

```
        self.ymax = 2
        self.initialization()
        self.eqnApprox()


    def initialization(self):
        self.dx = (self.xmax - self.xmin)/self.Mx
        self.dy = (self.ymax - self.ymin)/self.My

        self.x = np.arange(self.xmin, self.xmax+self.dx, self.dx)
        self.y = np.arange(self.ymin, self.ymax+self.dy, self.dy)

        #----- Initial condition -----#
        self.u0 = lambda r, s: 0.1*np.sin(np.pi*r)*np.sin(np.pi*s/2)

        #----- Initial velocity -----#
        self.v0 = lambda a, b: 0

        #----- Boundary conditions -----#
        self.bxyt = lambda left, right, time: 0

        self.dt = (self.tend - 0)/self.N
        self.t = np.arange(0, self.tend+self.dt/2, self.dt)

        # Assertion for the condition of r < 1, for stability
        r = 4*self.D*self.dt**2/(self.dx**2+self.dy**2);
        assert r < 1, "r is bigger than 1!"


    def eqnApprox(self):
        #----- Approximation equation properties -----#
        self.rx = self.D*self.dt**2/self.dx**2
        self.ry = self.D*self.dt**2/self.dy**2
        self.rxy1 = 1 - self.rx - self.ry
        self.rxy2 = self.rxy1*2

        #----- Initialization matrix u for solution -----#
        self.u = np.zeros((self.Mx+1, self.My+1))
        self.ut = np.zeros((self.Mx+1, self.My+1))
        self.u_1 = self.u.copy()

        #----- Fills initial condition and initial velocity -----#
        for j in range(1, self.Mx):
            for i in range(1, self.My):
                self.u[i,j] = self.u0(self.x[i], self.y[j])
```

```
63.                 self.ut[i,j] = self.v0(self.x[i], self.y[j])
64.
65.
66.     def solve_and_animate(self):
67.
68.         u_2 = np.zeros((self.Mx+1, self.My+1))
69.
70.         xx, yy = np.meshgrid(self.x, self.y)
71.
72.         fig = plt.figure()
73.         ax = fig.add_subplot(111, projection='3d')
74.
75.         wframe = None
76.
77.         k = 0
78.         nsteps = self.N
79.
80.         while k < nsteps:
81.             if wframe:
82.                 ax.collections.remove(wframe)
83.
84.             self.t = k*self.dt
85.
86.             #----- Fills in boundary condition along y-axis (vertical,
   columns 0 and Mx) -----#
87.             for i in range(self.My+1):
88.                 self.u[i, 0] = self.bxyt(self.x[0], self.y[i], self.t)
89.                 self.u[i,   self.Mx]   =   self.bxyt(self.x[self.Mx],
   self.y[i], self.t)
90.
91.             for j in range(self.Mx+1):
92.                 self.u[0, j] = self.bxyt(self.x[j], self.y[0], self.t)
93.                 self.u[self.My,     j]     =     self.bxyt(self.x[j],
   self.y[self.My], self.t)
94.
95.             if k == 0:
96.                 for j in range(1, self.My):
97.                     for i in range(1, self.Mx):
98.                         self.u[i,j] = 0.5*(self.rx*(self.u_1[i-1,j] +
   self.u_1[i+1,j])) \
99.                                 +    0.5*(self.ry*(self.u_1[i,j-1]    +
   self.u_1[i,j+1])) \
100.                                +        self.rxy1*self.u[i,j]        +
   self.dt*self.ut[i,j]
101.            else:
```

```
102.                for j in range(1, self.My):
103.                    for i in range(1, self.Mx):
104.                        self.u[i,j]   =   self.rx*(self.u_1[i-1,j]   +
   self.u_1[i+1,j]) \
105.                            +        self.ry*(self.u_1[i,j-1]        +
   self.u_1[i,j+1]) \
106.                            + self.rxy2*self.u[i,j] - u_2[i,j]
107.
108.            u_2 = self.u_1.copy()
109.            self.u_1 = self.u.copy()
110.
111.            wframe = ax.plot_surface(xx, yy, self.u, cmap=cm.coolwarm,
   linewidth=2,
112.                    antialiased=False)
113.
114.            ax.set_xlim3d(0, 2.0)
115.            ax.set_ylim3d(0, 2.0)
116.            ax.set_zlim3d(-1.5, 1.5)
117.
118.            ax.set_xticks([0, 0.5, 1.0, 1.5, 2.0])
119.            ax.set_yticks([0, 0.5, 1.0, 1.5, 2.0])
120.
121.            ax.set_xlabel("x")
122.            ax.set_ylabel("y")
123.            ax.set_zlabel("U")
124.
125.            plt.pause(0.01)
126.            k += 0.5
127.
128.
129.def main():
130.    simulator = WaveEquationFD(200, 0.25, 50, 50)
131.    simulator.solve_and_animate()
132.    plt.show()
133.
134.if __name__ == "__main__":
135.    main()
136.
137.
138.#N = 200
139.#D = 0.25
140.#Mx = 50
141.#My = 50
```

Ocean Wave Mechanics

Shoaling

Question1: Shoaling

Consider one-dimensional wave propagation with:

Variables:
1. Wave height *H(x)*
2. Wave period *T*
3. It in the x-direction over a uniform sloping seabed
Water depth is described by ***h=100.0-x(m)***.

Boundary conditions of wave:
H=2.0 m and T=6.0 s at the offshore end (x=0)

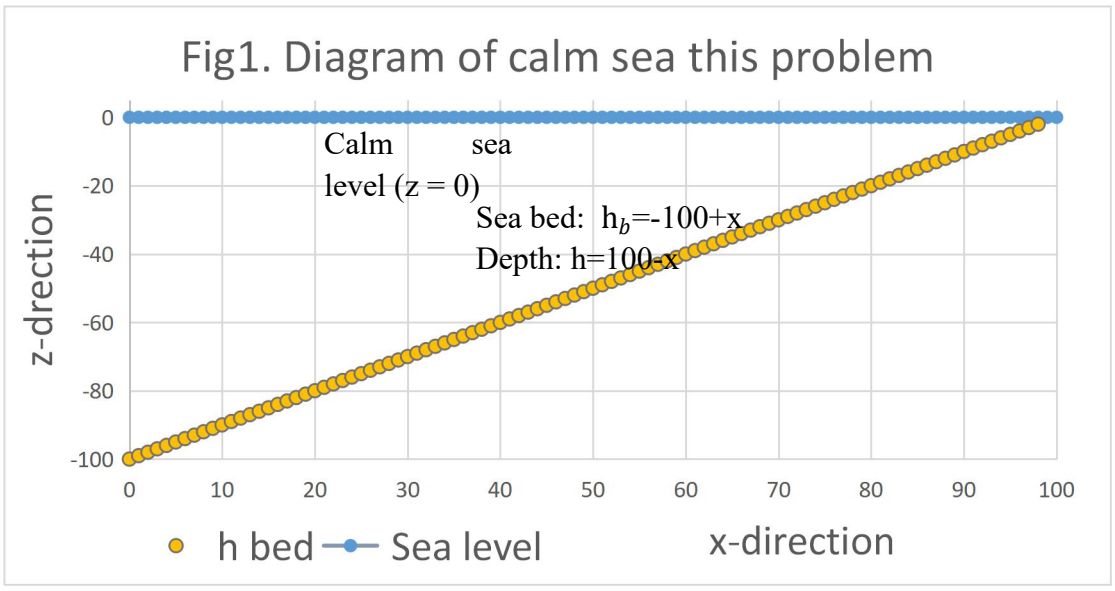
Target:
Calculate the cross-shore distributions of
1. the wave height
2. wave velocity
3. The maximum horizontal fluid velocity
In the range ($0 \leq x \leq 99m$).

Answer:

I started learn the theory of Wave Mechanics in this lecture, so in my opinions i must derive the function by some fundamental theory, it might be verbose.

This is not result of this quesion, but i try to obtain a function of wave, take it in the range fig1 below.

I did not really understand what is **velocity potential**. So i did learn about them, my answer start from it. I write down them just because i must start my understanding of this question from them. **Solution of question1 start from par7 in page10.**



Part 1: potential flow, velocity potential.

1. Potential flow describes the velocity field as **the gradient of a sclar function**.
2. A potential flow is characterized by **an irrotational velocity field**.

Then in potential flow model:

$$V = \nabla\varphi$$

Where V is a velocity field $[u,v,w]$, ∇ means gradient.

$$\nabla = \left[\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right]$$

Therefore:

$$\nabla \times V = \nabla \times \nabla\varphi$$

Here curl of a gradient is zero.

$$\nabla \times V = 0$$

Substitute $V = \nabla\varphi$ into this equation:

$$\nabla \times \nabla\varphi = 0$$

Then here obtain the Laplace Equation:

$$\nabla^2\varphi = 0 \qquad (\mathbf{1})$$

It is also the continuity equation for potential flow, discussion of velocity potential must in potential flow. This model be used here should be a non-visvous (so it is irrotational), and also obviously incompressible.

Use the Euler equation describe the flow this condition:

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u}\cdot\boldsymbol{\nabla})\boldsymbol{u} = -\frac{1}{\rho}\boldsymbol{\nabla}p + \boldsymbol{f} \qquad (\mathbf{2})$$

Where red part:

$$u \times (\boldsymbol{\nabla} \times u) = \boldsymbol{\nabla}\left(\frac{|\boldsymbol{u}|^2}{2}\right) - (\boldsymbol{u}\cdot\boldsymbol{\nabla})\boldsymbol{u}$$

$$(\boldsymbol{u}\cdot\boldsymbol{\nabla})\boldsymbol{u} = \boldsymbol{\nabla}\left(\frac{|\boldsymbol{u}|^2}{2}\right) - u \times (\boldsymbol{\nabla} \times u)$$

Substitute it into (1), where let $\boldsymbol{\nabla} \times u = \omega$ (vorticity).

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{\nabla}\left(\frac{|\boldsymbol{u}|^2}{2}\right) - u \times \omega = -\frac{1}{\rho}\boldsymbol{\nabla}p + \boldsymbol{f}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} - u \times \omega = -\boldsymbol{\nabla}\left(\frac{|\boldsymbol{u}|^2}{2}\right) - \frac{1}{\rho}\boldsymbol{\nabla}p + \boldsymbol{f} \qquad (\mathbf{3})$$

Where ρ should be a constant, and the external force $(\boldsymbol{f})$, it is a conservative force (usually from gravity) , having a potential χ as:

$$\boldsymbol{f} = -\boldsymbol{\nabla}\chi$$

Therefore (2) be rewritten as:

$$\frac{\partial \boldsymbol{u}}{\partial t} - u \times \omega = -\nabla\left(\frac{|\boldsymbol{u}|^2}{2}\right) - \frac{1}{\rho}\nabla p - \nabla\chi$$

$$\frac{\partial \boldsymbol{u}}{\partial t} - u \times \omega = -\nabla\left(\frac{|\boldsymbol{u}|^2}{2} + \frac{p}{\rho} + \chi\right) \quad (4)$$

There assumed an irrotational flow field, vorticity be zero: $\nabla \times u = \omega = 0$

$$\frac{\partial \boldsymbol{u}}{\partial t} = -\nabla\left(\frac{|\boldsymbol{u}|^2}{2} + \frac{p}{\rho} + \chi\right) \quad (5)$$

In irrotational flow field, the velocity vector can be expressed as a gradient of a scalar function, usually call it velocity potential and expressed as follows:

$$\boldsymbol{u} = \mathrm{grad}(\varphi) = \nabla\varphi \quad (6)$$

Where φ is a scalar called the velocuty potential.

Now substitute (5) into (4), and because of the conservative force this case from gravitational force:

$$\chi = \mathrm{g}z$$

Therefore:

$$\nabla\frac{\partial \varphi}{\partial t} + \nabla\left(\frac{|\boldsymbol{u}|^2}{2} + \frac{p}{\rho} + \mathrm{g}z\right) = 0$$

$$\nabla\left(\frac{\partial \varphi}{\partial t} + \frac{|\boldsymbol{u}|^2}{2} + \frac{p}{\rho} + \mathrm{g}z\right) = 0 \quad (7)$$

Then integrating (7) over space obtain:

$$\frac{\partial \varphi}{\partial t} + \frac{|\boldsymbol{u}|^2}{2} + \frac{p}{\rho} + \mathrm{g}z = \mathrm{C}(t) \quad (8)$$

Part 2: Linear Wave Theory

There are some common assumptions in wave theories:
1. The waves have regular profiles.
2. The flow is 2-D (in vertical x, z plane).
3. The wave propagation is unidirectional.
4. The fluid is ideal (inviscid, incompressible, irrotational).

First considering a Linear Wave Theory can be applied in Question1:

Water surface where free surface $\mathrm{p} = 0$, and assume $\mathrm{z} = \eta$ at the water surface , then from eq.(8):

(9)

$$\frac{\partial \varphi}{\partial t}+\frac{1}{2}\left(u^2+w^2\right)+g\eta=C(t)$$

There eq.(9) is a constraint be called the **dynamic boundary condition** at water surface.
And there if neglecting the mixing between air and water there is:

$$\frac{\partial \eta}{\partial t}+u\frac{\partial \eta}{\partial x}-w=0 \qquad (10)$$

It is derived from the streamlines equation, and it is be called the **kinematic boundary condition** at the water surface.

For equation (9) and (10), expressed by velocity potential:

$$u=\frac{\partial \varphi}{\partial x}\ ,\quad w=\frac{\partial \varphi}{\partial z} \qquad (11)$$

Therefore , (9) and (10) can be rewritten as:

$$\text{DFSBC: } \frac{\partial \varphi}{\partial t}+\frac{1}{2}\left[\left(\frac{\partial \varphi}{\partial x}\right)^2+\left(\frac{\partial \varphi}{\partial z}\right)^2\right]+g\eta=0\ \ (z=\eta) \qquad (12)$$

$$\text{KFSBC: } \frac{\partial \varphi}{\partial z}=\frac{\partial \eta}{\partial t}+\frac{\partial \varphi}{\partial x}\frac{\partial \eta}{\partial x}\ \ (z=\eta) \qquad (13)$$

These two boundary conditions are derived at surface.

The other condition, where bottom (z $=-$ h) , the bottom boundary condition should be:

$$\text{BBC：}\ \frac{\partial \varphi}{\partial z}+\frac{\partial \varphi}{\partial x}\frac{\partial h}{\partial x}=0\ \ (z=-h) \qquad (14)$$

In linear wave theory, it is assumed that the velocity potential (ϕ) depends on position and time and this is given by：

$$\phi(x,y,z)=X(x)Z(z)T(t) \qquad (15)$$

Where X, Z, T are initially unknown functions of x, z and t respectively, assumed to be independent of each other.

These unknown functions can be determined by making ϕ to satisfy

i. Laplace Equation (1).

ii. Linearised form of Bernoulli's Dynamic Equation at the free surface from (8), let z=η, C=0:

$$\frac{\partial \varphi}{\partial t}+\frac{1}{2}\left[\left(\frac{\partial \varphi}{\partial x}\right)^2+\left(\frac{\partial \varphi}{\partial z}\right)^2\right]+\frac{p}{\rho}+g\eta=0 \qquad (16)$$

Then, ignore $\left(\frac{\partial \varphi}{\partial x}\right)^2$, $\left(\frac{\partial \varphi}{\partial z}\right)^2$:

$$\frac{\partial \varphi}{\partial t} + \frac{p}{\rho} + g\eta = 0 \quad (\mathbf{17})$$

iii. Dynamic Free Surface Boundary Condition (12).
iv. Kinematic Free Surface Boundary Condition (13).
v. Bottom Boundary Condition (14).

From here i, ii, iii, iv, v, the expression for ϕ determined is:

$$\phi = \frac{gH}{2\sigma} \frac{\cosh(k(h+z))}{\cosh(kh)} \sin(kx - \sigma t) \quad (\mathbf{18})$$

Where:
H is wave height;
σ is Circular wave frequency equals to $2\pi/T$;
T is Wave period;
k is Wave number equals to $2\pi/L$;
L is Wave length;
z is Vertical co-ordinate of the point at which ϕ is being considered;
x is Horizontal co-ordinate of the point.
h is Water depth;
t is Time instant.

Part 3: How obtain (18):

Start from Bernoulli's Dynamic Equation:

$$\frac{\partial \varphi}{\partial t} + \frac{1}{2}\left[\left(\frac{\partial \varphi}{\partial x}\right)^2 + \left(\frac{\partial \varphi}{\partial z}\right)^2\right] + \frac{p}{\rho} + gz = C(t)$$

In linear theory we assume that the partial derivatives are small and so their product here $\left(\frac{\partial \varphi}{\partial x}\right)^2$ and $\left(\frac{\partial \varphi}{\partial z}\right)^2$ are negligible compared to other terms, and consider a point on the free surface, that point z = η;

$$\frac{\partial \varphi}{\partial t} + \frac{p}{\rho} + g\eta = 0$$

with DFSBC there p = 0, hence:

$$\frac{\partial \varphi}{\partial t} + g\eta = 0$$

$$\eta = -\frac{1}{g}\left(\frac{\partial \varphi}{\partial t}\right) \ (z = \eta)$$

And assuming the wave height (H), it is very small compared to wave length(L), so it obatin a approximated as:

$$\eta = -\frac{1}{g}\left(\frac{\partial \varphi}{\partial t}\right) \ (z = 0) \quad (\mathbf{19})$$

Back to KFSBC(13):

$$\frac{\partial \varphi}{\partial z}=\frac{\partial \eta}{\partial t}+\frac{\partial \varphi}{\partial x}\frac{\partial \eta}{\partial x}$$

And here product of two differentials can be neglected too, it reduced:

$$\frac{\partial \eta}{\partial t}\approx\frac{\partial \varphi}{\partial z}\quad (z=0=\eta) \tag{20}$$

From (19):

$$\frac{\partial \eta}{\partial t}=-\frac{1}{g}\left(\frac{\partial^2 \varphi}{\partial t^2}\right)$$

Substitute it into (20):

$$\frac{\partial \varphi}{\partial z}+\frac{1}{g}\left(\frac{\partial^2 \varphi}{\partial t^2}\right)=0$$

$$\frac{\partial^2 \varphi}{\partial t^2}+g\frac{\partial \varphi}{\partial z}=0 \tag{21}$$

This has solution of (15), from:

$$\phi(x,y,z)=X(x)Z(z)T(t)$$

(1) Laplace Equation in x-z plane:

$$\frac{\partial^2 \varphi}{\partial x^2}+\frac{\partial^2 \varphi}{\partial z^2}=0$$

Therefore (15) become:

$$\frac{\partial^2 \varphi}{\partial x^2}[Z(z)T(t)]+\frac{\partial^2 \varphi}{\partial z^2}[X(x)T(t)]=0$$

And divide it by (X Z T),so:

$$\frac{\partial^2 \varphi}{\partial x^2}\left[\frac{1}{X(x)}\right]+\frac{\partial^2 \varphi}{\partial z^2}\left[\frac{1}{Z(z)}\right]=0 \tag{22}$$

Here let first term of (22):

$$\frac{\partial^2 \varphi}{\partial x^2}\left[\frac{1}{X(x)}\right]=-k^2$$

And second term of (22) become:

$$\frac{\partial^2 \varphi}{\partial z^2}\left[\frac{1}{Z(z)}\right]=k^2$$

Therefore, here obtained:

$$\begin{cases}\dfrac{\partial^2 \varphi}{\partial x^2}+k^2X(x)=0\\[2ex] \dfrac{\partial^2 \varphi}{\partial z^2}-k^2Z(z)=0\end{cases} \tag{23}$$

There have standard solutions:

$$\begin{cases}X(x)=c_1\cos(kx)+c_2\cos(kx)\\ Z(z)=c_3e^{kz}+c_4e^{-kz}\end{cases} \tag{24}$$

And about T(t), object for ϕ is harmonic in time period(T), so write it as:

$$T(t)=\cos\left(\frac{2\pi}{T}\right)=\cos(\sigma t)\ \ or\ T(t)=\sin\left(\frac{2\pi}{T}\right)=\sin(\sigma t) \tag{25}$$

Therefore,from (22) (23) (24) and (25) here obtain the ϕ:

(26)

$$\phi = [c_1 \cos(kx) + c_2 \cos(kx)]\left(c_3 e^{kz} + c_4 e^{-kz}\right) \cos(\sigma t)$$

And

$$\phi = [c_1 \cos(kx) + c_2 \cos(kx)]\left(c_3 e^{kz} + c_4 e^{-kz}\right) \sin(\sigma t) \quad (\mathbf{27})$$

According to the property of the Laplace equation :

If solution of it, $\phi_{general} = \phi_a + \phi_b$, each of ϕ_a and ϕ_b, are particular solutions, them satisfying the Laplace equation too. So from (26), (27), here can form 4 particular solutions:

$$\begin{cases} \phi_1 = c_1 \cos(kx)\left(c_3 e^{kz} + c_4 e^{-kz}\right) \cos(\sigma t) \\ \phi_2 = c_1 \cos(kx)\left(c_3 e^{kz} + c_4 e^{-kz}\right) \sin(\sigma t) \\ \phi_3 = c_2 \cos(kx)\left(c_3 e^{kz} + c_4 e^{-kz}\right) \cos(\sigma t) \\ \phi_4 = c_2 \cos(kx)\left(c_3 e^{kz} + c_4 e^{-kz}\right) \sin(\sigma t) \end{cases} \quad (\mathbf{28})$$

Here just focus on the solution ϕ_1, apply BBC(14) on it:

$$\frac{\partial \phi_1}{\partial z} + \frac{\partial \phi_1}{\partial x}\frac{\partial h}{\partial x} = 0$$

Ignoring product of here two partial difference term:

$$\frac{\partial \phi_1}{\partial z} = 0 \ \ (z = -h)$$

$$\frac{\partial}{\partial z}\left[c_1 \cos(kx)\left(c_3 e^{kz} + c_4 e^{-kz}\right) \cos(\sigma t)\right] = 0$$

$$c_1 \cos(kx) \cos(\sigma t)\frac{\partial}{\partial z}\left(c_3 e^{kz} + c_4 e^{-kz}\right) = 0 \quad (\mathbf{29})$$

Here if let $c_3 e^{kz} = c_4 e^{-kz}$:

$$c_3 = c_4 e^{-2kz}$$

And at bottom (z = -h):

$$c_3 = c_4 e^{2kh}$$

Therefore from it , ϕ_1 for all x and t could be difined as:

$$\phi_1 = c_1 \cos(kx)\left(c_4 e^{2kh} e^{kz} + c_4 e^{-kz}\right) \cos(\sigma t)$$
$$\phi_1 = c_1 c_4 \cos(kx) \cos(\sigma t)\left(e^{2kh} e^{kz} + e^{-kz}\right)$$
$$\phi_1 = c_1 c_4 \cos(kx) \cos(\sigma t)\left(e^{kh} e^{kh} e^{kz} + e^{kh} e^{-kh} e^{-kz}\right)$$
$$\phi_1 = c_1 c_4 \cos(kx) \cos(\sigma t)\, e^{kh}\left(e^{kh} e^{kz} + e^{-kh} e^{-kz}\right)$$
$$\phi_1 = 2c_1 c_4 e^{kh}\frac{\left(e^{kh} e^{kz} + e^{-kh} e^{-kz}\right)}{2} \cos(kx) \cos(\sigma t)$$
$$\phi_1 = 2c_1 c_4 e^{kh}\left[\frac{\boldsymbol{e^{k(h+z)} + e^{(-k)(h+z)}}}{\mathbf{2}}\right] \cos(kx) \cos(\sigma t)$$

here use hyperbolic function to express it:

$$\phi_1 = 2c_1 c_4 e^{kh}\mathbf{cosh}\,[\boldsymbol{k(h+z)}] \cos(kx) \cos(\sigma t) \quad (\mathbf{30})$$

Then from (19):

$$\eta = -\frac{1}{g}\left(\frac{\partial \varphi}{\partial t}\right) \ (z = 0)$$

Here ϕ_1 was obtained in (30) , just substitute it into (19):

$$\eta = -\frac{1}{g}\frac{\partial}{\partial t}\{2c_1c_4e^{kh}\cosh[k(h+z)]\cos(kx)\cos(\sigma t)\}$$

$$\eta = -\frac{1}{g}\frac{\partial}{\partial t}\{2c_1c_4e^{kh}\cosh[k(h+z)]\cos(kx)\cos(\sigma t)\}$$

$$\eta = -\frac{1}{g}2c_1c_4e^{kh}\cosh[k(h+z)]\cos(kx)\frac{\partial\cos(\sigma t)}{\partial t}$$

$$\eta = -\frac{2c_1c_4}{g}e^{kh}\cosh[k(h+z)]\cos(kx)\frac{\partial\cos(\sigma t)}{\partial t}$$

$$\eta = -\frac{2c_1c_4}{g}e^{kh}\cosh[k(h+z)]\cos(kx)(-\sigma)\sin(\sigma t)$$

Rearranging it:

$$\eta = \frac{2c_1c_4\sigma}{g}e^{kh}\cosh[k(h+z)]\cos(kx)\sin(\sigma t) \qquad (31)$$

And from (31), Maximum of η should be at z = 0 while $\boldsymbol{\cos(kx)\sin(\sigma t)=1}$, from this amplitude can be defined. Therefore the Amplitude(A):

$$A = \frac{2c_1c_4\sigma}{g}e^{kh}\cosh(kh) \qquad (32)$$

$$\frac{Ag}{\sigma\cosh(kh)} = 2c_1c_4e^{kh}$$

Appling this result in (30):

$$\phi_1 = \frac{Ag}{\sigma\cosh(kh)}\cosh[k(h+z)]\cos(kx)\cos(\sigma t)$$

$$\phi_1 = \frac{Ag}{\sigma}\frac{\cosh[k(h+z)]}{\cosh(kh)}\cos(kx)\cos(\sigma t)$$

Also in (28):

$$\begin{cases} \phi_1 = \dfrac{Ag}{\sigma}\dfrac{\cosh[k(h+z)]}{\cosh(kh)}\cos(kx)\cos(\sigma t) \\ \phi_2 = -\dfrac{Ag}{\sigma}\dfrac{\cosh[k(h+z)]}{\cosh(kh)}\cos(kx)\sin(\sigma t) \\ \phi_3 = \dfrac{Ag}{\sigma}\dfrac{\cosh[k(h+z)]}{\cosh(kh)}\sin(kx)\cos(\sigma t) \\ \phi_4 = -\dfrac{Ag}{\sigma}\dfrac{\cosh[k(h+z)]}{\cosh(kh)}\sin(kx)\sin(\sigma t) \end{cases} \qquad (33)$$

Each of them is a solution of the Laplace qeuation.

Here for getting a general solution of the Laplace equation, combine 2 solutions linearly.
One solution from $(\phi_1 - \phi_4)$:

$$\phi = \frac{Ag}{\sigma}\frac{\cosh[k(h+z)]}{\cosh(kh)}[\cos(kx)\cos(\sigma t) + \sin(kx)\sin(\sigma t)]$$

Here $\cos(kx)\cos(\sigma t)+\sin(kx)\sin(\sigma t)=\cos(kx-\sigma t)$:

$$\phi=\frac{Ag}{\sigma}\frac{\cosh[k(h+z)]}{\cosh(kh)}\cos(kx-\sigma t) \quad (\mathbf{34})$$

Then one other solution from $(\phi_2+\phi_3)$:

$$\phi=\frac{Ag}{\sigma}\frac{\cosh[k(h+z)]}{\cosh(kh)}\sin(kx-\sigma t) \quad (\mathbf{35})$$

In linear theoty , wave height be difined as 2A (H=2A), so (35) also could be expressed as the form was referred at(18):

$$\phi=\frac{Hg}{2\sigma}\frac{\cosh[k(h+z)]}{\cosh(kh)}\sin(kx-\sigma t)$$

Part 4: How to express wave profile

From (19):

$$\eta=-\frac{1}{g}\left(\frac{\partial\varphi}{\partial t}\right)\ (\mathrm{z}=0)$$

And substituting (18) into (19):

$$\eta=-\frac{1}{g}\left(\frac{\partial}{\partial t}\frac{Hg}{2\sigma}\frac{\cosh[k(h+z)]}{\cosh(kh)}\sin(kx-\sigma t)\right)\ \ (\mathrm{z}=0)$$

$$\eta=-\frac{1}{g}\frac{Hg}{2\sigma}\frac{\cosh[k(h+z)]}{\cosh(kh)}\left(\frac{\partial}{\partial t}\sin(kx-\sigma t)\right)\ \ (\mathrm{z}=0)$$

$$\eta=-\frac{1}{g}\frac{Hg}{2\sigma}\frac{\cosh[k(h+z)]}{\cosh(kh)}(-\sigma)\cos(kx-\sigma t)\ \ (\mathrm{z}=0)$$

$$\eta=\frac{1}{g}\frac{Hg}{2\sigma}\sigma\frac{\cosh[k(h+z)]}{\cosh(kh)}\cos(kx-\sigma t)\ \ (\mathrm{z}=0)$$

$$\eta=\frac{1}{g}\frac{Hg}{2}\frac{\cosh[kh]}{\cosh(kh)}\cos(kx-\sigma t)$$

$$\eta=\frac{1}{g}\frac{Hg}{2}\cos(kx-\sigma t)$$

Finally obtain the profile of wave:

$$\eta=\frac{H}{2}\cos(kx-\sigma t) \quad (\mathbf{36})$$

Part 5: Wave Celerity and group velocity

Set reference system on a move wave, η should be appear stationary, so do in (36):

$$kx-\sigma t=const.$$

Then just do a derivation of t:

$$k\frac{\partial x}{\partial t}-\sigma=0$$

$$\frac{\partial x}{\partial t}=\frac{\sigma}{k}=\frac{(2\pi/\mathrm{T})}{(2\pi/\mathrm{L})}$$

$$\mathrm{C}=\frac{\partial x}{\partial t}=\frac{L}{T} \tag{37}$$

From (21):

$$\frac{\partial^2\varphi}{\partial t^2}+g\frac{\partial\varphi}{\partial z}=0\ \ (\mathrm{z}=0)$$

Substituting (18) into first term (21):

$$\frac{\partial^2\varphi}{\partial t^2}=\frac{Hg}{2\sigma}\frac{\cosh\left[k(h+z)\right]}{\cosh\left(kh\right)}\frac{\partial^2}{\partial t^2}\{\sin\left(kx-\sigma t\right)\}$$

$$=\frac{Hg}{2\sigma}\frac{\cosh\left[k(h+z)\right]}{\cosh\left(kh\right)}(-\sigma)\frac{\partial}{\partial t}\{\cos\left(kx-\sigma t\right)\}$$

$$=\frac{Hg}{2\sigma}\frac{\cosh\left[k(h+z)\right]}{\cosh\left(kh\right)}(-\sigma)(-\sigma)(-)\sin\left(kx-\sigma t\right)$$

$$\frac{\partial^2\varphi}{\partial t^2}=(-\sigma)\frac{Hg}{2}\frac{\cosh\left[k(h+z)\right]}{\cosh\left(kh\right)}\sin\left(kx-\sigma t\right) \tag{38}$$

And second term:

$$\frac{\partial\varphi}{\partial z}=\frac{Hg}{2\sigma}\frac{1}{\cosh\left(kh\right)}\sin\left(kx-\sigma t\right)\frac{\partial}{\partial z}\cosh\left[k(h+z)\right]$$

$$\frac{\partial\varphi}{\partial z}=\frac{Hg}{2\sigma}k\frac{\sinh\left[k(h+z)\right]}{\cosh\left(kh\right)}\sin\left(kx-\sigma t\right) \tag{39}$$

Substitute them into (21):

$$(-\sigma)\frac{Hg}{2}\frac{\cosh\left[k(h+z)\right]}{\cosh\left(kh\right)}\sin\left(kx-\sigma t\right)+g\frac{Hg}{2\sigma}k\frac{\sinh\left[k(h+z)\right]}{\cosh\left(kh\right)}\sin\left(kx-\sigma t\right)=0$$

Where z = 0:

$$(-\sigma)\frac{Hg}{2}\sin\left(kx-\sigma t\right)+\frac{g^2H}{2\sigma}k\sin\left(kx-\sigma t\right)\tanh\left(kh\right)=0$$

$$(-\sigma)\frac{Hg}{2}+\frac{g^2H}{2\sigma}k\tanh\left(kh\right)=0$$

$$(-\sigma)g+g^2\frac{k}{\sigma}\tanh\left(kh\right)=0$$

$$g\frac{k}{\sigma}\tanh\left(kh\right)=\sigma$$

$$\sigma^2=kg\tanh\left(kh\right)$$

Therefore, finally obtain:

(40)

$$\sigma^2 = kg\tanh(kh)$$

Eq. (40) is useful to obtain k or L from the wave frequency σ or T, here substituting

$$C = \frac{\sigma}{k}$$

into (40):

$$C^2k^2 = kg\tanh(kh)$$

$$C^2 = \frac{g}{k}\tanh(kh)$$

Also here $k = 2\pi/L$:

$$C^2 = \frac{gL}{2\pi}\tanh(kh)$$

$$C = C^{-1}\frac{gL}{2\pi}\tanh(kh)$$

From (37):

$$C = \left(\frac{L}{T}\right)^{-1}\frac{gL}{2\pi}\tanh(kh)$$

$$C = \frac{T}{L}\frac{gL}{2\pi}\tanh(kh)$$

$$C = \frac{gT}{2\pi}\tanh(kh) \quad (41)$$

And about group velocity of wave:

$$C_g = \frac{d\sigma}{dk} = \frac{d}{dk}\sqrt{kg\tanh(kh)}$$

$$C_g = \frac{1}{2}\left[1 + \frac{2kh}{\sinh(2kh)}\right]\frac{gT}{2\pi}\tanh(kh) \quad (42)$$

Part 6: Other expressions for solve question1

Some expressions could be used but here is not enough time to write down derivation of them:

1. Expression for wave energy:

$$\mathrm{E} = \frac{\gamma H^2}{8} \quad (43)$$

2. Expression for wave energy flux:

$$P = C_gE \quad (44)$$

3. In a shoaling process, solution based on **linear wave theory** and **conservation of energy flux**.

A shoaling coefficient dirived by linear wave theory should be:

$$K_S = \frac{H_i}{H_o} = \left(\frac{2\cosh^2(kh)}{\sinh(2kh) + 2kh}\right)^{\frac{1}{2}} \quad (45)$$

Where: K_S is the shoaling coefficient.
H_i is wave height at a particular point of interest (x_i).
H_o is the original wave height in deep water.

From these parts of linear wave theory, Part6 start solve question1:

Part 7: Calculate the cross-shore distributions of wave height (H).
Conditions were written in page1, where : H=2.0 m and T=6.0 s at the offshore end (x=0).

Start from (40):

$$\sigma^2 = kg\tanh(k\mathrm{h}) \tag{Q1.1}$$

Where (x=0), $\sigma = 2\pi/\mathrm{T}$:

$$(2\pi/\mathrm{T})^2 = kg\tanh(kh_{x=0})$$

h(depth)=100-x, $h_{x=0} = 100$, T=6.0 $\tanh(kh_{x=0}) \approx 1$:

$$(2\pi/6)^2 = kg$$

We want obtain k from this equation, rearranging it:

$$k = \frac{4\pi^2}{36g}$$

Using $\mathrm{g} = 9.8\mathrm{m/s}^2$:

$$k = \frac{4\pi^2}{36 \times 9.8}$$

Obtain:

$$\mathrm{k} \approx 0.1119 \tag{Q1.2}$$

$$\frac{2\pi}{\mathrm{L}} \approx 0.1119$$

$$\mathrm{L}_{x=0} \approx \frac{2\pi}{0.1119} \approx 56.15\mathrm{m}$$

Where, h=100m>>L/20=2.8075m, it turely a deep water conditon.

About shallow condition:

$$\mathrm{H}_{x=0} = 1.0m$$

There must be a critical water depth, that $k_d = k_s$

$$h_d > h_c > h_s \tag{Q1.3}$$

From (44):

$$P = C_g E = \frac{1}{2}\left[1 + \frac{2kh}{\sinh(2kh)}\right]\frac{g}{k}\tanh(kh)\frac{\gamma H^2}{8}$$

Where deep water:

$$P_{deep} = \frac{1}{2}\left[1 + \frac{4k_d h_d}{\mathrm{e}^{2k_d h_d}}\right] g h_d \frac{\gamma {H_d}^2}{8}$$

Where shallow water:

$$P_{shallow} = g h_s \frac{\gamma {H_s}^2}{8}$$

Assumed **conservation of energy flux** ,so let $P_{shallow} = P_{deep}$**:**

If $k_d = k_{x=100} = k,\ h_d = h_s = h,\ H_s \approx H_d$:

$$\frac{1}{2}\left[1 + \frac{4k_d h_d}{\mathrm{e}^{2k_d h_d}}\right] g h_d \frac{\gamma {H_d}^2}{8} = g h_s \frac{\gamma {H_s}^2}{8}$$

$$\frac{4k_d h_d}{\mathrm{e}^{2k_d h_d}} \approx 1 \qquad \textbf{(Q1.4)}$$

This function of h is less than 1 all the range difined, but can found a point that it have maximum.

Do derivation of it on h:

$$4k_d \approx 2k_d \mathrm{e}^{2k_d h}$$
$$2 \approx \mathrm{e}^{2k_d h}$$

Let $k_d = 0.1119$:

$$\mathrm{e}^{2\times 0.1119 h} = 2$$

Where, finally:

$$h_c = h \approx 3.1$$

So from (Q1.4) obtain the critical depth h_c:

$$h_d > 3.1\mathrm{m} > h_s \qquad \textbf{(Q1.5)}$$

If H_d do not equal to H_s:

$$g h_s \frac{\gamma {H_s}^2}{8} = \frac{1}{2}\left[1 + \frac{2kh}{\sinh(2kh)}\right] \frac{g}{k} \frac{\gamma H^2}{8}$$

$$h_s {H_s}^2 = \frac{1}{2}\left[1 + \frac{4kh}{e^{2kh}}\right] \frac{1}{k} H^2$$

So it reduced a function about Hs and hs, hs was difined in (Q1.5)

$$h_s {H_s}^2 = \frac{1}{2}\left[1 + \frac{4kh}{e^{2kh}}\right] \frac{1}{k} H^2$$

$$h_s {H_s}^2 = \left[\frac{1}{2k} + \frac{2h}{e^{2kh}}\right] H^2$$

Where $k_d = 0.1119, h_d = 100, H_d = 2, h_s$ is a variable the depth of shallow water:

$$h_s {H_s}^2 = \left[\frac{1}{2k} + \frac{2h}{e^{2kh}}\right] H^2$$

$$h_s {H_s}^2 \approx 35.75$$

$${H_s}^2 \approx \frac{35.75}{h_s} \quad (1 \le h_s < 3.1)$$

Finally shallow water conditon:

$$H_s \approx \sqrt{\frac{35.75}{h_s}} \quad (1 \le h < 3.1) \qquad \textbf{(Q1.6)}$$

Deep condition, we have known wave number in deep water so just substituting it into (45):

$$\frac{H_d}{2} = \left(\frac{2\cosh^2(kh)}{\sinh(2kh) + 2kh}\right)^{\frac{1}{2}}$$

Then the wave height H:

$$H = \left(\frac{8\cosh^2(kh)}{\sinh(2kh) + 2kh}\right)^{\frac{1}{2}} \qquad \textbf{(Q1.7)}$$

From (Q1.7), and the result from (Q1.2):

$$H_d = \left(\frac{8\cosh^2(0.1119h)}{\sinh(0.2238h) + 0.2238h}\right)^{\frac{1}{2}} \quad (6.032 \le h \le 100) \qquad \textbf{(Q1.8)}$$

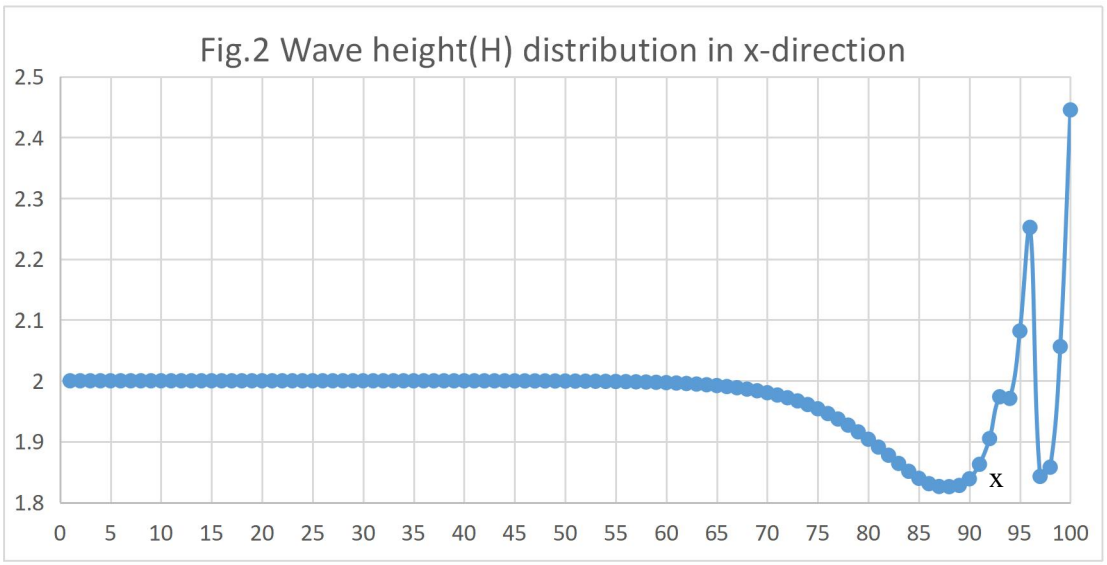
Where (Q1.6) and (Q1.8), h :

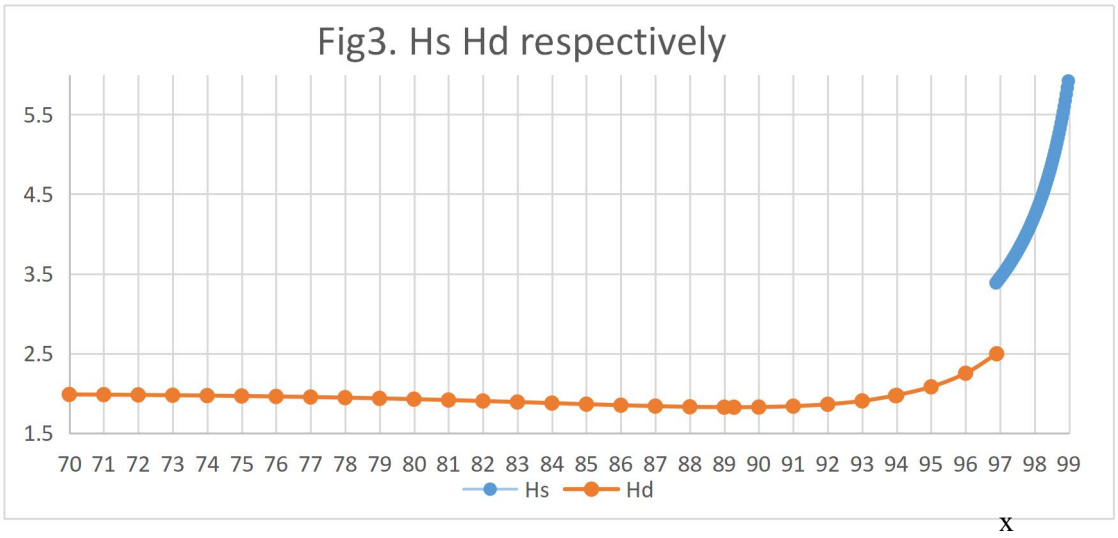
$$h = 100 - x \quad (0 \le \mathrm{x} \le 99)$$

Therefore finally, the result of the cross-shore distributions of the wave height (H):

$$\begin{cases} H_s \approx \sqrt{\dfrac{35.75}{h_s}} \quad (1 \le h < 3.1) \\ H_d = \left(\dfrac{8\cosh^2(0.1119h)}{\sinh(0.2238h) + 0.2238h}\right)^{\frac{1}{2}} \quad (3.1 \le h \le 100) \\ h = 100 - x \quad (0 \le \mathrm{x} \le 99) \end{cases} \qquad \textbf{(Q1.9)}$$

Plot (Q1.9) as fig.2 and fig.3 :





There is a discontinuity point between there 2 functions of wave height distribution, because the model was used in part 7, it is derived from flat bed, and maybe the wave brokes between x=95 and x=99.

But in my opinions shoaling coefficient can not be applied with a flat bed condition.

So next part , i tried to remodel this question, for solve this question more naturally.

Part 8: Re-modeling of this question.

From the model like Fig.4 , considering the gradient of sea bed is $\tan\alpha$:

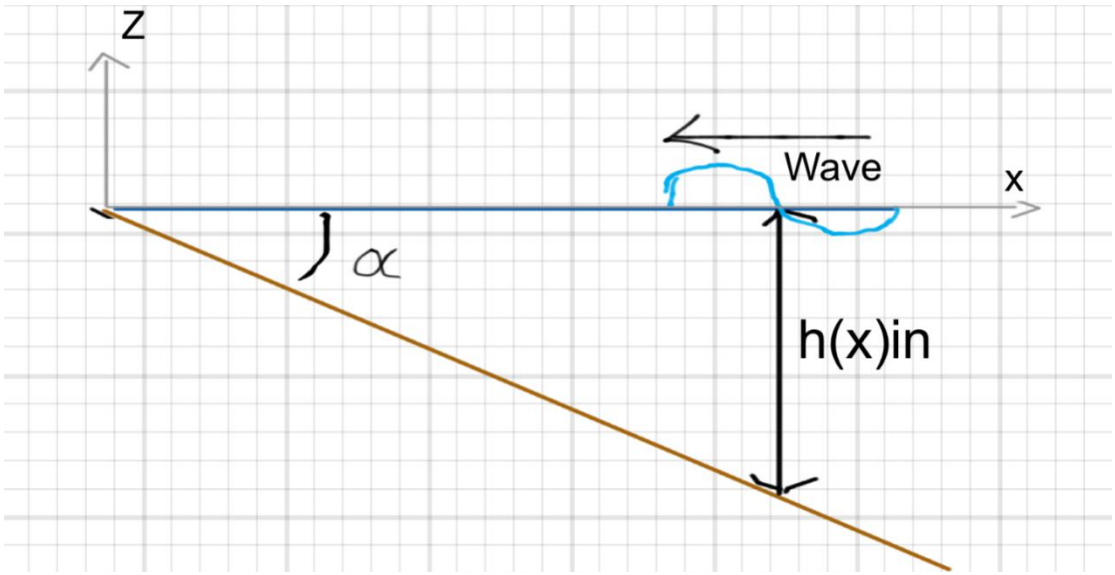


Fig.4 A sketch of the model

Where the position of coordinate have changed from question, but it is more easily to consider gradient of sea bed. Obtain the distribution of parameters, and transform them back to coordinate defined by question 1, is still easily.

And the express of depth become:

$$h(x) = x\,tan\,\alpha \qquad \textbf{(Q1.10)}$$

In this condition, α is equal to $\pi/4$, and x in the range $(1 \le x \le 100m)$ so it reduced :

$$h(x) = x\ (1 \le x \le 100m) \qquad \textbf{(Q1.11)}$$

Then with this simplified coordinate, slove Question1 based on these equations:

i. The Laplace equation, and the boundary was defined:

$$\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial z^2}=0,\ (1 \le x \le 100m,\ -h(x) \le z \le 0); \qquad \textbf{(Q1.12)}$$

ii. From KFSBC (21):

$$\frac{\partial^2\phi}{\partial t^2}+g\frac{\partial\phi}{\partial z}=0\ (\mathrm{z}=0); \qquad \textbf{(Q1.13)}$$

iii. From BBC(14) and bottom profile (Q1,10):

$$\frac{\partial\phi}{\partial z}+\frac{\partial\phi}{\partial x}\frac{\partial h}{\partial x}=0,\quad \frac{\partial h}{\partial x}=\tan\alpha\,;$$

Therefore obtain the BBC of shoaling process with α is equal to $\pi/4$:

$$\frac{\partial \phi}{\partial z}+\frac{\partial \phi}{\partial x}=0\ (1\le x\le 100m,\ -h(x)\le z\le 0); \tag{Q1.14}$$

iv. Conservation of wave energy flux:

$$\frac{1}{T}\int_{-h}^{0}-\rho\frac{\partial \phi}{\partial t}\frac{\partial \phi}{\partial x}dzdt=\frac{1}{16}pgH_d^2\sigma_d \tag{Q1.15}$$

Where: ϕ is the velocity potential function; T is Period of wave; t is Time instant; ρ is density of water(sea water); p is the presure at a point in wave; g is gravitational acceleration; H_d is the wave height in the range of deep water; σ_d is the wave circular frequency in the range of deep water. Then from (40) and the deep water condition $\tanh(k_d h)\cong 1$ obtain the equation below:

$$\sigma_d{}^2=k_d g \tag{Q1.16}$$

Where k_d is the wave number in the range of deep water.

Then, assumed that the solution of velocity potential as a form like:

$$\phi=\varphi(z)e^{i(kx-\sigma t)}. \tag{Q1.17}$$

Just like what was done in Part 3, substituting (Q1.17) into (Q1.12) and (Q1.13):

$$\phi=-\frac{A}{\sigma k}i\left[gk\cosh(kz)+\sigma^2\sinh(kz)\right]e^{i(kx-\sigma t)}, \tag{Q1.18}$$

Where A is the Amplitude of wave; i is imaginary symbol.

Then substituting (Q1.18) into (Q1.14) obtain:

$$\sigma^2=gk\frac{\cosh(kh)+i\sinh(kh)}{\sinh(kh)+i\cosh(kh)} \tag{Q1.19}$$

or

$$\mathrm{c}=\frac{\sigma}{k}=\left\{\frac{g[\cosh(kh)+i\sinh(kh)]}{k[\sinh(kh)+i\cosh(kh)]}\right\}^{\frac{1}{2}}. \tag{Q1.20}$$

Finally, velocity, period of wave and the wave length respective:

$$\mathrm{c}=\frac{gT}{2\pi}\frac{\cosh(kh)+i\sinh(kh)}{\sinh(kh)+i\cosh(kh)}, \tag{Q1.21}$$

$$T=\left\{\frac{2\pi L[\sinh(kh)+i\cosh(kh)]}{g[\cosh(kh)+i\sinh(kh)]}\right\}^{\frac{1}{2}}, \tag{Q1.22}$$

(Q1.23)

$$L = \frac{gT^2}{2\pi}\frac{\sinh(kh) + i\cosh(kh)}{\cosh(kh) + i\sinh(kh)}.$$

Part 9: How the sea bed gradient effect wave velocity and wave length.

A shoalling process here we assume that:

1. Period (T) is a constant from deep water to shallow water.
2. Energy of wave is conservative in z-x plane.
3. Wave length, Wave height and velocity of wave in depth water are L_d,H_d,c_d respectively, and there are L_s,H_s,c_s same as them in shallow water.

As 1. the Period (T) is a constant from deep water to shallow water:

$$\sigma^2 = gk\frac{\cosh(kh) + i\sinh(kh)}{\sinh(kh) + i\cosh(kh)} = \sigma_d^2 = gk_d \qquad (\mathbf{Q1.24})$$

By (Q1.21),(Q1.22),(Q1.23),(Q1.24), can obtain ratio of velocity and wave length and wave number by a constant T:

$$\frac{c_s}{c_d} = \frac{\mathrm{L}_s}{\mathrm{L}_d} = \frac{\mathrm{k}_s}{\mathrm{k}_d} = \frac{1 + i\tanh(2\pi\frac{h}{\mathrm{L}_d}\frac{\mathrm{L}_d}{\mathrm{L}_s})}{\tanh\left(2\pi\frac{h}{\mathrm{L}_d}\frac{\mathrm{L}_d}{\mathrm{L}_s}\right) + i} = \tanh\left(2\pi\frac{h}{\mathrm{L}_d}\frac{\mathrm{L}_d}{\mathrm{L}_s}\right). \qquad (\mathbf{Q1.25})$$

Now by that equation , we can start discuss the object of this question, wave height from (Q1.18), just take the real part of it:

$$Re\phi = -\frac{A}{\sigma k}\left[gk\cosh(kz) + \sigma^2\sinh(kz)\right]\sin(kx - \sigma t). \qquad (\mathbf{Q1.26})$$

Then, substituting (Q1.26) into the conservation of wave energy flux equation (Q1.15), we can obatin the expression of the Shoalling coefficient (K_s) from them.

$$K_s = \frac{H_s}{H_d} = \frac{\mathrm{L}_d}{\mathrm{L}_s} \times 2^{\frac{1}{2}}\left[\sinh(2kh)\left(1 - \frac{2kh}{\sinh(2kh)}\right) + 2\frac{\mathrm{L}_d}{\mathrm{L}_s}\cosh(2kh)\left(\frac{2kh}{\cosh(2kh)} - 1\right) + \left(\frac{\mathrm{L}_d}{\mathrm{L}_s}\right)^2\sinh(2kh)\left(1 + \frac{2kh}{\sinh(2kh)}\right)\right]^{-\frac{1}{2}}. \qquad (\mathbf{Q1.27})$$

Where $\mathrm{L}_d/\mathrm{L}_s$ can be confirmed by (Q1.25), this euqation can help us to found out the distribution of wave height.

Review the boundary condition in Question 1:

H=2.0 m and T=6.0 s at the offshore end (x=0).

Transform this condition to our coodinate:

H=2.0 m and T=6.0 s at the offshore end (x=100) and , where ($1 \le \mathrm{x} \le 100$).

This is the deep water condition:

$$\begin{cases} \sigma_d = 2\pi/T_d,\ T_d = 6s; \\ {\sigma_d}^2 = k_d g,\ k_d = 2\pi/L_d; \\ H_d = 2m. \end{cases}$$

(**Q1.28**)

$$\begin{cases} \sigma_d = \dfrac{\pi}{3},\ T_d = 6s; \\ k_d = \dfrac{2\pi}{L_d} = \dfrac{{\sigma_d}^2}{g} = \dfrac{\pi^2}{9\times 9.8} \cong 0.1119; \\ L_d = \dfrac{9\times 9.8\times 2\pi}{\pi^2} = \dfrac{18\times 9.8}{\pi} \cong 56.15m \\ H_d = 2m. \end{cases}$$

First, substituting (Q1.28) into (Q1.25) and (Q1.27), first (Q1.25):

$$\frac{\mathrm{L}_s}{L_d} = \frac{\left[1 + i\tanh(2\pi\frac{h}{\mathrm{L}_d}\frac{\mathrm{L}_d}{\mathrm{L}_s})\right]\left[\tanh\left(2\pi\frac{h}{\mathrm{L}_d}\frac{\mathrm{L}_d}{\mathrm{L}_s}\right) - i\right]}{\left[\tanh\left(2\pi\frac{h}{\mathrm{L}_d}\frac{\mathrm{L}_d}{\mathrm{L}_s}\right) + i\right]\left[\tanh\left(2\pi\frac{h}{\mathrm{L}_d}\frac{\mathrm{L}_d}{\mathrm{L}_s}\right) - i\right]},$$

$$\frac{\mathrm{L}_s}{L_d} = \frac{2\tanh\left(2\pi\frac{h}{\mathrm{L}_s}\right) + i\left[\tanh^2\left(2\pi\frac{h}{\mathrm{L}_s}\right) - 1\right]}{\tanh^2\left(2\pi\frac{h}{\mathrm{L}_s}\right) - 1},$$

$$\frac{\mathrm{L}_s}{L_d} = \left[\frac{2\tanh\left(2\pi\frac{h}{\mathrm{L}_s}\right)}{\tanh^2\left(2\pi\frac{h}{\mathrm{L}_s}\right) - 1} + i\frac{\tanh^2\left(2\pi\frac{h}{\mathrm{L}_s}\right) - 1}{\tanh^2\left(2\pi\frac{h}{\mathrm{L}_s}\right) - 1}\right],$$

$$\frac{\mathrm{L}_s}{L_d} = \left[\frac{2\tanh\left(2\pi\frac{h}{\mathrm{L}_s}\right)}{\tanh^2\left(2\pi\frac{h}{\mathrm{L}_s}\right) - 1} + i\right],$$

so

$$\frac{\mathrm{L}_d}{L_s} = \left[\frac{2\tanh\left(2\pi\frac{h}{\mathrm{L}_s}\right)}{\tanh^2\left(2\pi\frac{h}{\mathrm{L}_s}\right) - 1} + i\right];$$

Where $2\pi\frac{h}{\mathrm{L}_s} = kh$:

$$\frac{\mathrm{L}_d}{L_s} = \left[\frac{2\tanh(kh)}{\tanh^2(kh) - 1} + i\right]^{-1}$$

$$\frac{\mathrm{L}_d}{L_s} = \frac{\tanh^2(kh) - 1}{2\tanh(kh) + i}$$

$$\frac{\mathrm{L}_d}{L_s} = \frac{\left[\tanh^2(kh) - 1\right][2\tanh(kh)\ - i]}{[2\tanh(kh) + i][2\tanh(kh)\ - i]}$$

$$\frac{\mathrm{L}_d}{L_s} = \frac{2\left[\tanh^2(kh) - \tanh(kh)\right] + i\left[1 - \tanh^2(kh)\right]}{4\tanh^2(kh)\ - 1}$$

$$\frac{\mathrm{L}_d}{L_s}=\frac{2\left[\tanh^2(kh)-\tanh(kh)\right]}{4\tanh^2(kh)\ -1}+i\frac{1-\tanh^2(kh)}{4\tanh^2(kh)\ -1}$$

$$\mathrm{Re}\left(\frac{\mathrm{L_d}}{\mathrm{L_s}}\right)=\frac{2\left[\tanh^2(kh)-\tanh(kh)\right]}{4\tanh^2(kh)\ -1} \qquad \textbf{(Q1.29)}$$

$$\left(\frac{\mathrm{L}_d}{L_s}\right)^2=\left(\frac{\tanh^2(kh)-1}{2\tanh(kh)+i}\right)^2$$

$$\left(\frac{\mathrm{L}_d}{L_s}\right)^2=\frac{\tanh^4(kh)-2\tanh^2(kh)+1}{2\tanh(kh)+i}$$

$$\left(\frac{\mathrm{L}_d}{L_s}\right)^2=\frac{\left[\tanh^4(kh)-2\tanh^2(kh)+1\right][2\tanh(kh)-i]}{4\tanh^2(kh)\ -1}$$

$$\left(\frac{\mathrm{L}_d}{L_s}\right)^2=\frac{\left[\tanh^4(kh)-2\tanh^2(kh)+1\right][2\tanh(kh)-i]}{4\tanh^2(kh)\ -1}$$

$$\left(\frac{\mathrm{L}_d}{L_s}\right)^2=\frac{\left[2\tanh^5(kh)-4\tanh^3(kh)+2\tanh(kh)\right]}{4\tanh^2(kh)\ -1}+\frac{i\left[\tanh^2(kh)-\tanh^4(kh)-1\right]}{4\tanh^2(kh)\ -1}$$

$$Re\left(\frac{\mathrm{L}_d}{L_s}\right)^2=\frac{2\left[\tanh^5(kh)-2\tanh^3(kh)+\tanh(kh)\right]}{4\tanh^2(kh)\ -1} \qquad \textbf{(Q1.30)}$$

$$\left(\frac{\mathrm{L}_d}{L_s}\right)^3=\frac{\left[\tanh^4(kh)-4\tanh^2(kh)+1\right][2\tanh(kh)-i]}{4\tanh^2(kh)\ -1}\frac{\tanh^2(kh)-1}{2\tanh(kh)+i}$$

$$\left(\frac{\mathrm{L}_d}{L_s}\right)^3=\frac{\left[\tanh^4(kh)-4\tanh^2(kh)+1\right]}{4\tanh^2(kh)\ -1}\tanh^2(kh)-1$$

$$\left(\frac{\mathrm{L}_d}{L_s}\right)^3=\frac{\left[\tanh^4(kh)-4\tanh^2(kh)+1\right]\left[\tanh^2(kh)-1\right]}{4\tanh^2(kh)\ -1} \qquad \textbf{(Q1.31)}$$

The real part of (Q1.29) is a function of kh, here k is a constant euqal to 0.1119.

Then about (Q1.27): set (Q1.29) as $t(kh)$,(Q1.30) as $t_2(kh)$, (Q1.31) as $t_3(kh)$.

And $\sinh(2kh)$ as $s(2kh)$, $and\ \cosh(2kh)$ as c(2kh):

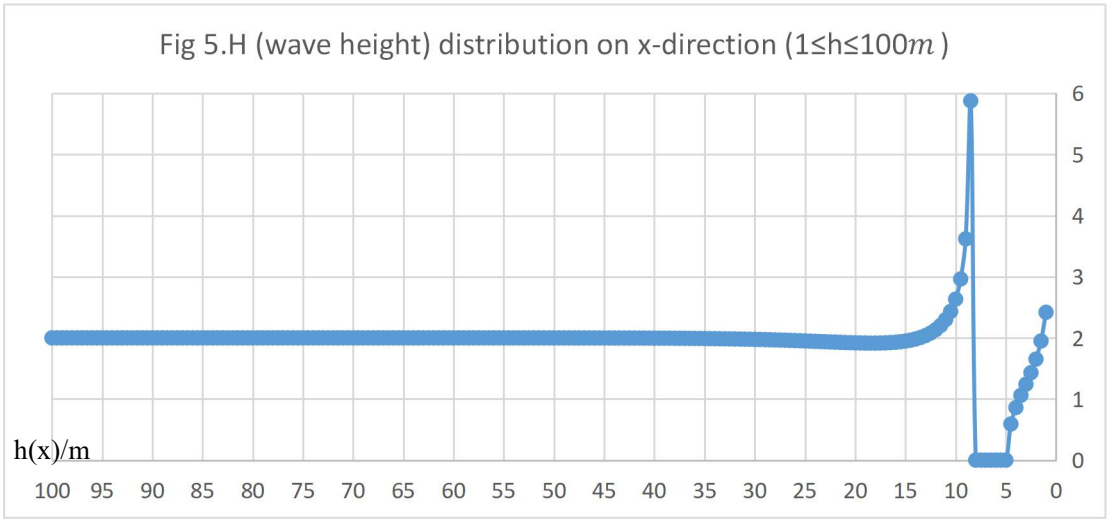
$$K_s=\frac{H_s}{H_d}=\sqrt{2}\left[t(kh)\mathrm{s}(2kh)\left(1-\frac{2kh}{\mathrm{s}\,(2kh)}\right)+2\,t_2(kh)\,\mathrm{c}\,(2kh)\left(\frac{2kh}{\mathrm{c}\,(2kh)}-1\right)\right.$$

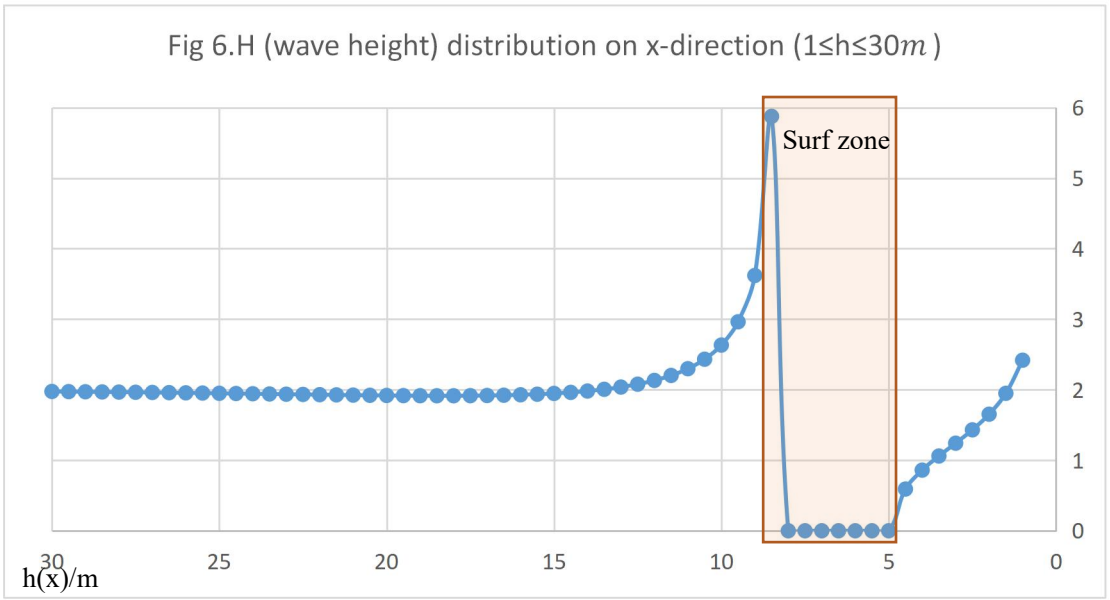
$$\left.+\,t_3(kh)\mathrm{s}(2kh)\left(1+\frac{2kh}{\mathrm{s}\,(2kh)}\right)\right]^{-\frac{1}{2}}. \qquad \textbf{(Q1.32)}$$

Where as (Q1.11):

$$h(x) = x\ (1 \le x \le 100m)$$

Finally, by (Q1.32) plot the distribution of H_s on x-direaction like Fig.5 and Fig.6 :
Obviously, **where the h changed from 9m to 4m the wave broke**.





Contrast Fig.5 , Fig.6 with Fig.4 Fig.3, obviously the distribution is close to flat condition out of Surf zone. So about others parts in Question1, using the model derived in flat bed.

Part 10: Distribution of wave velocity

There Wave Celerity and Group velocity both are included in wave velocity:

Here first discuss the Wave Celerity c:

First in deep water, from the conditions obtained in (Q1.28):

$$c_d^2 = \frac{g}{k} = \frac{9.8}{0.1119} \cong 87.58$$

$$c_d \cong 9.36$$

Then from (Q1.25):

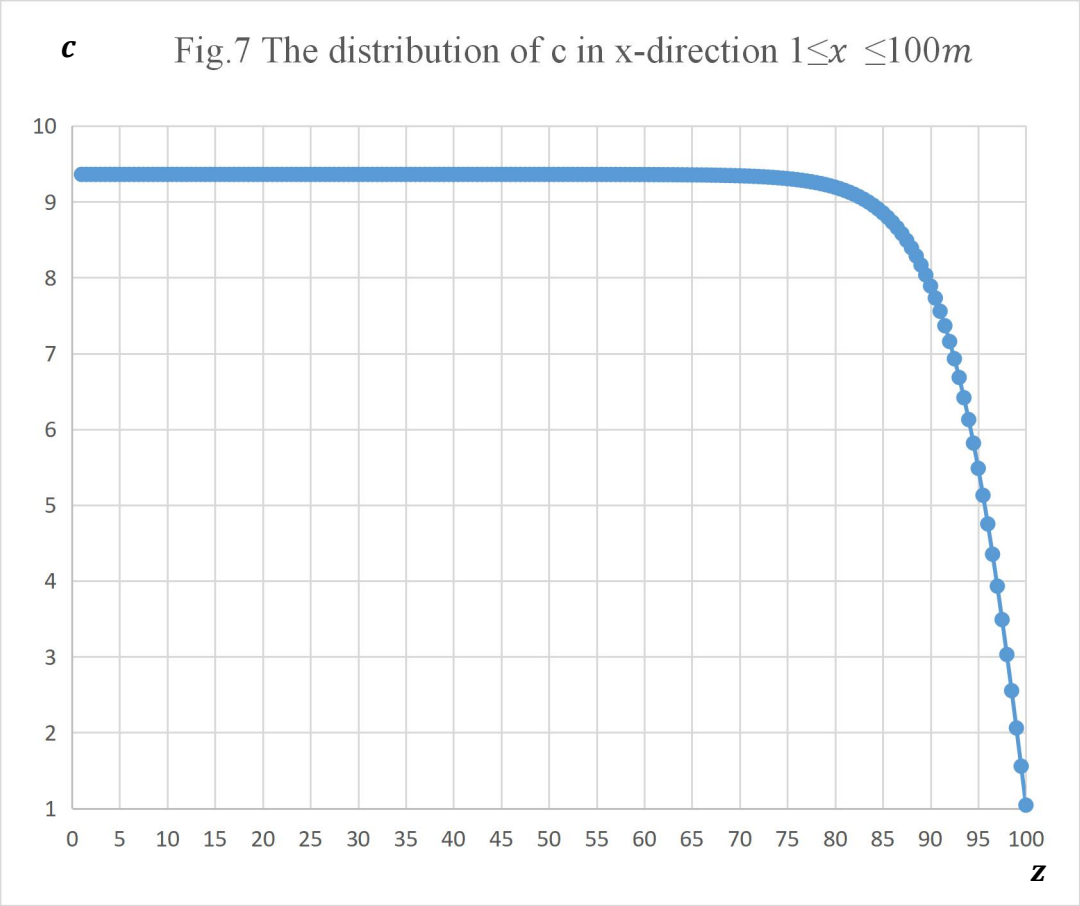
Just take the real part, and $2\pi\frac{h}{L_s} = kh$:

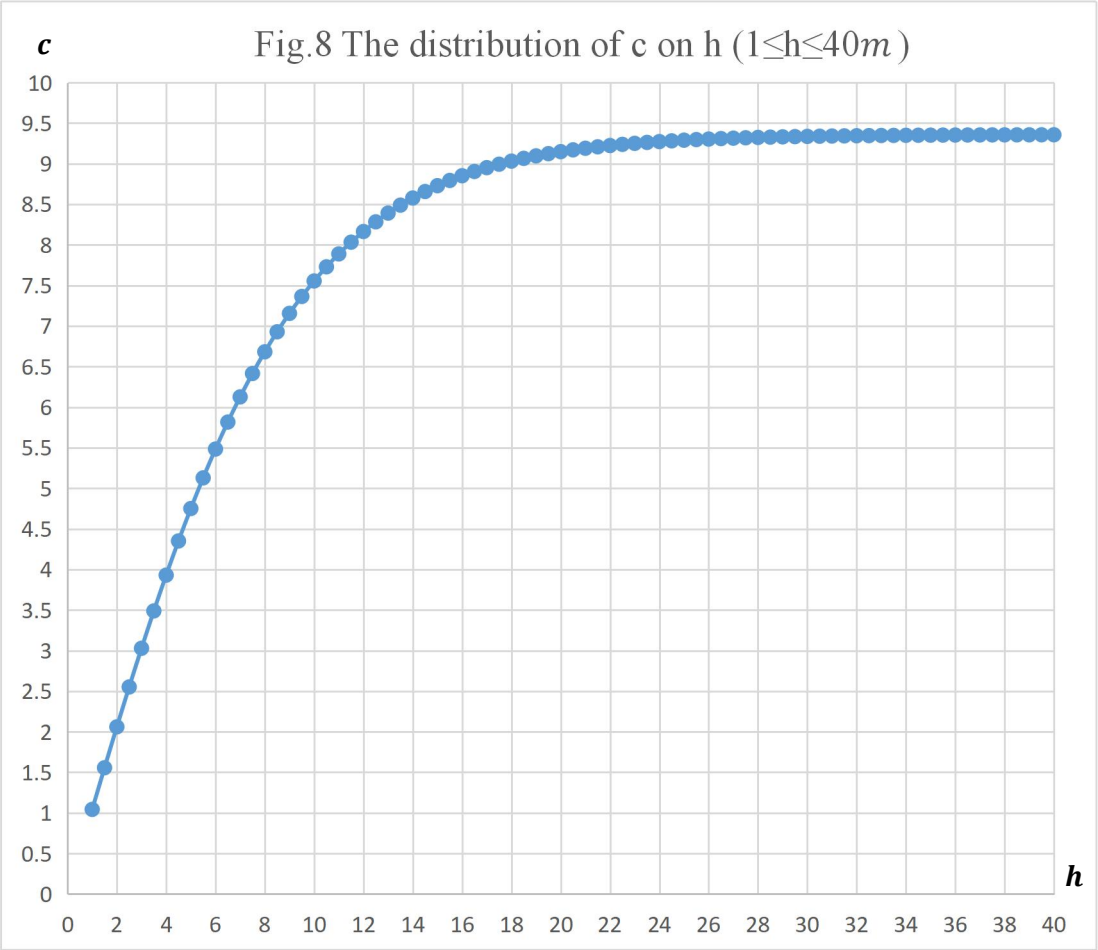
$$c_s = c_d \tanh(kh) \qquad \textbf{(Q1.33)}$$

Substituting constants into (Q1.33):

$$c_s = 9.36 \times \tanh(0.1119h) \qquad \textbf{(Q1.34)}$$

Plot (Q1.34) in the coordinate defined by Question1(Page1) as Fig.7 and Fig.8:





Then about Group velocity C_g:

From (42):

$$C_g = \frac{d\sigma}{dk}.$$

Here we assumed a constant k=0.1119.

$$\frac{C_{gs}}{C_{gd}} = \frac{\sigma_s}{\sigma_d} = \frac{T_s}{T_d} = 1. \qquad \textbf{(Q1.35)}$$

So the Group velocity in this condition under this assumption is a constant on x-direction.

Part 11. The maximum horizontal fluid velocity:

First from (18) and (21):

$$\phi = \frac{Hg}{2\sigma}\frac{\cosh[k(h+z)]}{\cosh(kh)}\sin(kx-\sigma t)$$

$$\frac{\partial^2 \phi}{\partial t^2} + g\frac{\partial \phi}{\partial z} = 0$$

Where $\sigma = 2\pi/T$, then ϕ can be expressed as:

$$\phi = \frac{\pi H}{kT}\frac{\cosh\left[k(h+z)\right]}{\cosh\left(kh\right)}\sin\left(kx - \sigma t\right) \qquad \textbf{(Q1.36)}$$

The horizontal velocity (in x-direction):

$$\mathrm{u} = \frac{\partial \phi}{\partial x} = \frac{\pi H}{T}\frac{\cosh\left[k(h+z)\right]}{\sinh\left(kh\right)}\cos\left(kx - \sigma t\right) \qquad \textbf{(Q1.37)}$$

The horizontal acceleration a_x:

$$a_x = \frac{\partial u}{\partial t} = \frac{2\pi^2 H}{T^2}\frac{\cosh\left[k(h+z)\right]}{\sinh\left(kh\right)}\sin\left(kx - \sigma t\right) \qquad \textbf{(Q1.38)}$$

When $\sin\left(kx - \sigma t\right) = 0$, $\cos\left(kx - \sigma t\right)$ is the maximum 1, then (Q1.37): become:

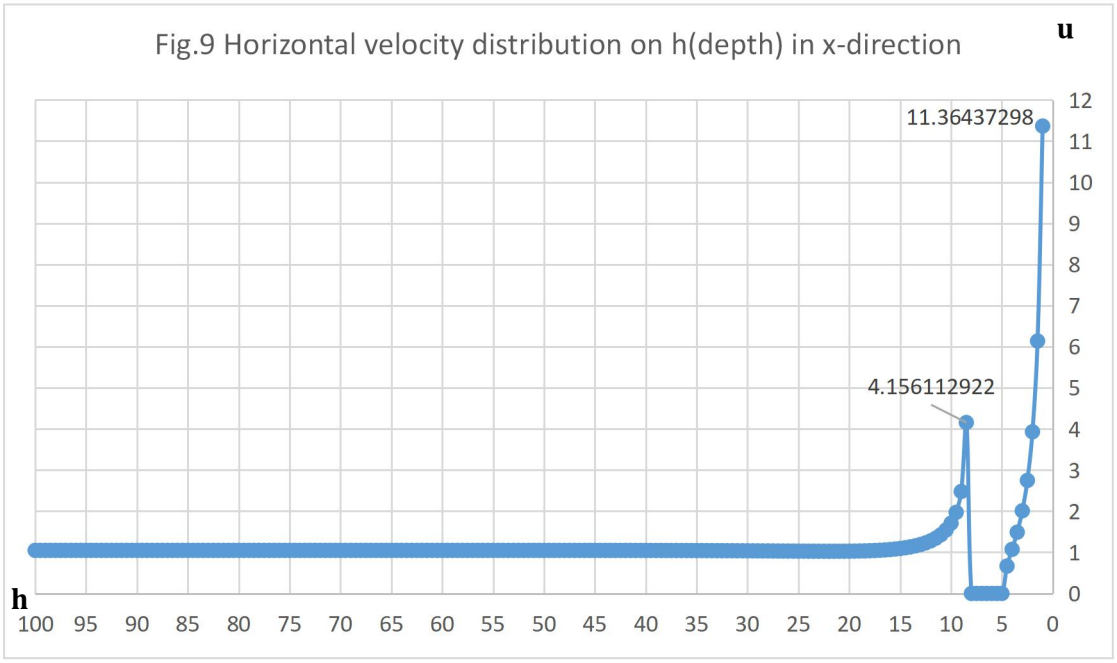
$$\mathrm{u} = \frac{\pi H}{T}\frac{\cosh\left[k(h+z)\right]}{\sinh\left(kh\right)}$$

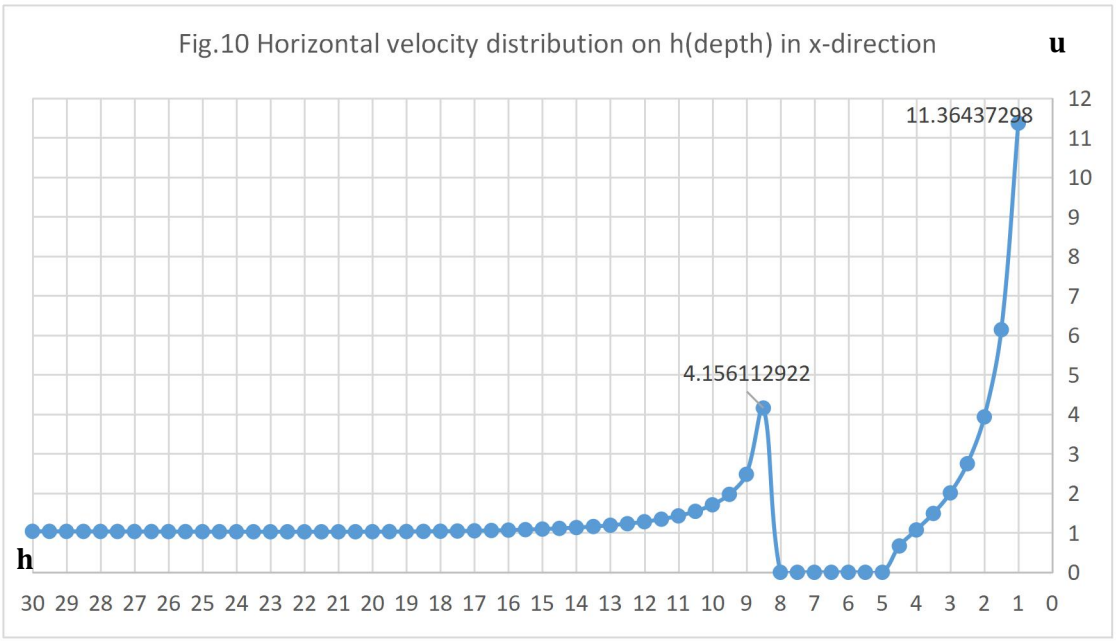
We set the coordinate at z = 0 so:

$$\mathrm{u} = \frac{\pi}{T}H\frac{\cosh\left(kh\right)}{\sinh\left(kh\right)}$$

$$\mathrm{u} = \frac{\pi}{T}H\tanh^{-1}\left(kh\right) \qquad \textbf{(Q1.39)}$$

Plot it as a function $u(H,h)$ Fig.9 and Fig.10 (Next page):





Therefore, before the surf zone, the maximum of horizontal fluid velocity should be $u \cong 4.156 m/s$ here, and in surf zone, the maximum become $u \cong 11.364 m/s$.

Stokes wave.

Using the wave height obtained in Qeustion1, Calculate surface elevation of the second-order Stokes wave at x=0 m and x=99.0 m.

Comment on the features of each surface shape (x=0 and x=99).

The surface elevation of second-order stokes waves:

$$\eta_2 = \frac{H^2}{16}\mathrm{k}\frac{\cosh 2\mathrm{kh} + 2}{\sinh^2(kh)\tanh(kh)}\cos 2(\mathrm{kx} - \sigma\mathrm{t}) \qquad \textbf{(Q2.1)}$$

Where

$$h = 100 - x \quad (0 \le \mathrm{x} \le 99)$$

Here when x=0 h=100,and x=99 h=1, the (Q2.1) reduced to 2 equation, at same time the k was obtained:

$$\eta_{21} = \frac{{H_{h=100}}^2}{16}\mathrm{k}\frac{\cosh(200\mathrm{k}) + 2}{\sinh^2(100k)\tanh(100k)}\cos(-2\sigma\mathrm{t})$$

Where: $H_{h=100} \cong 2m, \sigma = \pi/3$, $k = 0.1119$， reduced to a function of t：

$$\eta_{21} = \frac{k}{4}\frac{\cosh(200\mathrm{k}) + 2}{\sinh^2(100k)\tanh(100k)}\cos(-2\sigma\mathrm{t})$$

$$\eta_{21} = \frac{0.1119}{4}\frac{\cosh(22.38) + 2}{\sinh^2(11.19)\tanh(11.19)}\cos(-\frac{2\pi}{3}\mathrm{t})$$

$$\eta_{21} = 0.056\cos(-\frac{2\pi}{3}\mathrm{t}) \qquad \textbf{(Q2.2)}$$

Where: $H_{h=1} \cong 2.42m, \sigma = \pi/3$, $k = 0.1119$， reduced to a function of t：

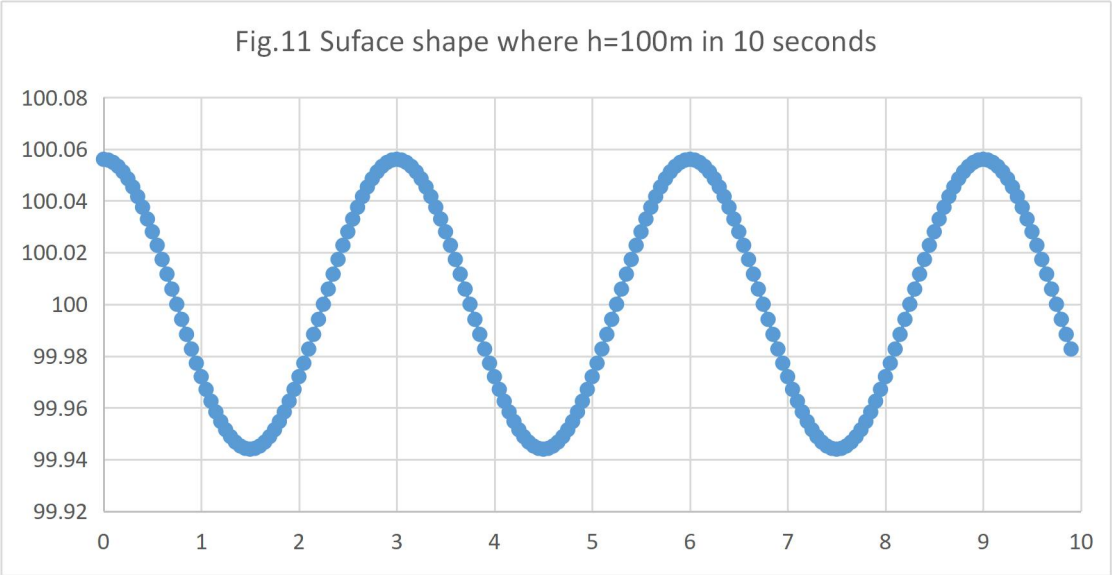
$$\eta_{22} = \frac{{H_{h=1}}^2}{16}\frac{\cosh(200\mathrm{k}) + 2}{\sinh^2(100k)\tanh(100k)}\cos(198\mathrm{k} - 2\sigma\mathrm{t})$$

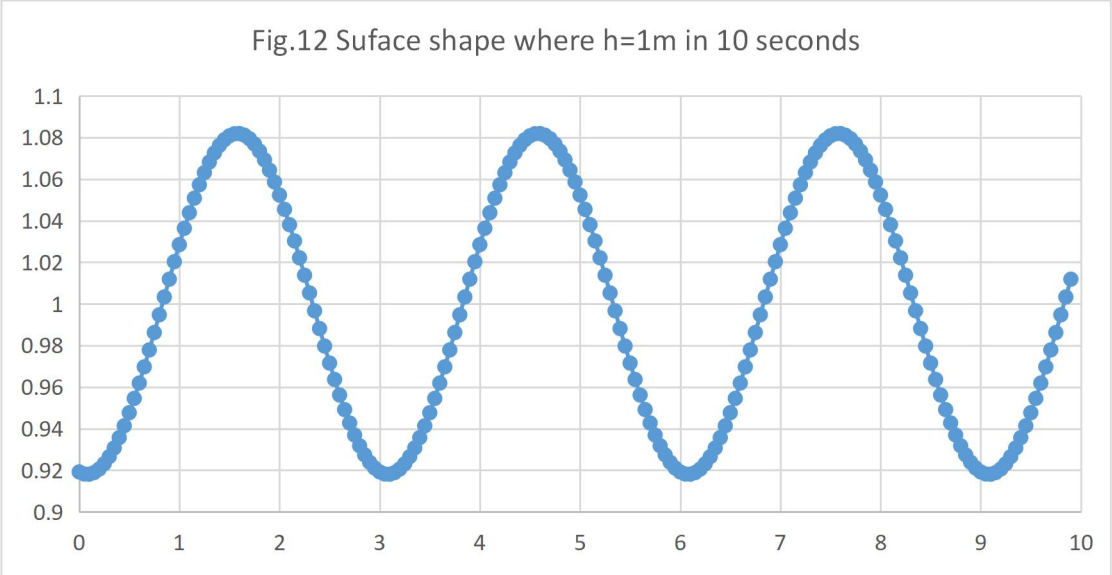
$$\eta_{22} = \frac{2.42^2}{16}0.1119\frac{\cosh(22.38) + 2}{\sinh^2(11.19)\tanh(11.19)}\cos(22.38 - 0.2238 - \frac{2\pi}{3}\mathrm{t})$$

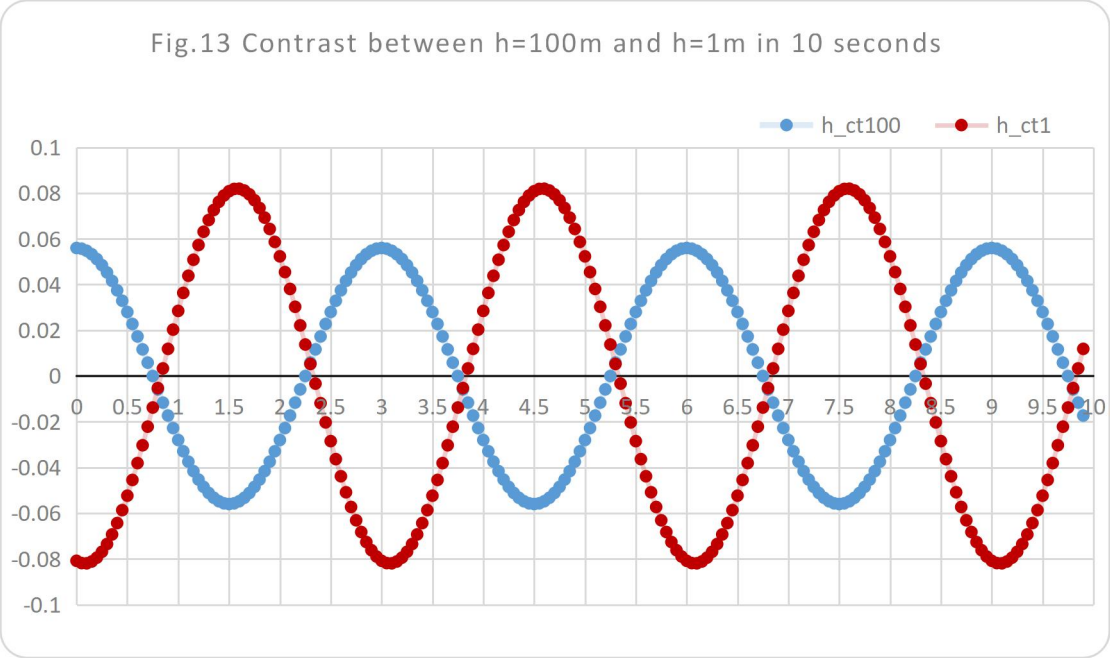
$$\eta_{22} = \frac{2.42^2}{16}0.1119\frac{\cosh(22.38) + 2}{\sinh^2(11.19)\tanh(11.19)}\cos(22.38 - 0.2238 - \frac{2\pi}{3}\mathrm{t})$$

$$\eta_{22} = 0.082\cos(22.16 - \frac{2\pi}{3}\mathrm{t}) \qquad \textbf{(Q2.3)}$$

Plot (Q2.2) (Q2.3) in Fig.11 and Fig.12, contrast them in Fig.13:







The surface elevation of the second-order Stokes wave at x=0 m and x=99.0 m were ploted as fig.11 and

fig.12. Comment on each shape based on fig.13.
From deep water to shallow water, the wave height become higher, and at these two points we selected, phase of them opposite with each other.
Beacuse of assumed the period is a constant, differences of them are phase, wave height and surely the maximum of vertical velocity of water particle.
We assumed the energy flux is constant in this process, with the wave trans to shallow area, change of wave height can be respected by liner wave theroy too, but stokes wave can describe the more tiny change of wave height where deep water like fig.11.

JONSWAP spectrum

Question:

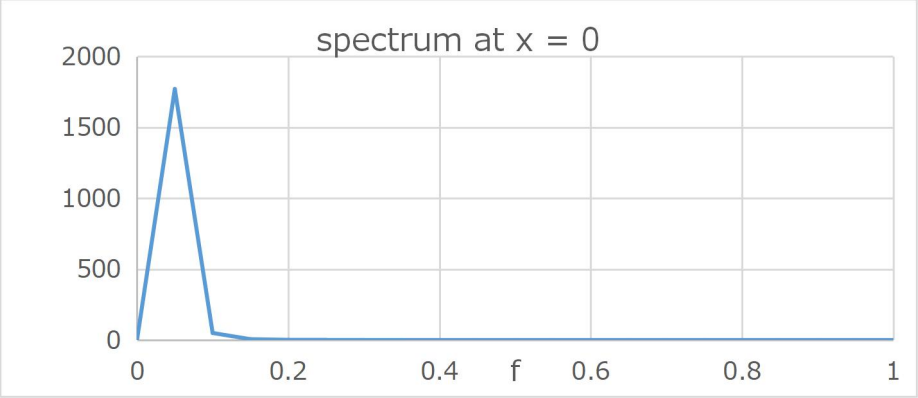
A fortran code to compute 1-D evolution of ocean wave specturm with shoaling process.

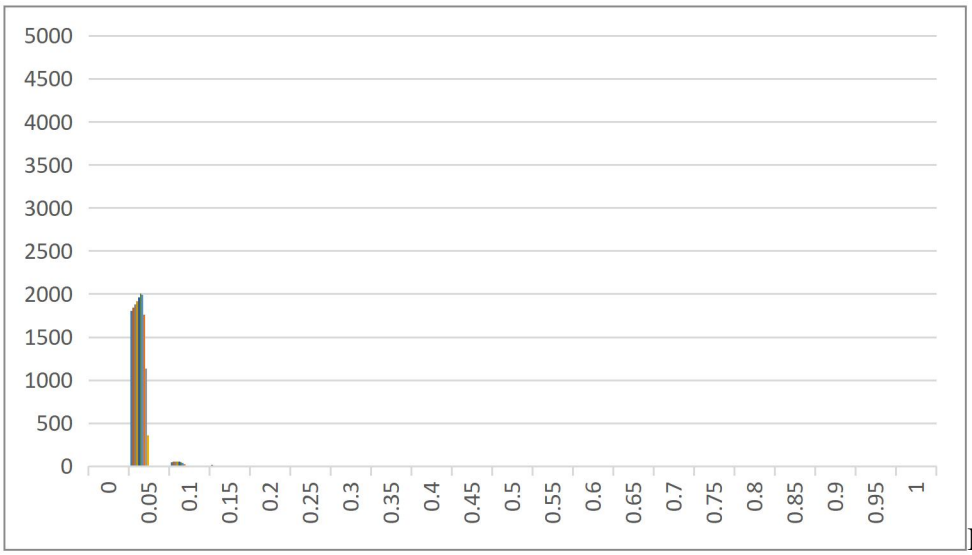
The code discribe wave field with the PM specturm is given at:

1. The offshore end (i = 0) with the offshore water depth h_0 = 12 m.
2. The wave spectrum is transported in x-direction.
3. The group velocity reforme in the shoaling process as the water depth decrease as
   $h = h_0 - 0.01x$ m.
4. The sequential spectrum deformation over the computational domain $0 \leq x \leq 1000$m.

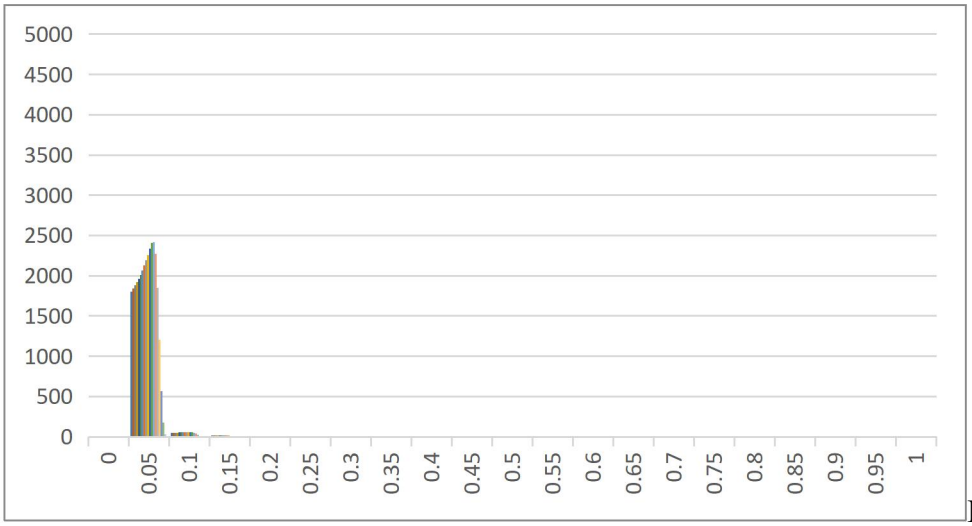
Modify the code (below) to compute the evolution of **JONSWAP** specturm, and create figures showing sequential specturm distributions.

Parameters: U_{10} = 20 [m/s], fetch = 10 [km]

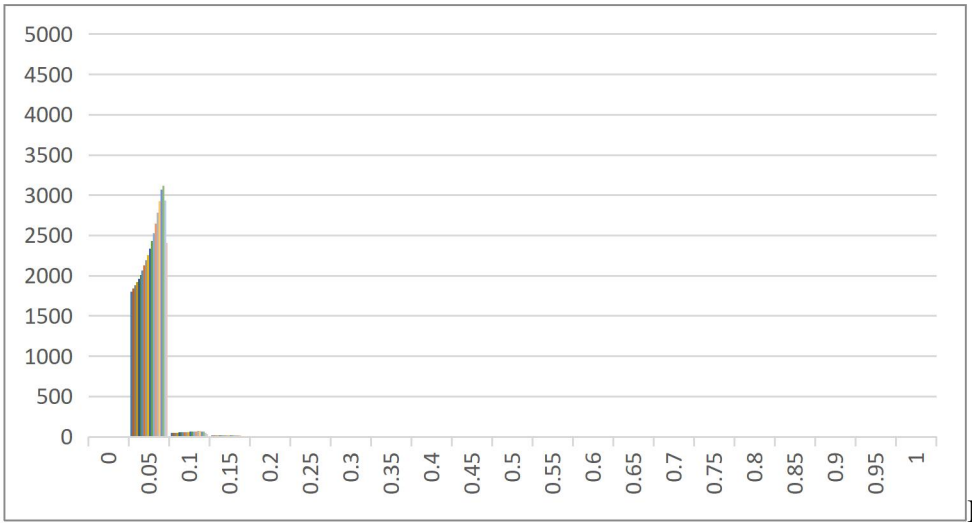




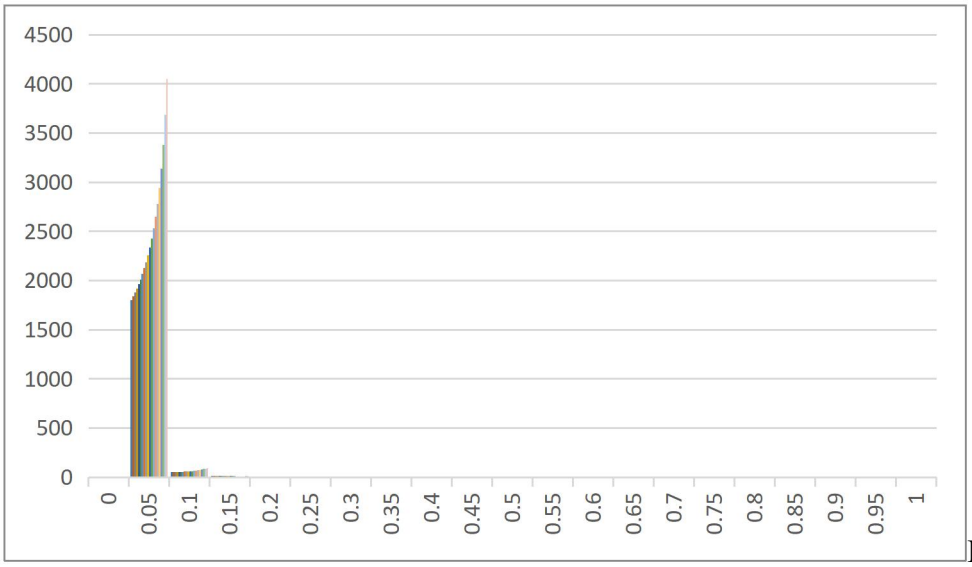
It=10



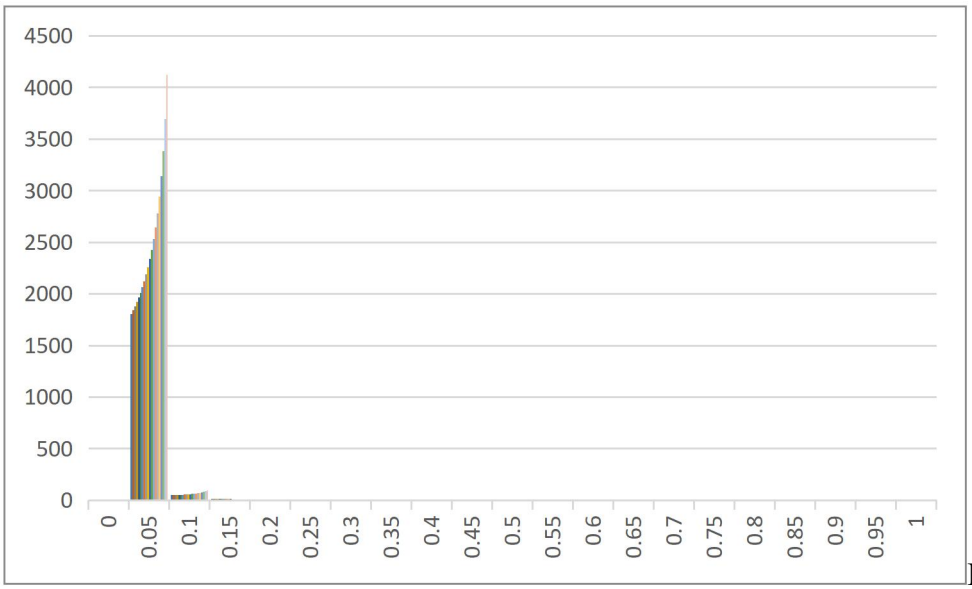
It=20



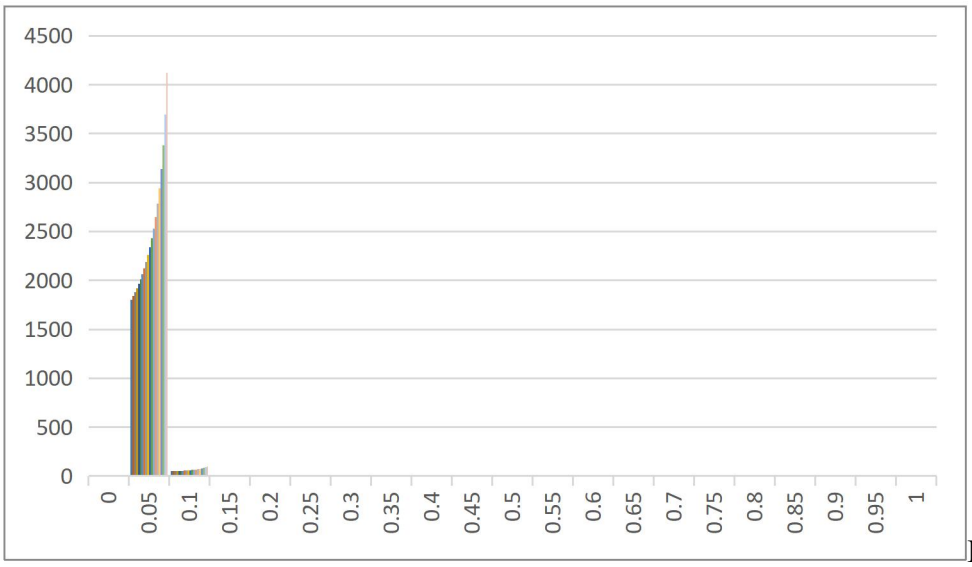
It=30



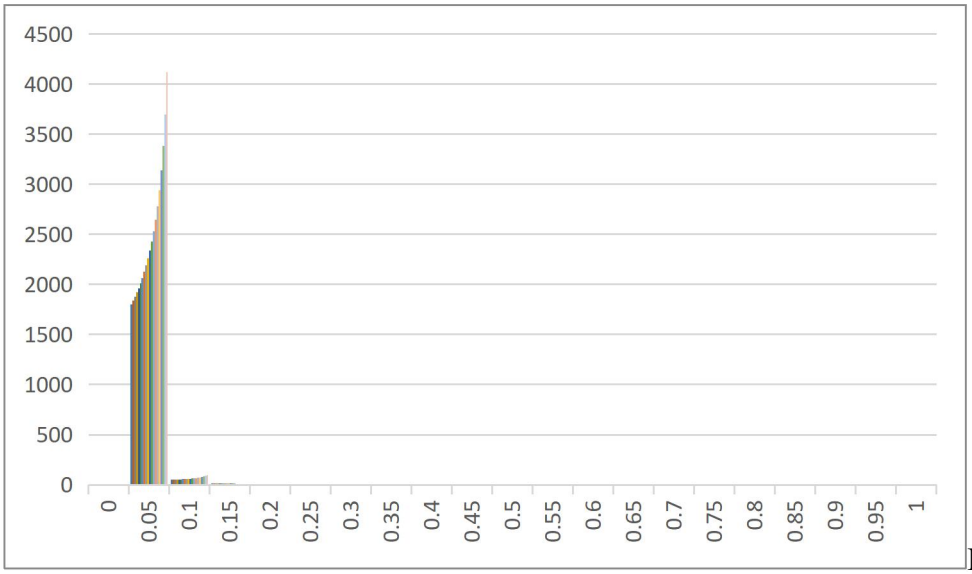
It=40



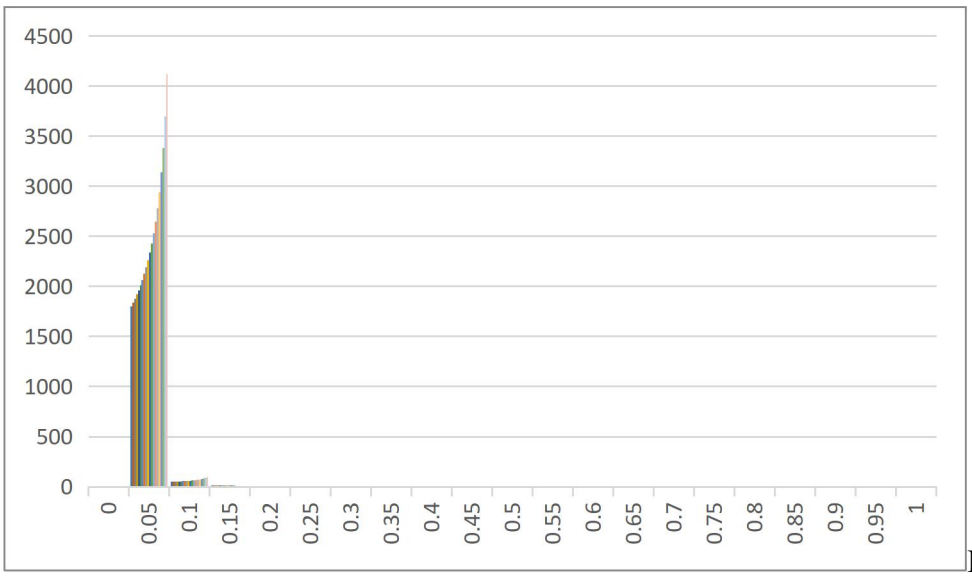
It=50



It=60



It=70



It=80

```
1.  !Computation of Action Ballance Equation
2.  program ebeq
3.   implicit none
4.      integer,parameter::dp_kind=kind(1.0d0),imx=20,jmx=20,nt=80
5.      integer i,j,it,it0
6.
   real(dp_kind)::ff,fd,sig,ep,w,wavelength,waveperiod,waterdepth,sigma
7.
   real(dp_kind)::df,u10,dt,c,k,k3,gamma,alpha,fet,fet_nodim,f,f_m,dx
8.
   real(dp_kind),parameter::grav=9.8,pi=3.1415926,frange=1.,inidepth=12
   .
9.      real(dp_kind),parameter::bottomslope=.01,complength=1000.
10.     real(dp_kind),dimension(0:imx+1,0:jmx)::waction,cg,sp,work
11.     character(20)::tmp
12. !JONSWAP spectrum
13.     print*,"input U10."
```

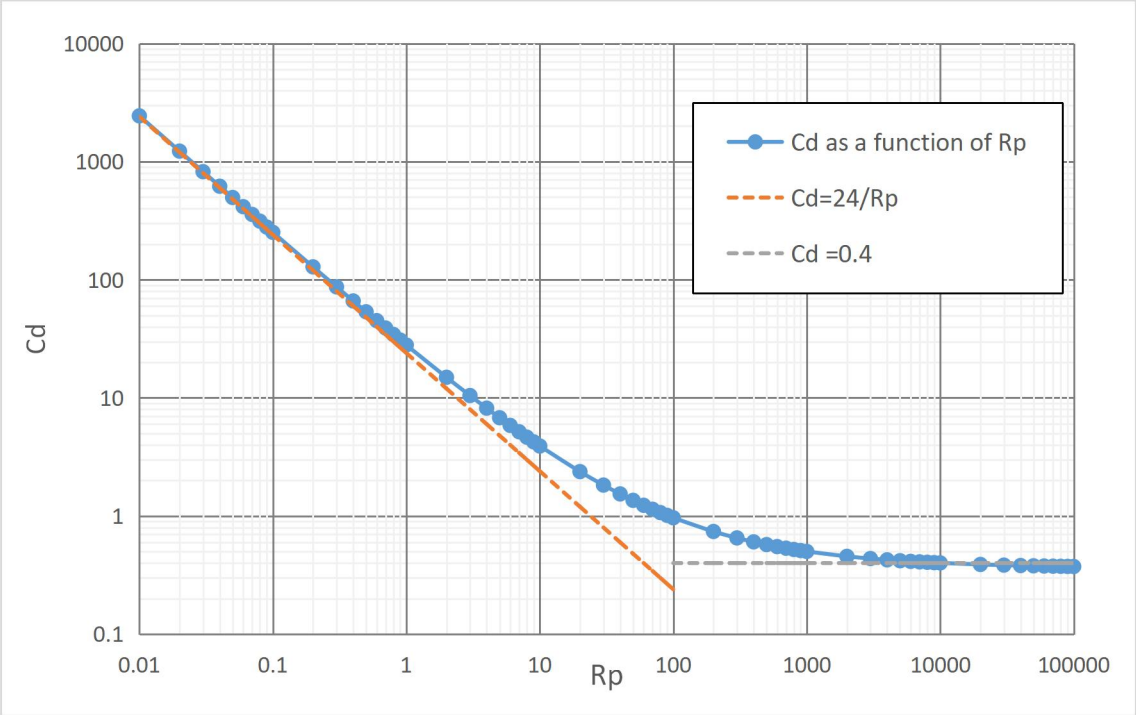
```
14.     read*,u10
15.     print*,"input Fetch."
16.     read*,fet
17.     df=frange/real(jmx)
18.     gamma=3.3d0
19.     alpha=0.076d0/fet**0.22d0
20.     fet_nodim=grav*fet/u10**2.
21.     f_m=3.5d0/fet_nodim**0.33d0
22.     do j=0,jmx  !frequency
23.     f=df*real(j)
24.     if (f<=f_m) then
25.         sigma=0.07d0
26.     else
27.         sigma=0.09d0
28.     endif
29.         k3=exp(-(f/f_m-1.)**2./2./sigma**2.)
30.
   sp(0,j)=alpha*grav**2./(2.*pi)**4./f**5.*exp(-1.25d0*f/f_m)*gamma**k
   3
31.         waction(0,j)=sp(0,j)/(2.*pi*f)
32.     enddo
33.     open(UNIT=1,FILE='sample00.data')
34.     write(1,FMT='(20e12.4)')(sp(0,j), j=1,jmx)
35.     write(1,*) 'U10=',u10,'fetch=',fet
36.     close(1)
37. !Group Velocity
38.     do i=0,imx
39.         dx=complength/real(imx)
40.         waterdepth=inidepth-bottomslope*dx*real(i)
41.         do j=1,jmx
42.             waveperiod=1./(df*real(j))
43.             call wlength(waveperiod,waterdepth,wavelength)
44.             write(*,*)waterdepth,waveperiod,wavelength
45.             c=wavelength/waveperiod
46.             k=2.*pi/wavelength
47.             cg(i,j)=c/2.*(1.+2.*k*waterdepth/sinh(2.*k*waterdepth))
48.         enddo
49.     enddo
50.
51.     it0=0
52.     do it=1,nt
53.         it0=it0+1
54.         dt=dx/sqrt(grav*inidepth)
55.         work=waction
56.         do i=1,imx
```

```
            do j=1,jmx
                waction(i,j)=waction(i,j)-dt/dx*(work(i,j)*cg(i,j)&
                &-work(i-1,j)*cg(i-1,j))
                sp(i,j)=waction(i,j)*2.*pi*df*real(j)
            enddo
        enddo
        if(it0==10)then
            write(tmp,"('sample',i2,'.data')")it
            open(UNIT=1,FILE=tmp)
            it0=0
            do i=1,imx
                write(1,FMT='(20e12.4)')(sp(i,j),j=1,jmx)
            enddo
            close(1)
        endif
    enddo
end program ebeq

subroutine wlength(waveperiod,waterdepth,wavelength)
    implicit none
    integer,parameter::dp_kind=kind(1.0d0)
    real(dp_kind)::ff,fd,sig,ep,w,wavelength,waveperiod,waterdepth
    real(dp_kind),parameter::grav=9.8,pi=3.1415926
    character::tmp
!Initial condition
    w=1.0; ep=9999.;
    sig=2.*pi/waveperiod
    do while(abs(ep)>.000001)
        ff=grav*w/sig/sig*tanh(w*waterdepth)-1.
        fd=grav/sig/sig*(tanh(w*waterdepth)+&
        &w*waterdepth/cosh(w*waterdepth)/cosh(w*waterdepth))
        ep=-ff/fd
        w=w+ep
    enddo
    wavelength=2.*pi/w
end subroutine wlength
```

Sediment Transport

Drag coefficient, Fall velocity and Particle Reynolds number

Plot CD as a function of Rp.



2. Discuss the behavior of CD in the ranges of sufficiently small and large Rp.
When Rp is small enough (Rp < 1), the relation between Rp and CD approximates to the formula CD = 24 / Rp. Because the following formula of Cd can be written as

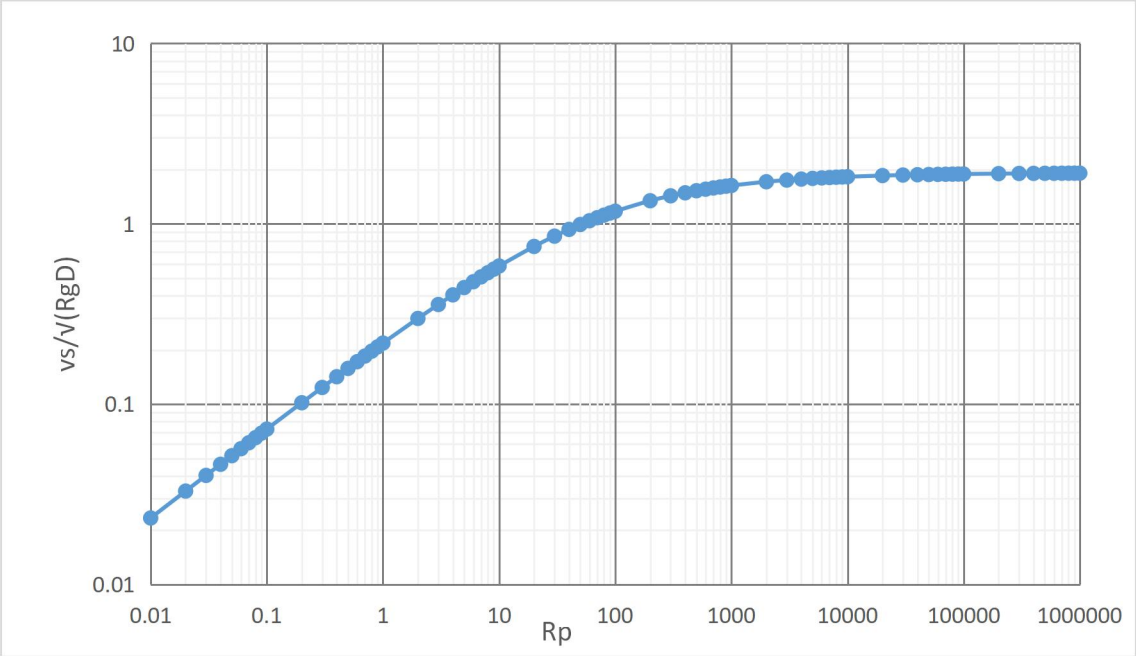
$$\mathrm{Cd} = \frac{24}{\mathrm{Rp}}\left(1 + 0.152\sqrt{\mathrm{Rp}} + 0.0151\mathrm{Rp}\right) \rightarrow \frac{24}{\mathrm{Rp}}$$

(Rp is small enough to exclude the latter terms)
When Rp is large enough, the value of Rp converges to about 0.4 because the Cd formula

$$\mathrm{Cd} = \frac{24}{\mathrm{Rp}}\left(1 + 0.152\sqrt{\mathrm{Rp}} + 0.0151\mathrm{Rp}\right) = \frac{24}{\mathrm{Rp}} + 24 \times 0.152\frac{1}{\sqrt{\mathrm{Rp}}} + 24 \times 0.0151$$

$$(\mathrm{Rp} \rightarrow \infty) \rightarrow 24 \times 0.0151 \sim 0.36$$

3. Plot the non-dimensional fall velocity vs/ √ (RgD) as a function of Rp

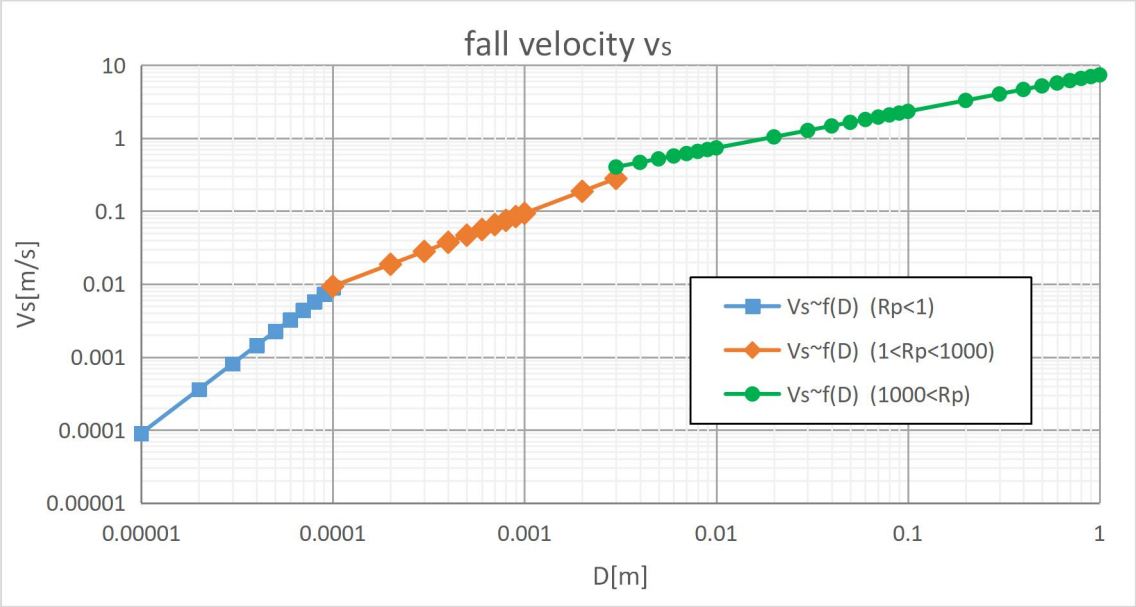


4. Discuss the behavior of the non-dimensional fall velocity in sufficiently large Rp.
When Rp is large enough, non-dimensional fall velocity converges to about 1.9

$$\mathrm{Cd} = \frac{24}{\mathrm{Rp}}\left(1 + 0.152\sqrt{\mathrm{Rp}} + 0.0151\mathrm{Rp}\right) \rightarrow (\mathrm{Rp} \rightarrow \infty) \rightarrow 24 \times 0.0151 \sim 0.36$$

$$\frac{\mathrm{Vs}}{\sqrt{\mathrm{RgD}}} = \sqrt{\frac{4}{3\mathrm{Cd}}} \rightarrow \sqrt{\frac{4}{3 \times 0.36}} \sim 1.9$$

5. Plot the dimensional fall velocity Vs as a function of the dimensional particle diameter D



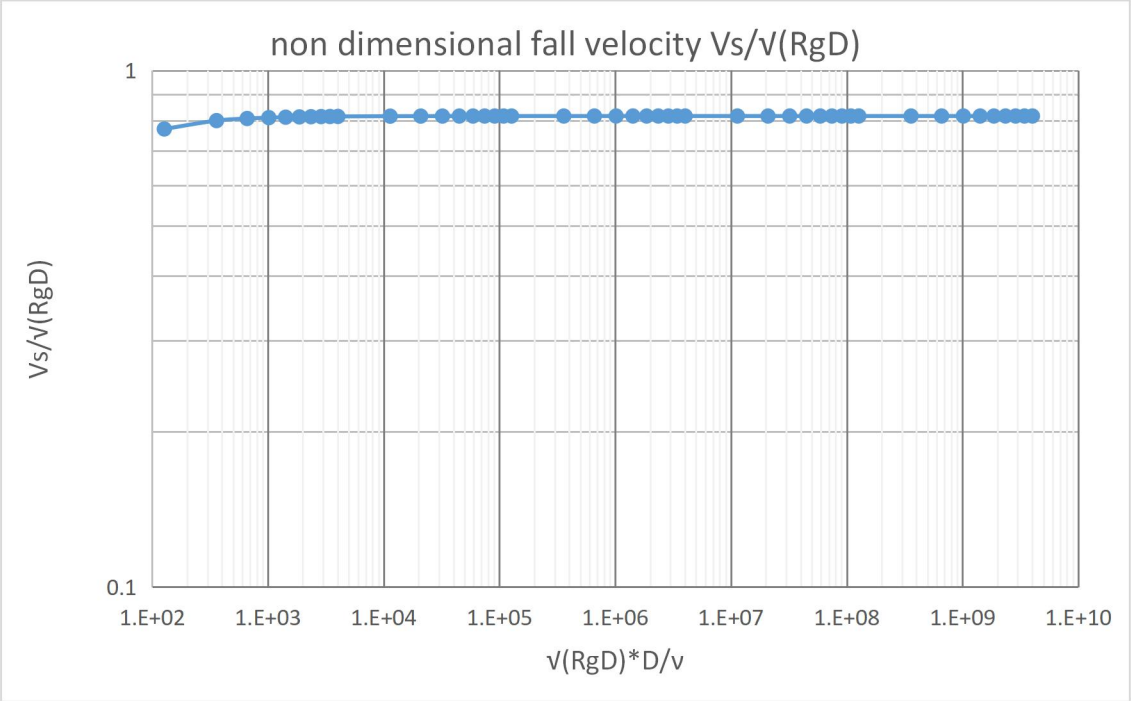
I separated the drag coefficient Cd into 3 types by the range of Rp as followings.

$$\mathrm{Cd} = \begin{cases} 24/\mathrm{Rp} & (\mathrm{Rp} < 1) \\ 24/\sqrt{\mathrm{Rp}} & (1 < \mathrm{Rp} < 1000) \\ 0.4 & (1000 < \mathrm{Rp}) \end{cases}$$

With these Cd, I calculated dimensional fall velocity Vs. And I calculated Rp again using this Vs to

check the value of Rp is in the ranges above.

6. Plot the non-dimensional fall velocity Vs/ √ (RgD) as a function of another Reynolds number √ (RgD)D/v.



I used following formula to estimate non-dimensional fall velocity.

$$\frac{Vs}{\sqrt{RgD}} = \sqrt{\frac{2}{3} + \frac{36\nu^2}{RgD^3}} - \sqrt{\frac{36\nu^2}{RgD^3}}$$

7. Discuss the behavior of the non-dimensional fall velocity in the range of sufficiently large another Reynolds number.

The non-dimensional fall velocity takes almost constant value (about 0.8) with any another Reynolds number, in opposition to the result of vs/ √ (RgD)~ Rp relation above.

Critical non-dimensional bed shear stress τ* c

Drag coefficient cD is expressed by

$$cD = \frac{24}{R_P}\left(1 + 0.152\sqrt{R_P} + 0.0151R_P\right)$$

$$\frac{u_f}{u_*} = F\left(\frac{u_* D}{\nu}\right)$$

$$\tau_c^* = \frac{4}{3}\frac{\mu}{c_D + \mu c_L}\frac{1}{F^2\left(\frac{u_* D}{\nu}\right)}$$

Addition to this equation, some values are assumed to be

μ =0.84, $c_L = 0.85c_D$.

So the equation of τ_c^* is rewritten to the equation of

$$\tau_c^* = \frac{4}{3}\frac{1}{c_D}\frac{0.84}{1.714}\frac{1}{F^2\left(\frac{u_* D}{\nu}\right)}$$

I calculate τ_c^* from the upper equation. I compiled the result in table1. And figure1 shows the result.

If we use particle Reynolds number, the equation varies.

$$\tau_c^* = \frac{4}{3}\frac{1}{c_D}\frac{0.84}{1.714}\frac{1}{F^2\left(R_{ep}\sqrt{\tau_c^*}\right)}$$

If $R_{ep}\sqrt{\tau_c^*} < 13.5$, $\tau_c^* = \frac{4}{R_{ep}}\sqrt{\frac{0.84}{3c_D 1.714}}$. And if $R_{ep}\sqrt{\tau_c^*} > 13.5$, $\tau_c^* = \frac{4}{3}\frac{1}{c_D}\frac{0.84}{1.714}\frac{1}{6.77^2}$

I calculate τ_c^* from the upper equation. I compiled the result in table2. τ_c^* that do not satisfy the range of each Rep become red. In table3, I compiled τ_c^* that satisfy the range of each Rep. Figure2 shows the result. In comparison with figure1, in the large area of Rp, the τ_c^* seems to be stable.

Table 1

cD(drag coefficient)	RP(shear Reynolds number)	F	τc*
240365.162	0.0001	0.00005	1087.416
24115.722	0.001	0.0005	108.3844
2436.842	0.01	0.005	10.72605
251.898	0.1	0.05	1.037628
28.010	1	0.5	0.093314
3.916	10	5	0.006675
0.967	100	6.77	0.014741
0.502	1000	6.77	0.028414
0.401	10000	6.77	0.035529
0.374	100000	6.77	0.038103
0.366	1000000	6.77	0.038946

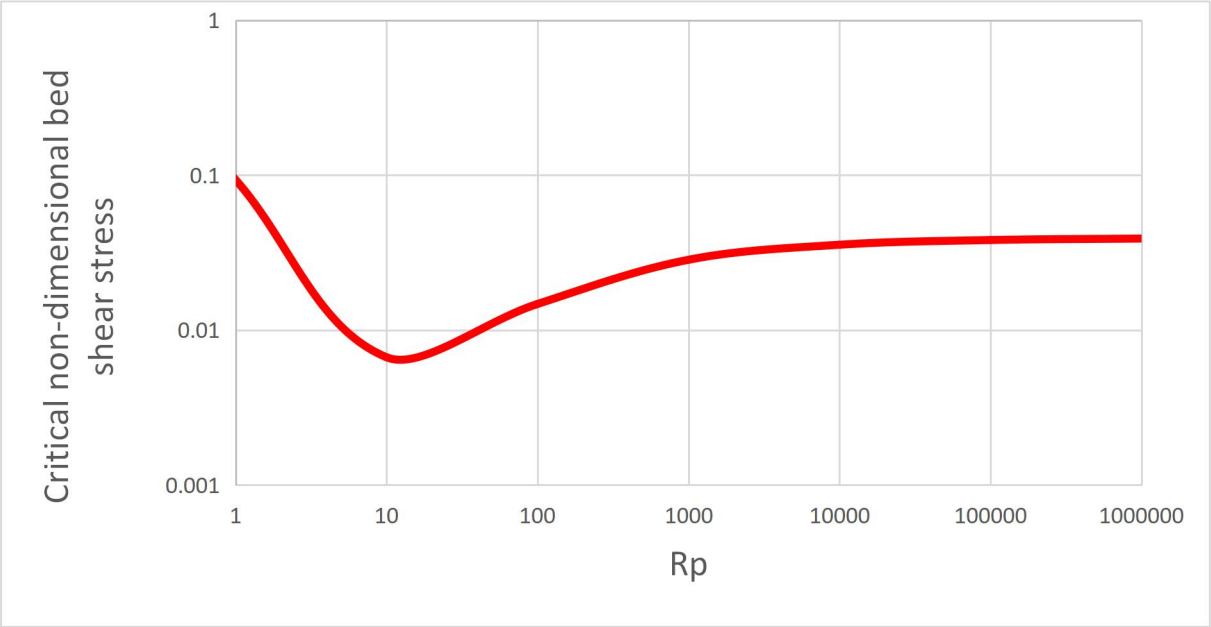


Figure 1

Table 2
F < 13.5

cD(drag coefficient)	R_P (shear Reynolds number)	Rep	τc*	F
240365.162	0.0001	0.0001	32.97598955	0.000574
24115.722	0.001	0.001	10.41078461	0.003227
2436.842	0.01	0.01	3.275064646	0.018097
251.898	0.1	0.1	1.018640416	0.100928
28.010	1	1	0.305473783	0.552697
3.916	10	10	0.081698168	2.858289
0.967	100	100	0.016439002	12.82147
0.502	1000	1000	0.002282368	47.77413
0.401	10000	10000	0.000255217	159.7552
0.374	100000	100000	2.64299E-05	514.1003
0.366	1000000	1000000	2.67209E-06	1634.651

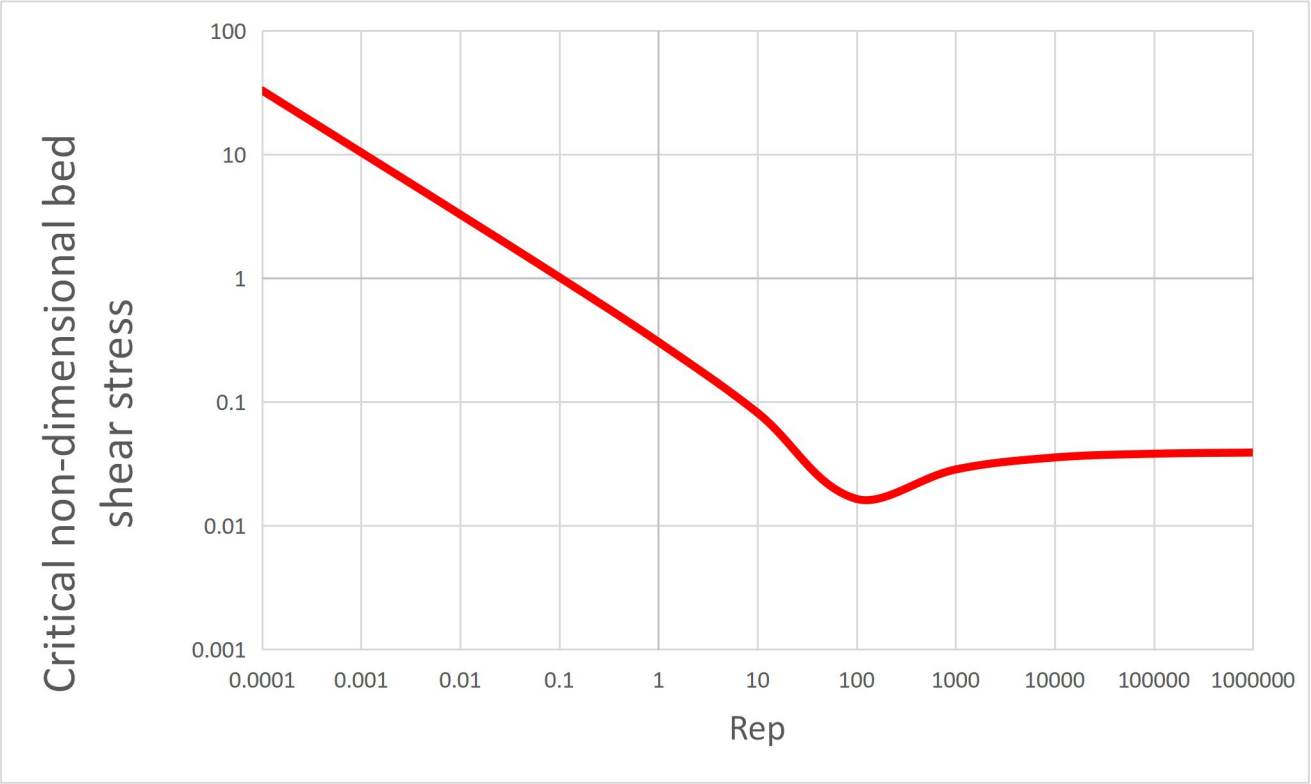
F >13.5

cD(drag coefficient)	R_P (shear Reynolds number)	Rep	τc*	F
240365.162	0.0001	0.0001	5.93142E-08	2.43545E-08
24115.722	0.001	0.001	5.91193E-07	7.68891E-07
2436.842	0.01	0.01	5.85063E-06	2.41881E-05

251.898	0.1	0.1	5.65984E-05	0.000752319
28.010	1	1	0.000508992	0.022560841
3.916	10	10	0.00364072	0.603383811
0.967	100	100	0.014740546	12.14106516
0.502	1000	1000	0.028414101	168.5648287
0.401	10000	10000	0.035528948	1884.912422
0.374	100000	100000	0.038102542	19519.87249
0.366	1000000	1000000	0.038946044	197347.5222

Table 3

Rep	τc*
0.0001	32.97599
0.001	10.41078
0.01	3.275065
0.1	1.01864
1	0.305474
10	0.081698
100	0.016439
1000	0.028414
10000	0.035529
100000	0.038103
1000000	0.038946



Mixing length and eddy viscosity

The logarithmic velocity distribution for open channel flow can be derived from the mixing length hypothesis for wall turbulence in the following manner.
The mixing length hypothesis is formulated by

$$v_T = l^2\left|\frac{d\bar{u}}{dz}\right|, l = kz \quad (1)$$

where v_T is the eddy viscosity, l is the mixing length, $\bar{u}$ is the Raynolds-averaged velocity, and z is the coordinate in the depth direction. The shear stress τ can be expressed by

$$\tau = \rho v_T \frac{d\bar{u}}{dz} \quad (2)$$

Substituting (1) into (2), we obtain

$$\tau = \rho l^2\left|\frac{d\bar{u}}{dz}\right|\frac{d\bar{u}}{dz} \quad (a)$$

Let us think of a sufficiently thin layer near the bed. We can assume that the shear stress is constant in the thin layer. Therefore, the shear stress is approximated to be identical to the bed shear stress, such that

$$\tau \approx \tau_b = \rho u_*^2 \quad (3)$$

where u∗ is the shear velocity defined by $\sqrt{\tau_b / \rho}$. With the use of the relation $d\overline{u}/dz > 0$, (1)–(3) reduce to

$$u_* = kz\frac{d\overline{u}}{dz} \quad (4)$$

Assuming the logarithmic velocity distribution, we have

$$\frac{u_*}{kz} = \frac{d\overline{u}}{dz} \quad (5)$$

Substituting (5) into (a), we obtain

$$\tau = \rho l^2\left(\frac{u_*}{\kappa z}\right)^2 \quad (6)$$

The shear stress

The above equation can be integrated with a boundary condition found in experimental observations. The result for a rough boundary is

$$\frac{\overline{u}}{u_*} = \frac{1}{k}\ln\frac{z}{k_s} + 8.5 = \frac{1}{k}\ln\frac{30z}{k_s} \quad (7)$$

Despite the assumption made in (3) that the shear stress is constant equal to the bed shear stress satisfied only in the vicinity of the bed, the logarithmic velocity distribution (6) performs very well in the whole region up to the water surface of open channel flow

If we think of the whole region up to the water surface, the shear stress τ actually has a linear distribution of the form

$$\tau = \tau_b\left(1 - \frac{z}{H}\right) \quad (8)$$

where H is the flow depth.

Substituting (8) into (6), we obtain

$$\tau_b\left(1 - \frac{z}{H}\right) = \rho l^2\left(\frac{u_*}{\kappa z}\right)^2 \quad (9)$$

Because $\tau_b = \rho u_*^{\,2}$, the above equation is rewritten in the form

$$\rho u_*^{\,2}\left(1 - \frac{z}{H}\right) = \rho l^2\left(\frac{u_*}{\kappa z}\right)^2 \quad (10)$$

Solving the above equation for l , we obtain

$$l = \kappa z\sqrt{1 - \frac{z}{H}} \quad (b)$$

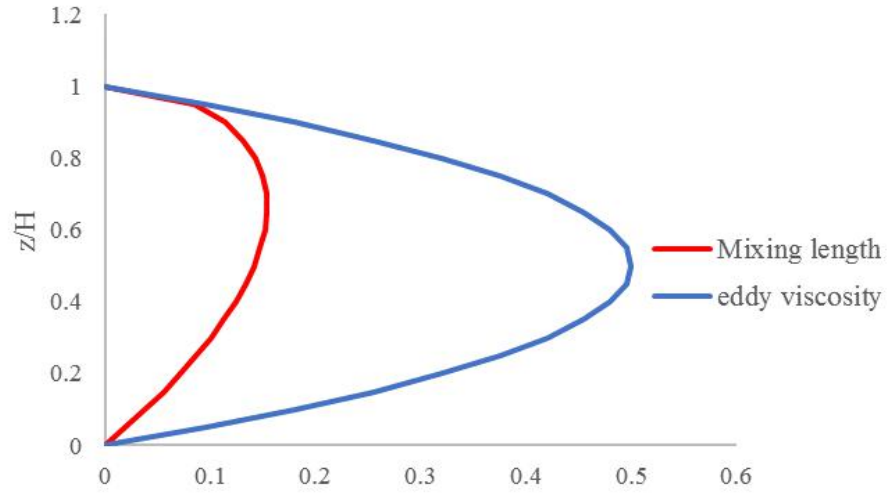
Substituting (b) and (5) into (1) , we obtain

$$v_T = \kappa u_* \left(1 - \frac{z}{H}\right)$$

Considering that the logarithmic velocity distribution holds in the whole region up to the water surface, back calculate the eddy viscosity v_T from (2), (6) and (7). Then we can plot the v_T and l as follows:

κ	H	u*
0.4	2	1

z	v_T (eddy viscosity)	l(mixing length)	z/H	l/H	$v_T/(u_*\kappa)$
0	0	0	0	0	0
0.1	0.038	0.038987177	0.05	0.019494	0.095
0.2	0.072	0.075894664	0.1	0.037947	0.18
0.3	0.102	0.110634533	0.15	0.055317	0.255
0.4	0.128	0.143108351	0.2	0.071554	0.32
0.5	0.15	0.173205081	0.25	0.086603	0.375
0.6	0.168	0.200798406	0.3	0.100399	0.42
0.7	0.182	0.225743217	0.35	0.112872	0.455
0.8	0.192	0.247870934	0.4	0.123935	0.48
0.9	0.198	0.266983146	0.45	0.133492	0.495
1	0.2	0.282842712	0.5	0.141421	0.5
1.1	0.198	0.295160973	0.55	0.14758	0.495
1.2	0.192	0.303578655	0.6	0.151789	0.48
1.3	0.182	0.307636149	0.65	0.153818	0.455
1.4	0.168	0.306724632	0.7	0.153362	0.42
1.5	0.15	0.3	0.75	0.15	0.375
1.6	0.128	0.286216701	0.8	0.143108	0.32
1.7	0.102	0.263362868	0.85	0.131681	0.255
1.8	0.072	0.227683992	0.9	0.113842	0.18
1.9	0.038	0.169941166	0.95	0.084971	0.095
2	0	0	1	0	0



$$v_T/(u_*\kappa), l/H$$

Suspended sediment concentration

Assuming that the eddy diffusivity Dd takes the same value as the eddy viscosity v_T , derived the Rousean distribution of the suspended sediment concentration by solving the dispersive-diffusive equation for the normal flow condition. Then the dispersive-diffusion equation of suspended sediment concentration is

$$v_s\bar{c} - D_d\frac{d\bar{c}}{dz} = 0 \quad (11)$$

Substituting (b) into (1), we obtain

$$D_d = v_T = \kappa^2 z^2(1-\frac{z}{H})\left|\frac{d\bar{u}}{dz}\right| = \kappa^2 z^2(1-\frac{z}{H})\frac{u_*}{\kappa z} = \kappa u_* z\left(1-\frac{z}{H}\right)$$

Substituting the above equation into (11), we obtain

$$v_s\bar{c} - \kappa u_* z\left(1-\frac{z}{H}\right)\frac{d\bar{c}}{dz} = 0$$

Modifying the above equation, we obtain

$$\frac{d\bar{c}}{\bar{c}} = -\frac{v_s}{\kappa u_*}\left(\frac{1}{z}+\frac{1}{H-Z}\right)dz$$

Integrate the above equation, we obtain

$$\ln \bar{c} = -\frac{v_s}{\kappa u_*}\left(\ln \bar{z} + \ln(H-z)\right) + C$$

$$= -\frac{v_s}{\kappa u_*}\left(\ln \frac{\bar{z}}{H-z}\right) + C \qquad \text{(d)}$$

Applying the following boundary condition:

$c = c_a$ and z=a

We obtain

$$\ln c_a = -\frac{v_s}{\kappa u_*}\left(\ln \frac{a}{H-a}\right) + C$$

Where C is an integral constant. Solving the above equation for C, we obtain

$$C = \ln c_a + \frac{v_s}{\kappa u_*}\left(\ln \frac{a}{H-a}\right)$$

Substituting the above equation into (d), we obtain

$$\ln \bar{c} - \ln c_a = -\frac{v_s}{\kappa u_*}\left(\ln \frac{\bar{z}(H-a)}{a(H-z)}\right)$$

Modifying the above equation, we obtain

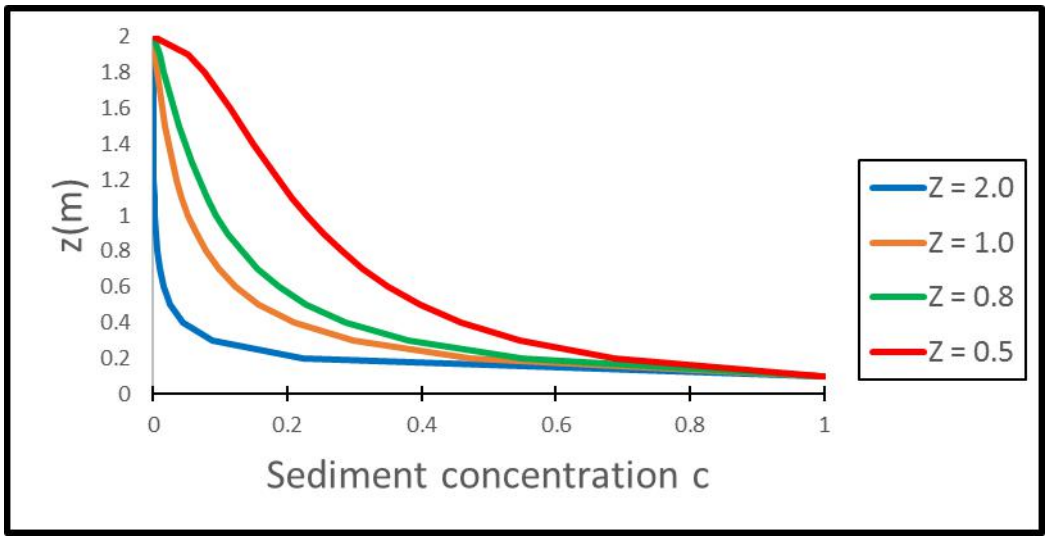
$$\bar{c} = c_a\left(\frac{a(H-z)}{z(H-a)}\right)^Z \qquad \text{(e)}$$

where $Z = v_s/(\kappa u_*)$. Equation (e) is plotted in the following figure

c_a	H	a
1	2	0.1

Z=2		Z=1		Z=0.8		Z=0.5	
z	c	z	c	z	c	z	c
0	0	0	0	0	0	0	0
0.1	1	0.1	1	0.1	1	0.1	1
0.2	0.224377	0.2	0.473684	0.2	0.550036	0.2	0.688247
0.3	0.08895	0.3	0.298246	0.3	0.379891	0.3	0.546119
0.4	0.044321	0.4	0.210526	0.4	0.287505	0.4	0.458831
0.5	0.024931	0.5	0.157895	0.5	0.228399	0.5	0.39736
0.6	0.015082	0.6	0.122807	0.6	0.186801	0.6	0.350438
0.7	0.009554	0.7	0.097744	0.7	0.155623	0.7	0.312641
0.8	0.006233	0.8	0.078947	0.8	0.131181	0.8	0.280976

0.9	0.004138	0.9	0.064327	0.9	0.111357	0.9	0.253629
1	0.00277	1	0.052632	1	0.094841	1	0.229416
1.1	0.001854	1.1	0.043062	1.1	0.080775	1.1	0.207514
1.2	0.001231	1.2	0.035088	1.2	0.068568	1.2	0.187317
1.3	0.000803	1.3	0.02834	1.3	0.057799	1.3	0.168345
1.4	0.000509	1.4	0.022556	1.4	0.048152	1.4	0.150188
1.5	0.000308	1.5	0.017544	1.5	0.039382	1.5	0.132453
1.6	0.000173	1.6	0.013158	1.6	0.031286	1.6	0.114708
1.7	8.63E-05	1.7	0.009288	1.7	0.023677	1.7	0.096374
1.8	3.42E-05	1.8	0.005848	1.8	0.016353	1.8	0.076472
1.9	7.67E-06	1.9	0.00277	1.9	0.008995	1.9	0.052632
2	0	2	0	2	0	2	0



The diffusion coefficient Dd is depth-averaged to be

$$\overline{D}_d = \frac{1}{H}\int_0^H D_d dz$$

$$= \kappa u_* \int_0^H z\left(1-\frac{z}{H}\right)dz$$

$$= \kappa u_* \left[\frac{z^2}{2} - \frac{z^3}{3H}\right]_0^H \quad \text{(e)}$$

$$= \frac{1}{6}\kappa u_* H$$

Modifying the dispersive-diffusion equation of suspended sediment concentration

$$v_s \overline{c} - D_d \frac{d\overline{c}}{dz} = 0$$

We obtain

$$\frac{d\bar{c}}{\bar{c}} = -\frac{v_s}{\overline{D}_d} dz$$

Integrating the above equation, we obtain

$$\ln \bar{c} = -\frac{v_s}{\overline{D}_d} z + C'$$

Solving the above equation, we obtain

$$C' = \ln \bar{c} + \frac{v_s}{\overline{D}_d} a$$

Thus, we have

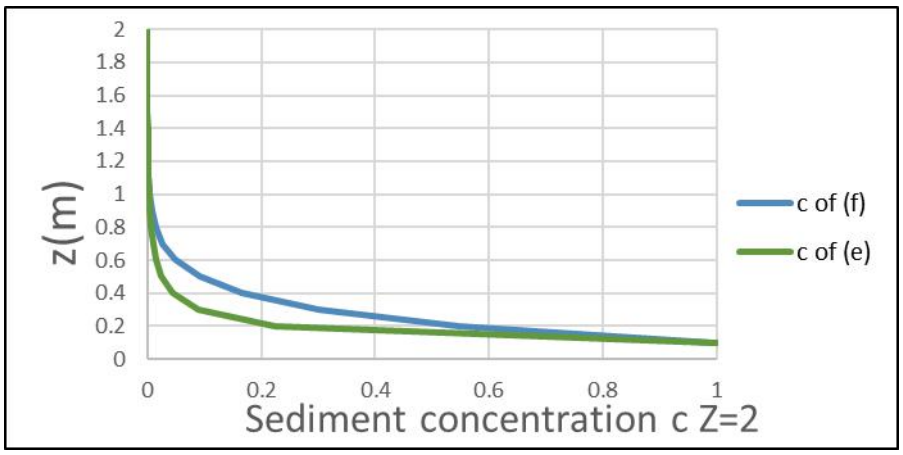
$$\bar{c} = c_a \exp\left[-\frac{v_s}{\overline{D}_d} (z - a) \right]$$

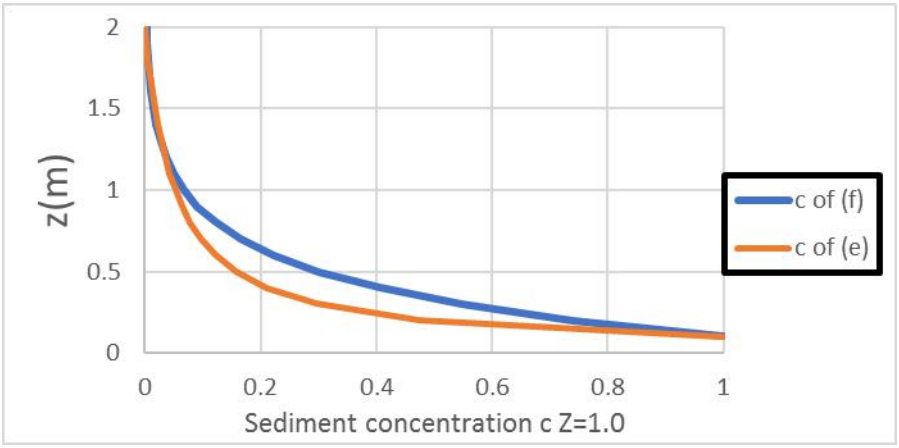
Substituting (e),

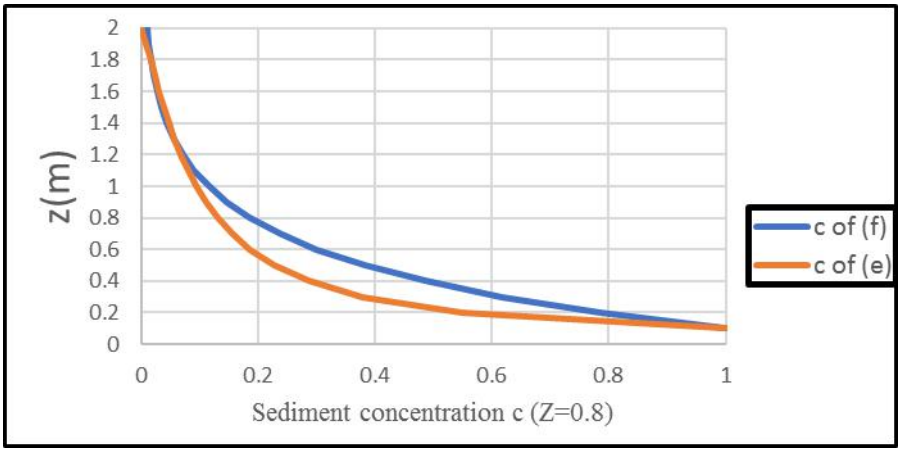
$$\bar{c} = c_a \exp\left[-\frac{6 v_s}{\kappa u_* H} (z - a) \right]$$

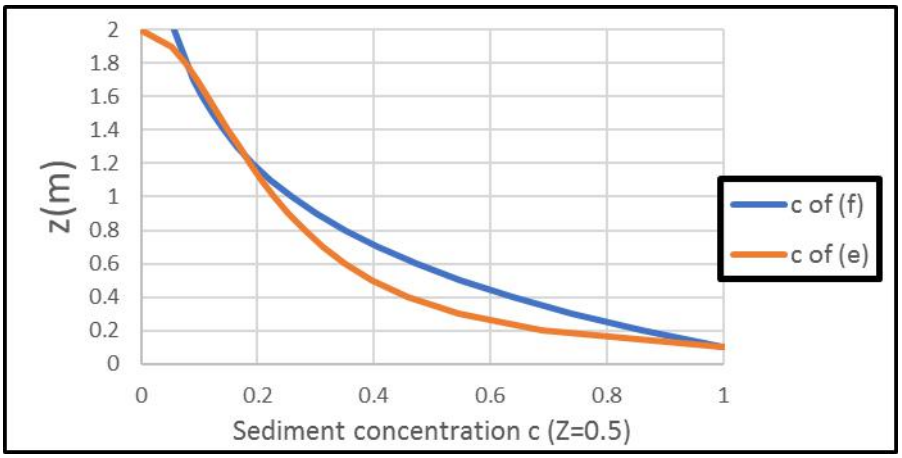
$$= c_a \exp\left[\frac{6Z}{H} (z - a) \right] \qquad \text{(f)}$$

c_a	H	a
1	2	0.1









Bedforms Mechanics

Discuss the heuristic analysis for the following cases:

1. Case D: Φ slightly greater than 0, $q_{star} > 0$.

2. Case E: Φ slightly less than 0, $q_{\mathrm{star}} < 0$.

3. Case F: Φ slightly greater than 0, $q_{\mathrm{star}} < 0$.

First, about the heuristic analysis:

Consider flow and bedload transport in the s-direction in an erodible-bed channel.

Let the bed have a rhythmic cosine shape at some time, as that time = 0, then bed elevation can be represented as:

$$\eta = \eta_0 + \eta_{star}\, cos\,(ks) \tag{1}$$

where η_{star} is a positive real number satisfying the condition $\eta_{star} \ll H$, where H is flow depth. where η_0 should be a constant for flat initial bed elevation.

When t = 0, **the Exner equation** of sediments and bed of channel in 1-D(s-direction):

$$\left.\frac{\partial \eta}{\partial t}\right|_{t=0} = -\frac{1}{1-\lambda_p}\left.\frac{\partial q}{\partial s}\right|_{t=0} \tag{2}$$

This 1-D Exner equation of equation is based on conservation of mass between sediments in the bed of channel between sediment is being transported.

Where λ_p equals the bed porosity, a typical value for natural systems range from 0.25 to 0.55, and for spherical grains is 0.36.

Then, the sediment transport fluctuation could be represented as:

$$q = q_0 + q_{star} \cdot cos\,(ks - \phi) \tag{3}$$

Where q_{star} should be a real number satisfying the condition $|q_{star}| \ll |q_0|$.

From (2), rearranging this equation:

$$(1-\lambda_p)\frac{\partial \eta}{\partial t} + \frac{\partial q}{\partial s} = 0 \tag{4}$$

where k in (1) and (2) is defined by $\lambda = 2\pi/k$, where λ is wave length of

fluctuation.

So $k = 2\pi/\lambda$ is the angular wavenumber of fluctuation.

then, using the equations (3) and (4), start the analysis for Case D, E and F.

1. Case D: Φ slightly greater than 0, and $q_{\text{star}} > 0$.

   Because we all know the ϕ could be a constant in our analysis (because it just a phase in the initial condition), so for convenient, there assume a consist value for Φ slightly greater than 0, there assumed $\Phi = \pi/4$.

   From (1):

$$\eta = \eta_0 + \eta_{star} \cdot cos\left(\frac{2\pi}{\lambda}s\right) \quad \textbf{(5)}$$

   Partial differential in s-direction:

$$\frac{\partial \eta}{\partial s} = \eta_{star} \cdot \left(-\frac{2\pi}{\lambda}\right) sin\left(\frac{2\pi}{\lambda}s\right) \quad \textbf{(6)}$$

   From (3):

$$q = q_0 + q_{star} \cdot cos\left(\frac{2\pi}{\lambda}s - \frac{\pi}{4}\right) \quad \textbf{(7)}$$

   Partial differential in s-direction:

$$\frac{\partial q}{\partial s} = q_{star} \cdot \left(-\frac{2\pi}{\lambda}\right) sin\left(\frac{2\pi}{\lambda}s - \frac{\pi}{4}\right) \quad \textbf{(8)}$$

   in this case both η_{star} and q_{star} are positive. As η_{star}=0.05 $q_{star} = 0.05, \lambda = 0.1$. **(9)**

$$\frac{\partial \eta}{\partial t} = \frac{\partial \eta}{\partial s}\frac{\partial s}{\partial t} = \frac{\partial s}{\partial t}\eta_{star} \cdot \left(-\frac{2\pi}{\lambda}\right) sin\left(\frac{2\pi}{\lambda}s\right)$$

   Finally, for plot conveniently, as $\eta_0 = 0$ and $q_0 = 0.3$, $\frac{\partial s}{\partial t} = 0.02$:

(10)

$$\frac{\partial \eta}{\partial t} = \frac{\partial \eta}{\partial s}\frac{\partial s}{\partial t} = -\,0.02\eta_{star} \cdot \frac{2\pi}{\lambda} sin\left(\frac{2\pi}{\lambda}s\right)$$

then considering the condition of η in $t_0 = 0$ and $t_1 = 0+\Delta t$, and as $\Delta t = 0.8s$.

$$\eta(t_1) = \eta(t_0) + \Delta t\frac{\partial \eta}{\partial t} \tag{11}$$

Then substituting (5) and (9) in to (11):

$$\eta(t_1) = \eta + \Delta t\frac{\partial s}{\partial t}\eta_{star} \cdot (-\frac{2\pi}{\lambda})\, sin\left(\frac{2\pi}{\lambda}s\right) \tag{12}$$

For plot it conveniently, first substituting all assumed constant in equation (12), note there $\eta(t_0) = \eta(t = 0) = the\ \eta\ in\ (5)$:

$$\eta(t_1) = \eta + 0.8 \times 0.02 \times 0.05 \cdot (-\frac{2\pi}{0.1})\, sin\left(\frac{2\pi}{0.1}s\right) \tag{13}$$

and from equation (4):

$$\frac{\partial q}{\partial s} = (1-\lambda_p)\eta_{star} \cdot (\frac{2\pi}{\lambda})\, sin\left(\frac{2\pi}{\lambda}s\right) \tag{14}$$

then integer this equation in s-direction:

$$q = q_0 + q_{star} \cdot \cos\left(\frac{2\pi}{\lambda}s - \frac{\pi}{4}\right) \tag{15}$$

$$q = -\,(1-\lambda_p)\eta_{star} \cdot \cos\left(\frac{2\pi}{\lambda}s\right) + const. \tag{16}$$

There const. in (16) is q_0, we obtain another form of the Exner equation we need:

$$q = q_0 - (1-\lambda_p)\eta_{star} \cdot \cos\left(\frac{2\pi}{\lambda}s\right) = q_0 + q_{star} \cdot \cos\left(\frac{2\pi}{\lambda}s - \frac{\pi}{4}\right)$$

Just rearranging:

(17)

$$-(1-\lambda_{\boldsymbol{p}})\eta_{star}\cdot\cos\left(\frac{2\pi}{\lambda}s\right)=q_{star}\cdot\cos\left(\frac{2\pi}{\lambda}s-\frac{\pi}{4}\right)$$

So using (17), and we assumed $\eta_{star}\ and\ q_{star}\ and\ \lambda$, can obtain a λ_p fit them, there just explain how we did not assume the λ_p, in fact it been included in other assumptions.

There are all the constants was assumed:

$$\begin{cases}\lambda(wave\ length)=0.1\ ,\quad \Phi=\pi/4\\ q_{star}=\pm\ 0.05\ (in\ case\ D,F\ is\ positive)\\ \eta_{star}=0.05\ (be\ assumed\ positive)\\ \Delta t=0.8(time\ step)\\ (\partial s/\partial t)=0.02(wave\ velocity)\end{cases}\quad\textbf{(18)}$$

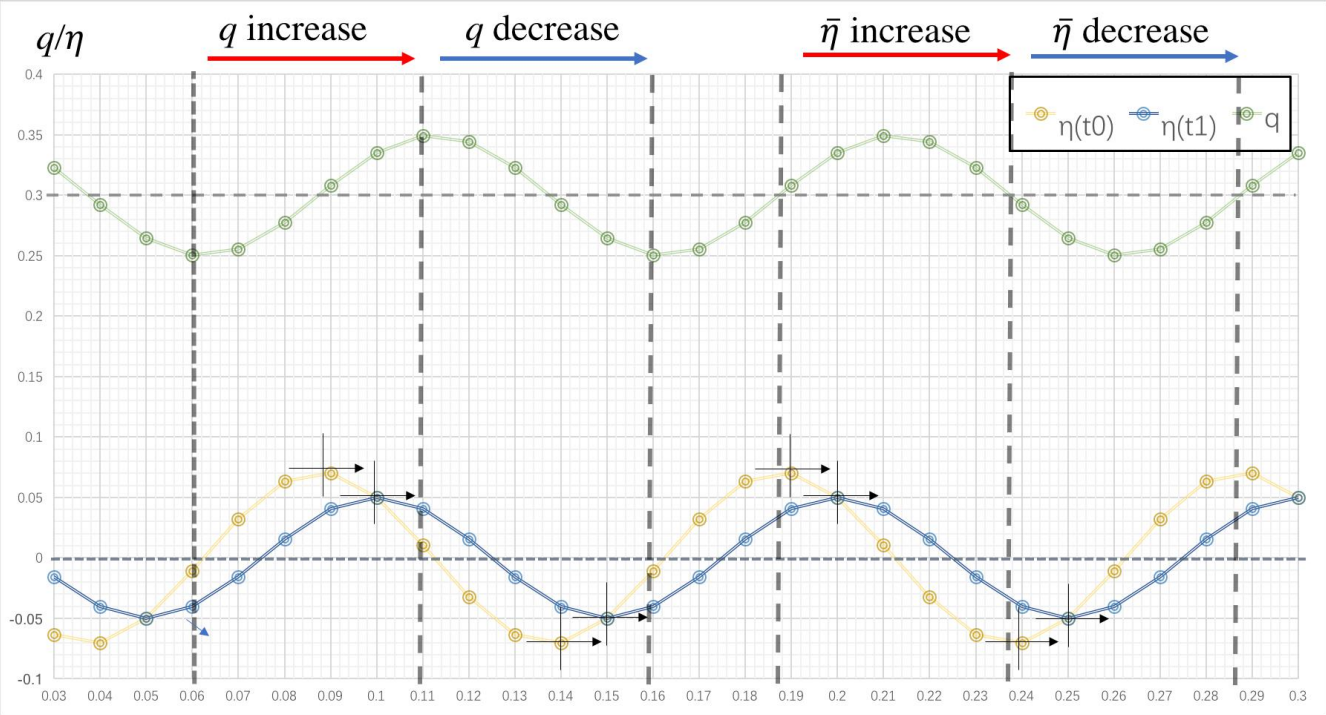
Then plot (5), (13) and (16) as constants were assumed in (18) like Fig.1, this should be one of conditions of case D, so about the case D:

There with $\eta\ changed$ from $\eta(t=0)$ the yellow line to $\eta\ (t=\Delta t)$ the blue line:

1.The migrate direction of bars slightly in the wave crest and the wave trough is in downstream. (there has defined the s increasing direction is the downstream direction.)

2.Amplitude of the bedform $(\eta-\eta_0)$ slightly decrease with deformation process.

Fig1: Case D Φ slightly greater than 0, $q_{star}>0$.


q/η
q increase
q decrease
η̄ increase
η̄ decrease
η(t0)
η(t1)
q
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
-0.05
-0.1
0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1 0.11 0.12 0.13 0.14 0.15 0.16 0.17 0.18 0.19 0.2 0.21 0.22 0.23 0.24 0.25 0.26 0.27 0.28 0.29 0.3

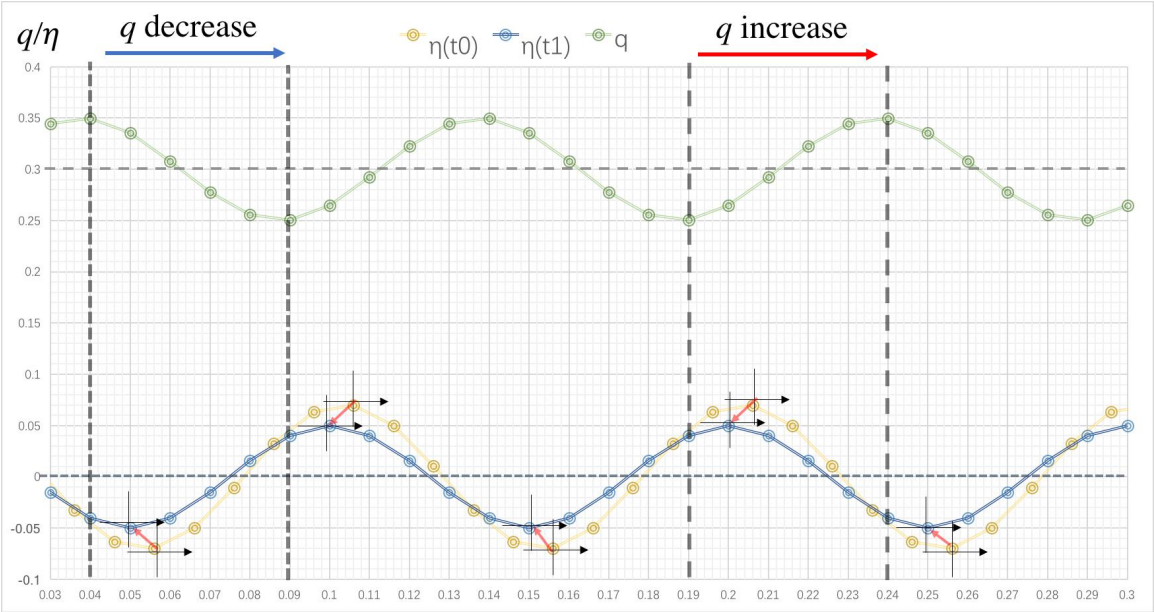
S

2. Case E: Φ slightly less than 0, $q_{star} < 0$.

Just like Case D: plot it by (5), (13) and (16) as constants were assumed in (18) like Fig.2, this should be one of conditions of case E, so about the case E:

(note that in this case the $q_{star} = -0.05$)

Fig2: Case E: Φ slightly less than 0, $q_{star} < 0$.



From $\eta(t_0)$ the yellow line to $\eta(t_1)$ the blue line:

1.It is slightly in Fig.2 that migrate direction of bars were marked by red arrows, all of them migrated to upstream direction.

2.It is slightly the amplitude of the bedform $(\eta - \eta_0)$ is decrease with deformation process.

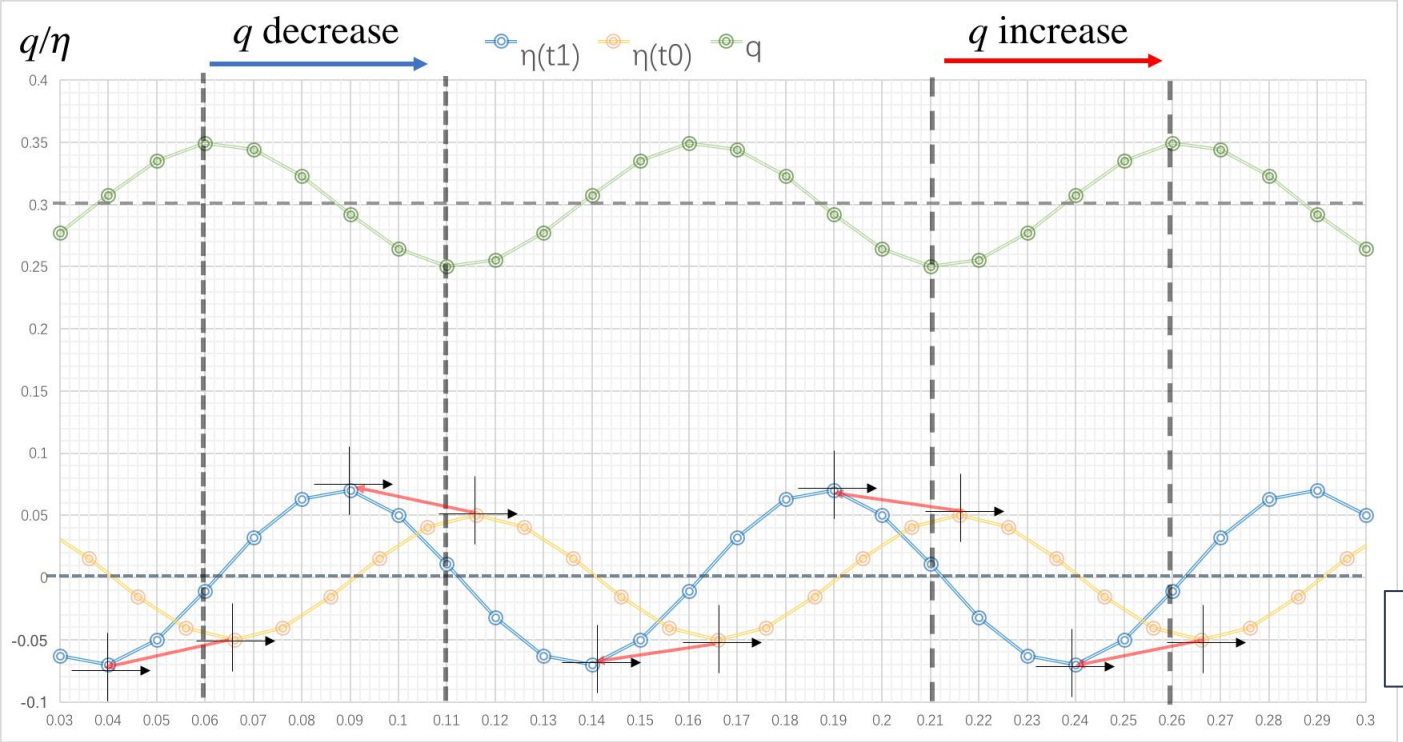
s

3.Case F: Φ slightly greater than 0, $q_{star} < 0$.

Just like Case D and E: plot it by (5), (13) and (16) as constants were assumed in (18) like Fig.3, this is one of conditions for case F, so about the case F:

(note that in this case the $q_{star} = 0.05$)

Fig3: Case F: Φ slightly greater than 0, $q_{star} < 0$



From $\eta(t_0)$ the yellow line to $\eta(t_1)$ the blue line:

1.It is slightly in Fig.3 that migrate direction of bars were marked by red arrows, all of them migrated to upstream direction.

2.But it is slightly the amplitude of the bedform ($\eta - \eta_0$) is increase with deformation process different from Case D and Case E.

Finally, the results of three conditions (them were plotted in Fig.1,2 and 3.):

Case	*Conditions*	*Migrate direction*	*The bedform amplitude*
1.D	Φ slightly greater than 0, $q_{star} > 0$	to downstream	Decrease with process
2.E	Φ slightly less than 0, $q_{star} < 0$	to Upstream	Decrease with process
3.F	Φ slightly greater than 0, $q_{star} < 0$	to Upstream	Increase with process

Derive the non-dimensional governing equations for bedrock channel

For solve ***Question 1***: derive the non-dimensional governing equations for bedrock channel:
The erosion rate is assumed to be a power function of flow velocity:

$$\widetilde{E} = \alpha \tilde{\tau}_b^{\gamma} \tag{1.1}$$

Where $\widetilde{E}$ is the erosion rate, $\tilde{\tau}_b$ is the bed shear stress, α and γ are empirical constants.

And the time variation of the bed elevation can be obtained by (1.1)

$$\frac{\partial \widetilde{Z}}{\partial \tilde{t}} = -\widetilde{E} = -\alpha \tilde{\tau}_b^{\gamma} \tag{1.2}$$

Where $\widetilde{Z}$ is the bed elevation, and $\tilde{t}$ is time. Then written them with the shallow water equations written as:

$$\begin{cases} \dfrac{\partial \widetilde{U}}{\partial \tilde{t}} + \widetilde{U}\dfrac{\partial \widetilde{U}}{\partial \tilde{x}} = -g\cos\theta\dfrac{\partial \widetilde{H}}{\partial \tilde{x}} - g\cos\theta\dfrac{\partial \widetilde{Z}}{\partial \tilde{x}} - g\sin\theta - \dfrac{\tilde{\tau}_b}{\rho \widetilde{H}} \ (Momentum) & \textbf{(1.3)} \\ \dfrac{\partial \widetilde{H}}{\partial \tilde{t}} + \dfrac{\partial \widetilde{U}\widetilde{H}}{\partial \tilde{x}} = 0 \ (Continunity) & \textbf{(1.4)} \\ \dfrac{\partial \widetilde{Z}}{\partial \tilde{t}} + \alpha \tilde{\tau}_b^{\gamma} = 0 \ (Erosion\ on\ bedrock\ bed) & \textbf{(1.5)} \end{cases}$$

Where

$$\tilde{\tau}_b = \rho C_f \widetilde{U}^2 \tag{1.6}$$

And do normalization with: For distinct them i still use * to express non-dimensional quantities.

$$\tilde{U} = \tilde{U}_n U^*,\ (\tilde{H},\tilde{Z}) = \tilde{H}_n(H^*,Z^*),\ \tilde{x} = \frac{\tilde{H}_n}{\sin\theta}x^*,\ \tilde{t} = \frac{\tilde{H}_n}{\tilde{E}_n}t^* \text{ and } U^*H^* = 1$$

And about the order of non-dimensional quantities:

$$U^* = 1 + u',\ H^* = 1 + h',\ Z^* = 1 + z',\ \tau^* = 1 + \tau'$$

in normal flow condition:

$$\tilde{\tau}_{bn} = \rho C_f \tilde{U}_n^2 = \rho g \tilde{H}_n S,\ C_f = \frac{\tilde{H}_n}{\tilde{U}_n^2} gS$$

Where :

1. $\tilde{U}_n$: The normal flow velocity.
2. $\tilde{H}_n$: The normal flow depth.
3. $\tilde{E}_n$: The erosion rate in the normal flow condition, and difined by:

$$\tilde{E}_n = \alpha \tilde{\tau}_{bn}^{\gamma}$$

First from (1.3) the momentum equation, first divide it into 6 parts:

$$\underset{\text{Term1}}{\frac{\partial \tilde{U}}{\partial \tilde{t}}} + \underset{\text{Term2}}{\tilde{U}\frac{\partial \tilde{U}}{\partial \tilde{x}}} = \underset{\text{Term3}}{-g\cos\theta\frac{\partial \tilde{H}}{\partial \tilde{x}}} \underset{\text{Term4}}{-g\cos\theta\frac{\partial \tilde{Z}}{\partial \tilde{x}}} \underset{\text{term5}}{-g\sin\theta} \underset{\text{term6}}{-\frac{\tilde{\tau}_b}{\rho\tilde{H}}}$$

Then transform it term by term, for **term 1:**

$$\frac{\partial \tilde{U}}{\partial \tilde{t}} = \frac{\partial \tilde{U}_n U^*}{\partial \frac{\tilde{H}_n}{\tilde{E}_n}t^*} = \tilde{E}_n \frac{\tilde{U}_n}{\tilde{H}_n}\frac{\partial U^*}{\partial t^*}$$

Term 2:

$$\tilde{U}\frac{\partial \tilde{U}}{\partial \tilde{x}} = \tilde{U}_n U^* \frac{\partial \tilde{U}_n U^*}{\partial \tilde{H}_n x^*}\cdot\sin\theta = \frac{\tilde{U}_n^2}{\tilde{H}_n}\cdot U^*\frac{\partial U^*}{\partial x^*}\cdot\sin\theta$$

Term 3:

$$-g\cos\theta\frac{\partial \tilde{H}}{\partial \tilde{x}} = -g\cos\theta\sin\theta\cdot\frac{\tilde{H}_n}{\tilde{H}_n}\frac{\partial H^*}{\partial x^*} = -g\sin\theta\cos\theta\frac{\partial H^*}{\partial x^*}$$

Term 4:

There set bed elevation $\tilde{Z}_0 = \tilde{Z}_i - S\cdot x$, where $\tilde{Z}_i$ is a constant elevation initial, and $\tilde{Z}_0$ is a the point we choose close to $\tilde{Z}_i$ from downstream , just like $\tilde{Z}(x)\ and\ \tilde{Z}(x+dx)$:

$$\tilde{Z}_{deviation} = \tilde{Z} - \tilde{Z}_0 = \tilde{H}_n Z^*$$

So there:

$$\tilde{Z} = \tilde{H}_n Z^* + \tilde{Z}_0 = \tilde{H}_n Z^* + \tilde{Z}_i - S\cdot x \text{ , where S} = \tan\theta$$

$$\tilde{Z} = \tilde{H}_n Z^* + \tilde{Z}_i - S\cdot\tilde{x}$$

$$\frac{\partial \tilde{Z}}{\partial \tilde{x}} = \sin\theta\cdot\frac{\tilde{H}_n}{\tilde{H}_n}\frac{\partial Z^*}{\partial x^*} - S = \sin\theta\cdot\frac{\tilde{H}_n}{\tilde{H}_n}\frac{\partial Z^*}{\partial x^*} - \tan\theta$$

$$-g\cos\theta\frac{\partial \tilde{Z}}{\partial \tilde{x}} = -g\cos\theta\left(\sin\theta\frac{\partial Z^*}{\partial x^*} - \frac{\sin\theta}{\cos\theta}\right) = -g\sin\theta\cos\theta\frac{\partial Z^*}{\partial x^*} + g\sin\theta$$

Note that term 5 in this case is a constant: $-g\sin\theta$ this should was a part of term 4 this equation from :

set $\widetilde{Z}_0 = \widetilde{Z}_i + S\cdot x$ and $\widetilde{Z} = \widetilde{H}_n Z^* + \widetilde{Z}_0 = \widetilde{H}_n Z^* + S\cdot x$

then:

$$\frac{\partial \widetilde{Z}}{\partial \tilde{x}} = \sin\theta\cdot\frac{\widetilde{H}_n}{\widetilde{H}_n}\frac{\partial Z^*}{\partial x^*} + S$$

$$-g\cos\theta\frac{\partial \widetilde{Z}}{\partial \tilde{x}} = -g\sin\theta\cos\theta\frac{\partial Z^*}{\partial x^*} - g\sin\theta$$

By it confirmed in this case bed evalution are increase with x-direction, gravity term and drag term have same a direction. Or, there θ in this condition are in a range ($\pi < \theta < 0$).

Term 6:

$$-\frac{\tilde{\tau}_b}{\rho\widetilde{H}} = -\frac{\rho C_f \widetilde{U}^2}{\rho\widetilde{H}} = -\frac{\widetilde{U}_n^2}{\widetilde{H}_n}\cdot C_f\frac{U^{*2}}{H^*}$$

From term1 to 6, obatin:

$$\widetilde{E}_n\frac{\widetilde{U}_n}{\widetilde{H}_n}\frac{\partial U^*}{\partial t^*} + \frac{\widetilde{U}_n^2}{\widetilde{H}_n}\cdot U^*\frac{\partial U^*}{\partial x^*}\cdot\sin\theta = -g\sin\theta\cos\theta\frac{\partial H^*}{\partial x^*} - g\sin\theta\cos\theta\frac{\partial Z^*}{\partial x^*} - g\sin\theta - \frac{\widetilde{U}_n^2}{\widetilde{H}_n}\cdot C_f\frac{U^{*2}}{H^*}$$

Then rearranging it:

$$\widetilde{E}_n\frac{\widetilde{U}_n}{g\widetilde{H}_n}\frac{\partial U^*}{\partial t^*} + \frac{\widetilde{U}_n^2}{g\widetilde{H}_n}\cdot U^*\frac{\partial U^*}{\partial x^*}\cdot\sin\theta = -\sin\theta\cos\theta\frac{\partial}{\partial x^*}(H^* + Z^*) - \sin\theta - \frac{\widetilde{U}_n^2}{g\widetilde{H}_n}C_f\frac{U^{*2}}{H^*}$$

Let $F^2 = \frac{\widetilde{U}_n^2}{g\widetilde{H}_n}$, and $\widetilde{E}_n = \alpha\tilde{\tau}_{bn}^{\gamma}$, obtain (1.7):

$$\widetilde{E}_n\widetilde{U}_n^{-1}\cdot\frac{\partial U^*}{\partial t^*} + U^*\frac{\partial U^*}{\partial x^*}\cdot\sin\theta = -F^{-2}\sin\theta\cos\theta\frac{\partial}{\partial x^*}(H^* + Z^*) - F^{-2}sin\theta - C_f\frac{U^{*2}}{H^*} \qquad \textbf{(1.7)}$$

There $\alpha \ll 1$, $\widetilde{E}_n \ll 1$ so also try to ignore the term $\alpha\tilde{\tau}_{bn}^{\gamma}\widetilde{U}_n^{-1}\cdot\frac{\partial U^*}{\partial t^*}$, so finally (1.8):

$$U^*\frac{\partial U^*}{\partial x^*} = -F^{-2}\cos\theta\frac{\partial(H^* + Z^*)}{\partial x^*} - F^{-2} - \frac{1}{\sin\theta}\cdot C_f\frac{U^{*2}}{H^*} \qquad \textbf{(1.8)}$$

Then from (1.4) Continunity equation should be:

$$\frac{\partial\widetilde{H}}{\partial\tilde{t}} + \frac{\partial\widetilde{U}\widetilde{H}}{\partial\tilde{x}} = 0 \xrightarrow{\text{Nomalize}} \widetilde{E}_n\frac{\widetilde{H}_n}{\widetilde{H}_n}\frac{\partial U^*}{\partial t^*} + \widetilde{U}_n\frac{\widetilde{H}_n}{\widetilde{H}_n}\frac{\partial U^*H^*}{\partial x^*} = 0$$

$$\frac{\widetilde{E}_n}{\widetilde{U}_n}\frac{\partial U^*}{\partial t^*} + \frac{\partial U^*H^*}{\partial x^*} = 0$$

$$\alpha\tilde{\tau}_{bn}^{\gamma}\widetilde{U}_n^{-1}\cdot\frac{\partial U^*}{\partial t^*} + \frac{\partial U^*H^*}{\partial x^*} = 0 \qquad \textbf{(1.9)}$$

$$\frac{\partial U^*H^*}{\partial x^*} = 0 \qquad \textbf{(1.10)}$$

From (1.5) Erosion on bedrock bed should be:

$$\frac{\partial \tilde{Z}}{\partial \tilde{t}} + \alpha \tilde{\tau}_b^{\gamma} = 0 \xrightarrow{\text{Nomalize}} \tilde{E}_n \frac{\partial Z^*}{\partial t^*} + \alpha (\tilde{\tau}_{bn} \cdot \tau^*)^{\gamma} = 0$$

$$\frac{\partial Z^*}{\partial t^*} + \frac{\alpha (\tilde{\tau}_{bn} \cdot \tau^*)^{\gamma}}{\alpha \tilde{\tau}_{bn}^{\gamma}} = 0$$

And rearranging:

$$\frac{\partial Z^*}{\partial t^*} + \tau_b^{*\gamma} = 0 \tag{1.11}$$

Rearranging them like (1.12a,b,c) :

$$\begin{cases} U^* \dfrac{\partial U^*}{\partial x^*} = -F^{-2} \cos\theta \dfrac{\partial (H^* + Z^*)}{\partial x^*} - F^{-2} - \dfrac{1}{\sin\theta} \cdot C_f \dfrac{U^{*2}}{H^*} \\ \dfrac{\partial U^* H^*}{\partial x^*} = 0 \\ \dfrac{\partial Z^*}{\partial t^*} + \tau_b^{*\gamma} = 0 \end{cases} \tag{1.12a,b,c}$$

And try to do lineaization for small-amplitude perturbations one by one:

First, *Momentum equation* (1.12*a*):

$$\boldsymbol{U^* \frac{\partial U^*}{\partial x^*}} \cdot \sin\theta = -F^{-2} \sin\theta \cos\theta \frac{\partial}{\partial x^*}(H^* + Z^*) - F^{-2} \sin\theta - \boldsymbol{C_f \frac{U^{*2}}{H^*}} \tag{1.12a}$$

Assume that:

$$U^* H^* = 1\,, then: U^* = 1 + u'; \quad H^* = 1 + h'; \quad Z^* = 1 + z'\,; \ \tau_b^* = 1 + \tau_b' = 1 + 2u'$$

So from (1.14a), there red term << 1 could be ignored:

$$\boldsymbol{U^* \frac{\partial U^*}{\partial x^*}} = (1 + u') \frac{\partial (1 + u')}{\partial x^*} = 1 \frac{\cancel{\partial 1}}{\partial x^*} + u' \frac{\cancel{\partial 1}}{\partial x^*} + 1 \frac{\partial u'}{\partial x^*} + \cancel{u'} \frac{\cancel{\partial u'}}{\cancel{\partial x^*}} = \boldsymbol{\frac{\partial u'}{\partial x^*}}$$

$$-F^{-2} \sin\theta \cos\theta \frac{\partial}{\partial x^*}(H^* + Z^*) = -F^{-2} \sin\theta \cos\theta \left(\frac{\partial z'}{\partial x^*} + \frac{\partial h'}{\partial x^*} \right)$$

$$\boldsymbol{C_f \frac{U^{*2}}{H^*}} = C_f \frac{(1 + u')^2}{1 + h'} = C_f \frac{1 + 2u' + \cancel{u'^2}}{1 + h'} = C_f (1 + 2u')(1 - h') = \boldsymbol{C_f (1 + 2u' - h')}$$

Then substituting them into (1.12a), do rearranging step by step:

$$\frac{\partial u'}{\partial x^*} \sin\theta = -F^{-2} \sin\theta \cos\theta \frac{\partial}{\partial x^*}(z' + h') - F^{-2} \sin\theta - C_f(\mathbf{1} + 2u' - h')$$

$$\frac{\partial u'}{\partial x^*} \sin\theta = -F^{-2} \sin\theta \cos\theta \left(\frac{\partial z'}{\partial x^*} + \frac{\partial h'}{\partial x^*} \right) - F^{-2} \sin\theta - \boldsymbol{C_f} - C_f(2u' - h')$$

$$\frac{\partial u'}{\partial x^*} \sin\theta = -F^{-2} \sin\theta \cos\theta \left(\frac{\partial z'}{\partial x^*} + \frac{\partial h'}{\partial x^*} \right) - \left(F^{-2} \sin\theta + \boldsymbol{C_f} \right) - C_f(2u' - h')$$

$$\frac{\partial u'}{\partial x^*} \sin\theta = -F^{-2} \sin\theta \cos\theta \left(\frac{\partial z'}{\partial x^*} + \frac{\partial h'}{\partial x^*} \right) - C_f(2u' - h') - \left(F^{-2} \sin\theta + \boldsymbol{C_f} \right) \tag{1.13a}$$

Note that the final term, if want that term has same order like others in (1.13a)

$$\left(F^{-2} \sin\theta + \boldsymbol{C_f} \right) \to 0$$

And because of $\boldsymbol{C_f}$ must be positive 1 order, both of them should approch to 0, or u', h'

respectively.

$$F^{-2}\sin\theta \to -1$$

So there θ in our case should be in the range of $(\pi < \theta < 0)$, $\sin\theta$ in the range $(-1 < \sin\theta < 0)$. at same time. Noted at end of page2, bed elevation is not increase, but decrease, so for convinient set the $\theta > 0$, and transform (1.12a) in another form (1.14a) :

$$U^*\frac{\partial U^*}{\partial x^*}\sin\theta = -F^{-2}\sin\theta\cos\theta\frac{\partial(H^*+Z^*)}{\partial x^*} + F^{-2}\sin\theta - C_f\frac{U^{*2}}{H^*} \quad \textbf{(1.14a)}$$

With the condition normal flow:

$$\tilde{\tau}_{bn} = \rho C_f\tilde{U}_n^2 = \rho g\tilde{H}_n S,\ C_f = \frac{g\tilde{H}_n}{\tilde{U}_n^2}S = F^{-2}\tan\theta$$

$$\frac{\partial u'}{\partial x^*}\sin\theta = -F^{-2}\sin\theta\cos\theta\left(\frac{\partial z'}{\partial x^*}+\frac{\partial h'}{\partial x^*}\right) - C_f(2u'-h') + (F^{-2}\sin\theta - C_f) \quad \textbf{(1.15a)}$$

Obviously, the condition of make sure 1.15a quasi-steady validity:

$$(F^{-2}\sin\theta - F^{-2}\tan\theta) \cong 0$$

$$\sin\theta \cong \tan\theta$$

(1.15a) become (in flat condition):

$$\frac{\partial u'}{\partial x^*}\cancel{S} = -F^{-2}\cancel{S}\left(\frac{\partial z'}{\partial x^*}+\frac{\partial h'}{\partial x^*}\right) - F^{-2}\cancel{S}(2u'-h')$$

$$\frac{\partial u'}{\partial x^*} = -F^{-2}\left(\frac{\partial z'}{\partial x^*}+\frac{\partial h'}{\partial x^*}\right) - F^{-2}(2u'-h')$$

Finally (1.16a) should be used in question2:

$$F^2\frac{\partial u'}{\partial x^*}+\frac{\partial z'}{\partial x^*}+\frac{\partial h'}{\partial x^*}+2u'-h' = 0 \quad \textbf{(1.16a)}$$

Second, Continunity equation:

$$\frac{\partial U^*H^*}{\partial x^*} = 0 \quad \textbf{(1.12b)}$$

$$\frac{\partial(1+u')(1+h')}{\partial x^*} = \frac{\partial(1+u'+h'+\cancel{u'h'})}{\partial x^*} = \frac{\cancel{\partial 1}}{\cancel{\partial x^*}}+\frac{\partial u'}{\partial x^*}+\frac{\partial h'}{\partial x^*}$$

Then, obtain:

$$\frac{\partial u'}{\partial x^*}+\frac{\partial h'}{\partial x^*} = 0 \quad \textbf{(1.13b)}$$

With integral in x^*:

$$\int\frac{\partial u'}{\partial x^*}dx^* + \int\frac{\partial u'}{\partial x^*}dx^* = 0$$

$$u'+h' = 0 \quad \textbf{(1.14b)}$$

Then, Erosion rate equation on bedrock bed:

$$\frac{\partial Z^*}{\partial t^*}+\tau_b^{*\gamma} = 0$$

(1.12c)

$$\tau_b^* = 1 + \tau_b^{'} = 1 + 2u'$$

$$\frac{\partial Z^*}{\partial t^*} + (1 + 2u')^{\gamma} = 0 \qquad \textbf{(1.13c)}$$

$$\frac{\partial Z^*}{\partial t^*} + 1 + \gamma 2u'^{\gamma} = 0$$

$$\int \frac{\partial Z^*}{\partial t^*} dt^* + \int 1 + \gamma 2u'^{\gamma} dt^* = \mathbf{0} \qquad \textbf{(1.14c)}$$

$$Z^* + t^* = 0$$

This result can not directly be used for our analysis, come back to the base state:

$$\frac{\partial \widetilde{Z}}{\partial \tilde{t}} + \widetilde{E} = 0$$

$$\widetilde{E} = \alpha \tilde{\tau}_b^{\gamma}$$

And there can not confirm dimension of the erosion rate of bedrock by this empirically equation:

Where with steay, uniform flow , flat bed:

$$U = \widetilde{U}_n, H = \widetilde{H}_n, \tilde{\tau}_{bn} = \rho C_f \widetilde{U}_n^{\,2}$$

With conservation of mass, the bedrock eroded is equal to sediments increase.

where $\tilde{q}$ *is flux of sediments*.

λ_{pb} ***is porosity of bedrock , it is nondimensional and decided by material.***

$$\widetilde{E} = \frac{1}{1 - \lambda_{pb}} \frac{\partial \tilde{q}}{\partial \tilde{x}}$$

$$(1 - \lambda_{pb})\widetilde{E} = \frac{\partial \tilde{q}}{\partial \tilde{x}} \xrightarrow{\text{interger}} \tilde{q} = \int (1 - \lambda_{pb})\widetilde{E} d\tilde{x}$$

In the base state:

$$\tilde{q}_n = (1 - \lambda_{pb}) \int \widetilde{E}_n d\tilde{x} = (1 - \lambda_{pb})\widetilde{E}_n \tilde{x}$$

Then constituting them into (1.2):

$$\frac{\partial \widetilde{Z}}{\partial \tilde{t}} + \frac{1}{1 - \lambda_{pb}} \frac{\partial \tilde{q}}{\partial \tilde{x}} = 0 \qquad \textbf{(1.15c)}$$

$$\widetilde{E}_n \frac{\partial Z^*}{\partial t^*} + \frac{1}{1 - \lambda_{pb}} \frac{\partial \tilde{q}_n q^*}{\partial\, \widetilde{H}_n x^* / \sin\theta} = 0 \longrightarrow \frac{\partial Z^*}{\partial t^*} + \frac{1}{1 - \lambda_{pb}} \frac{\tilde{q}_n \sin\theta}{\widetilde{H}_n \widetilde{E}_n} \frac{\partial q^*}{\partial x^*} = 0$$

$$\frac{\partial Z^*}{\partial t^*} + \frac{1}{1 - \lambda_{pb}} \frac{\tilde{q}_n \sin\theta}{\widetilde{H}_n \widetilde{E}_n} \frac{\partial q^*}{\partial x^*} = 0$$

There first treat nondimentional term:

$$\frac{1}{1 - \lambda_{pb}} \frac{\tilde{\mathbf{q}}_{\mathbf{n}} \sin\theta}{\widetilde{H}_n \widetilde{E}_n} = \frac{1}{1 - \lambda_{pb}} \frac{(\mathbf{1} - \lambda_{\mathbf{pb}})\widetilde{\mathbf{E}}_{\mathbf{n}} \tilde{\mathbf{x}} \cdot \sin\theta}{\widetilde{E}_n \widetilde{H}_n} = \frac{\tilde{x} \cdot \sin\theta}{\widetilde{H}_n} = x^*$$

(1.16c)

$$\frac{\partial Z^*}{\partial t^*} + x^* \frac{\partial q^*}{\partial x^*} = 0 \; q^* = q\left[\left(\tau_b^*\right)^\gamma\right]$$
$$\frac{\partial (1+z')}{\partial t^*} + x^* \frac{\partial (1+2u')^\gamma}{\partial x^*} = 0$$
$$\frac{\partial 1}{\partial t^*} + \frac{\partial z'}{\partial t^*} + x^* \frac{\partial (1+2\gamma u')}{\partial x^*} = 0$$
$$\frac{\partial z'}{\partial t^*} + x^* \frac{\partial (2\gamma u')}{\partial x^*} = 0$$
$$\frac{\partial z'}{\partial t^*} + 2\gamma x^* \frac{\partial u'}{\partial x^*} = 0$$

There red parts are obviously non-dimentional, then let $\gamma x^* = N$ finally:

$$\frac{\partial z'}{\partial t^*} + 2N \frac{\partial u'}{\partial x^*} = 0 \qquad \textbf{(1.18c)}$$

Now we can obtain the answer of quesiton1:

The non-dimensional governing equations:

$$\begin{cases} U\dfrac{\partial U}{\partial x}\sin\theta = -F^{-2}\sin\theta\cos\theta\dfrac{\partial (H+Z)}{\partial x} + F^{-2}\sin\theta - C_f\dfrac{U^2}{H} \\ \dfrac{\partial UH}{\partial x} = 0 \quad (UH = 1) \\ \dfrac{\partial Z}{\partial t} + x\dfrac{\partial q}{\partial x} = 0 \end{cases}$$

For question removed * from:

$$\begin{cases} U^*\dfrac{\partial U^*}{\partial x^*}\sin\theta = -F^{-2}\sin\theta\cos\theta\dfrac{\partial (H^*+Z^*)}{\partial x^*} + F^{-2}\sin\theta - C_f\dfrac{U^{*2}}{H^*} & \textbf{(1.14a)} \\ \dfrac{\partial U^*H^*}{\partial x^*} = 0 \quad (U^*H^* = 1) & \textbf{(1.12b)} \\ \dfrac{\partial Z^*}{\partial t^*} + x^*\dfrac{\partial q^*}{\partial x^*} = 0 & \textbf{(1.16c)} \end{cases}$$

In flat condition, for perform linear stabiliyu analysis, did lineaization for small-amplitude perturbations:

$$\begin{cases} F^2\dfrac{\partial u'}{\partial x^*} + \dfrac{\partial z'}{\partial x^*} + \dfrac{\partial h'}{\partial x^*} + 2u' - h' = 0 & \textbf{(1.16a)} \\ u' + h' = 0 & \textbf{(1.14b)} \\ \dfrac{\partial z'}{\partial t^*} + 2N\dfrac{\partial u'}{\partial x^*} = 0 & \textbf{(1.18c)} \end{cases} \quad or \quad \begin{cases} F^2\dfrac{\partial u'}{\partial x} + \dfrac{\partial z'}{\partial x} + \dfrac{\partial h'}{\partial x} + 2u' - h' = 0 \\ u' + h' = 0 \\ \dfrac{\partial z'}{\partial t} + 2N\dfrac{\partial u'}{\partial x} = 0 \end{cases}$$

Base on this result, start perform linear stabiliy analysis in question2.

Stability analysis

Question2: (1) Perform linear stability analysis.

(2) Discuss the stability of the flat bed.

(3)If γ vary from 0.5 to 5, How the stability is affected by value of γ ?

First focus (1), based on results of Question1, :

(2.1a)

(2.1b)

(2.1c)

$$\begin{cases} U\dfrac{\partial U}{\partial x}\sin\theta = -F^{-2}\sin\theta\cos\theta\dfrac{\partial(H+Z)}{\partial x} + F^{-2}\sin\theta - C_f\dfrac{U^2}{H} \\ \dfrac{\partial UH}{\partial x} = 0 \quad (UH=1) \\ \dfrac{\partial Z}{\partial t} + x\dfrac{\partial q}{\partial x} = 0 \end{cases}$$

and

$$\begin{cases} F^2\dfrac{\partial u'}{\partial x} + \dfrac{\partial z'}{\partial x} + \dfrac{\partial h'}{\partial x} + 2u' - h' = 0 \\ u' + h' = 0 \\ \dfrac{\partial z'}{\partial t} + 2N\dfrac{\partial u'}{\partial x} = 0 \end{cases}$$

Where $N = \gamma x$.

Note that (2.2a1) is an equation just fitting flat condition, if did not use that $C_f = \frac{g\tilde{H}_n}{\tilde{U}_n^2}S = F^{-2}\tan\theta$, from (2.1a):

$$\frac{\partial u'}{\partial x}\sin\theta = -F^{-2}\sin\theta\cos\theta\frac{\partial}{\partial x}(h'+z') + F^{-2}\sin\theta - C_f(1+2u'-h')$$

$$\sin\theta F^2\frac{\partial u'}{\partial x} + \sin\theta\cos\theta\frac{\partial}{\partial x}(h'+z') + \sin\theta - F^2C_f(1+2u'-h') = 0$$

$$\sin\theta F^2\frac{\partial u'}{\partial x} + \sin\theta\cos\theta\frac{\partial}{\partial x}(h'+z') + F^2C_f(2u'-h') + (\sin\theta - C_fF^2) = 0$$

There order of red part in (2.2a2) can not be confirmed now. let them equal to 0:

$$\begin{cases} \sin\theta\dfrac{\partial u'}{\partial x} + F^{-2}\sin\theta\cos\theta\dfrac{\partial}{\partial x}(h'+z') + C_f(2u'-h') = 0 \\ \sin\theta - C_fF^2 = 0 \end{cases}$$

Finally obtain that equation set:

$$\begin{cases} \sin\theta\dfrac{\partial u'}{\partial x} + F^{-2}\sin\theta\cos\theta\dfrac{\partial}{\partial x}(h'+z') + C_f(2u'-h') = 0 \\ u' + h' = 0 \\ \dfrac{\partial z'}{\partial t} + 2N\dfrac{\partial u'}{\partial x} = 0 \qquad (N=\gamma x) \\ \sin\theta = C_fF^2 \qquad (Assumed) \end{cases}$$

Then let corresponding perturbations for them with:

$$\begin{cases} z' = z_*e^{ik(x-ct)} + \epsilon.\epsilon \\ u' = u_*e^{ik(x-ct)} + \epsilon.\epsilon \\ h' = h_*e^{ik(x-ct)} + \epsilon.\epsilon \end{cases}$$

Substituting them into (2.2a2) :

$$\sin\theta\frac{\partial}{\partial x}u_*e^{ik(x-ct)} + F^{-2}\sin\theta\cos\theta\frac{\partial}{\partial x}(h_*e^{ik(x-ct)} + z_*e^{ik(x-ct)}) + C_f(2u_*e^{ik(x-ct)} - h_*e^{ik(x-ct)}) = 0$$

$$\sin\theta\, \boldsymbol{ik} u_* e^{ik(x-ct)} + F^{-2}\sin\theta\cos\theta\left(\boldsymbol{ik} h_* e^{ik(x-ct)} + \boldsymbol{ik} z_* e^{ik(x-ct)}\right)$$

$$+ C_f\left(2u_* e^{ik(x-ct)} - h_* e^{ik(x-ct)}\right) = 0$$

$$\sin\theta\, \boldsymbol{ik} u_* + F^{-2}\sin\theta\cos\theta\, \boldsymbol{ik}(h_* + z_*) + C_f(2u_* - h_*) = 0$$

Then:

$$\left(\sin\theta\, \boldsymbol{ik} + 2C_f\right)u_* + \left(F^{-2}\sin\theta\cos\theta\, \boldsymbol{ik} - C_f\right)h_* + (F^{-2}\sin\theta\cos\theta\, \boldsymbol{ik})z_* = 0$$

Substituting them into (2.2b) :

$$u_* e^{ik(x-ct)} + h_* e^{ik(x-ct)} = 0$$
$$u_* e^{ik(x-ct)} + h_* e^{ik(x-ct)} = 0$$
$$u_* + h_* = 0$$

Subtituting them into (2.2c) :

$$\frac{\partial}{\partial t} z_* e^{ik(x-ct)} + 2N\frac{\partial}{\partial x} u_* e^{ik(x-ct)} = 0$$
$$-ikcz_* e^{ik(x-ct)} + 2N \cdot iku_* e^{ik(x-ct)} = 0$$
$$2N \cdot iku_* - ikcz_* = 0$$

Let (2.3 a,b,c) together:

$$\begin{cases} \left(\sin\theta\, \boldsymbol{ik} + 2C_f\right)u_* + \left(F^{-2}\sin\theta\cos\theta\, \boldsymbol{ik} - C_f\right)h_* + (F^{-2}\sin\theta\cos\theta\, \boldsymbol{ik})z_* = 0 \\ u_* + h_* = 0 \\ 2N \cdot iku_* - ikcz_* = 0 \end{cases}$$

Then, from 2.3(a1,b) obtain:

$$h_* = -u_*$$

$$\left(\sin\theta\, \boldsymbol{ik} + 2C_f\right)u_* - \left(F^{-2}\sin\theta\cos\theta\, \boldsymbol{ik} - C_f\right)u_* + (F^{-2}\sin\theta\cos\theta\, \boldsymbol{ik})z_* = 0$$

$$\left(F^{-2}\sin\theta\cos\theta\, \boldsymbol{ik} - \sin\theta\, \boldsymbol{ik} - 3C_f\right)u_* = (F^{-2}\sin\theta\cos\theta\, \boldsymbol{ik})z_*$$

$$u_* = \frac{F^{-2}\sin\theta\cos\theta\, \boldsymbol{ik}}{F^{-2}\sin\theta\cos\theta\, \boldsymbol{ik} - \sin\theta\, \boldsymbol{ik} - 3C_f} z_*$$

$$u_* = \frac{\sin\theta\cos\theta\, \boldsymbol{ik}}{\sin\theta\cos\theta\, \boldsymbol{ik} - F^2\sin\theta\, \boldsymbol{ik} - F^2 3C_f} z_*$$

$$u_* = \frac{\boldsymbol{ik}}{\boldsymbol{ik} - \dfrac{F^2}{\cos\theta}\boldsymbol{ik} - \dfrac{F^2 3C_f}{\sin\theta\cos\theta}} z_*$$

$$u_* = \frac{\boldsymbol{iik}}{(\boldsymbol{1} - \dfrac{F^2}{\cos\theta})\boldsymbol{iik} - \boldsymbol{i}\dfrac{F^2 3C_f}{\sin\theta\cos\theta}} z_*$$

$$u_* = \frac{-\boldsymbol{k}}{\left(\boldsymbol{1} - \dfrac{F^2}{\cos\theta}\right)(-\boldsymbol{k}) - \boldsymbol{i}\dfrac{F^2 3C_f}{\sin\theta\cos\theta}} z_*$$

$$u_* = \frac{\mathbf{1}}{\left(\mathbf{1} - \frac{F^2}{\cos\theta}\right) + \boldsymbol{i}\frac{F^2 3C_f}{k\sin\theta\cos\theta}} z_*$$

There, for convenient let:

$$\frac{F^2}{\cos\theta} = F_c^2$$

$$u_* = \frac{\mathbf{1}}{\left(\mathbf{1} - F_c^2\right) + \boldsymbol{i}\frac{F_c^2 3C_f}{k\sin\theta}} z_*$$

$$u_* = \frac{\left(\mathbf{1} - F_c^2\right) - \boldsymbol{i}\frac{F_c^2 3C_f}{k\sin\theta}}{\left(\mathbf{1} - F_c^2\right)^{\mathbf{2}} + \left(\frac{F_c^2 3C_f}{k\sin\theta}\right)^{\mathbf{2}}} z_*$$

$$z_* = \frac{\left(\mathbf{1} - F_c^2\right)^{\mathbf{2}} + \left(\frac{F_c^2 3C_f}{k\sin\theta}\right)^{\mathbf{2}}}{\left(\mathbf{1} - F_c^2\right) - \boldsymbol{i}\frac{F_c^2 3C_f}{k\sin\theta}} u_*$$

There k was estimated to be order-one, but C_f in an ignorable scaling, let us neglect C_f *term*:

$$u_* \cong \frac{\left(\mathbf{1} - F_c^2\right)}{\left(\mathbf{1} - F_c^2\right)^{\mathbf{2}}} z_*$$

$$u_* \cong \frac{\mathbf{1}}{\left(\mathbf{1} - F_c^2\right)^{\mathbf{2}}} z_*$$

$$u_* \cong \frac{\mathbf{1}}{\left(\mathbf{1} - \frac{F^2}{\cos\theta}\right)^{\mathbf{2}}} z_*$$

Stability analysis:

First substituting (2.4b) into (2.3c):

$$2N \cdot iku_* = ikcz_*$$

Then:

$$2N \cdot iku_* = ikc\frac{\left(\mathbf{1} - F_c^2\right)^{\mathbf{2}} + \left(\frac{F_c^2 3C_f}{k\sin\theta}\right)^{\mathbf{2}}}{\left(\mathbf{1} - F_c^2\right) - \boldsymbol{i}\frac{F_c^2 3C_f}{k\sin\theta}} u_*$$

$$2N \cdot ik = ikc\frac{\left(1-F_c^2\right)^2+\left(\frac{F_c^2 3C_f}{k\sin\theta}\right)^2}{\left(1-F_c^2\right)-i\frac{F_c^2 3C_f}{k\sin\theta}}$$

$$2N = c\frac{\left(1-F_c^2\right)^2+\left(\frac{F_c^2 3C_f}{k\sin\theta}\right)^2}{\left(1-F_c^2\right)-i\frac{F_c^2 3C_f}{k\sin\theta}}$$

From (2.5) let: $c = c_r + ic_i$,

where c_r:

$$2N = c_r\frac{\left(1-F_c^2\right)^2+\left(\frac{F_c^2 3C_f}{k\sin\theta}\right)^2}{\left(1-F_c^2\right)}$$

just like obtain the (2.4c):

$$2N = c_r\frac{\left(1-F_c^2\right)^2}{\left(1-F_c^2\right)}$$

$$c_r = \frac{2N\left(1-F_c^2\right)}{\left(1-F_c^2\right)^2}$$

$$c_r = \frac{2N}{\left(1-F_c^2\right)} \quad (where\ F_c^2 = \frac{F^2}{\cos\theta})$$

$$c_r = \frac{2N}{\left(1-\frac{F^2}{\cos\theta}\right)}$$

where c_i :

$$2N = ic_i\frac{\left(1-F_c^2\right)^2+\left(\frac{F_c^2 3C_f}{k\sin\theta}\right)^2}{-i\frac{F_c^2 3C_f}{k\sin\theta}}$$

$$ic_i = 2N\frac{-i\frac{F_c^2 3C_f}{k\sin\theta}}{\left(1-F_c^2\right)^2+\left(\frac{F_c^2 3C_f}{k\sin\theta}\right)^2}$$

$$c_i = 2N\frac{-F_c^2 3C_f k\sin\theta}{k^2\sin^2\theta\left(1-F_c^2\right)^2+\cancel{\left(F_c^2 3C_f\right)^2}}$$

$$c_i = 2N\frac{-F_c^2 3C_f k\sin\theta}{k^2\sin^2\theta\left(1-F_c^2\right)^2}$$

$$c_i = -2N\frac{F_c^2 3C_f}{k\sin\theta\left(1-F_c^2\right)^2}$$

$$c_i = -\frac{2NF_c^2 3C_f}{k\left(1-F_c^2\right)^2\sin\theta}$$

$$c_i = -\frac{6NC_f F_c^2}{k\left(1-F_c^2\right)^2}\cdot\frac{1}{\sin\theta}\quad (where\ F_c^2 = \frac{F^2}{\cos\theta})$$

$$c_i = -\frac{6NC_f\dfrac{F^2}{\cos\theta}}{k\left(1-\dfrac{F^2}{\cos\theta}\right)^2\sin\theta}$$

Finally:

$$\begin{cases} c_r = \dfrac{2N}{\left(1-\dfrac{F^2}{\cos\theta}\right)} \\ c_i = -\dfrac{6NC_f\dfrac{F^2}{\cos\theta}}{k\left(1-\dfrac{F^2}{\cos\theta}\right)^2\sin\theta} \end{cases}$$

Discuss the Flat condition:

When flat condition : $there\ \cos\theta \cong 1,\ \sin\theta \cong \tan\theta = S$: where $\sin\theta = C_f F^2$ (assumed), $N = \gamma x$.

From (2.6a,b):

$$\begin{cases} c_r = \dfrac{2N}{(1-F^2)} \\ c_i = -\dfrac{6NC_f F^2}{k(1-F^2)^2\sin\theta} \end{cases}$$

$$c_i = -\frac{6NC_f F^2}{k(1-F^2)^2 C_f F^2} = -\frac{6N}{k(1-F^2)^2}$$

with the flat assumption:

$$\begin{cases} c_r = \dfrac{2x\gamma}{(1-F^2)} \\ c_i = -\dfrac{6x\gamma}{k(1-F^2)^2} \end{cases}$$

From 2.7a, about c_r for subcritical flow, there:

$$0 < F^2 < 1; \qquad (1 - F^2) > 0$$

$$c_r = \frac{2x\gamma}{(1 - F^2)} > 0$$

Modelling with TELEMAC-TOMAWAC-GAIA/SYSIPHE

Viollet

Purpose
This test demonstrates the ability of TELEMAC-3D to model thermal and stratified flow.
1.2 Description
The test case considers the stable configuration of Pierre-Louis Viollet's experimentation (1980) with a Froude number of 0.9, which consists in a 2 layer flow of same height, h = 0.1 m. The lower layer has a velocity U2 and a temperature T2. The upper layer has a velocity U1 < U2 and a temperature T1 > T2.
1.2.1 Reference
1.2.2 Geometry and Mesh
Bathymetry
Channel tilt = 5.30921 × 10−6 (see figure 3.18.2)
Geometry
Channel length = 10 m (100h)
Channel width = 1 m (10h)
Mesh
1280 triangular elements (see figure 3.18.2)
697 nodes
27 planes regularly spaced on the vertical (σ transformation).
1.2.3 Physical parameters
Turbulence: k-ε in both directions
Prandtl number: 0.71
Karman constant: 0.41
Bottom friction: Haaland law with coefficient equal to 63.4505112
Density law is a function of temperature.

1.2.4 Initial and Boundary Conditions
Initial conditions
U1 = 0.05 m/s T1 = 25.35◦C
U2= 0.05 m/s T2 = 20◦C
Constant height of 0.2 m (2h)
Boundary conditions
Closed boundaries on sides.
Upstream prescribed flow rate: 0.01 m3/s
Downstream prescribed water level: 0.19995 m
A double logarithmic velocity profile is imposed for the lower layer and a logarithmic profile for the upper layer according the following formulae defined in the BORD3D subroutine:
where z1 and z2 are the levels in upper and lower layers respectively (starting from the lower level of each layer). ξs = 10−4 m et dz is the distance between two planes (i.e. dz = h/26 since

there are 27 planes).
Upstream k and ε profiles are imposed according the following formulae defined in the KEPCL3 subroutine:
where nturb = 5 × 10−3, Cμ = 0.09 and δ = 10−6
Surface and bottom boundary condition for ε are defined in the KEPICL subroutine: Neumann at bottom and Dirichlet at surface.

1.2.5 General parameters

Time step: 0.1 s
Simulation duration: 500 s (8 min 20 s)

1.2.6 Numerical parameters

Non-hydrostatic computation
Advection of velocities, temperature and k-ε: N-type MURD scheme

Computational Setting

```
1.  / 'Pierre-Louis VIOLLET' test case
2.  /
3.  / TELEMAC-3D V7P071 - AUGUST 2015
4.  /
5.  / M. FERRAND, E. GAGNAIRE-RENOU
6.  / C.-T. PHAM - J. DESOMBRE - P. LANG
7.  /
8.  /-----------------------------------------------------------------
   ----------
9.  /                          TELEMAC-3D
10. /
11. TITLE = 'TELEMAC 3D : PLV TEST CASE'
12. /
13. FORTRAN FILE            : 'user_fortran'
14. BOUNDARY CONDITIONS FILE : geo_viollet.cli
15. GEOMETRY FILE           : geo_viollet.slf
16. FORMATTED DATA FILE 1    : froud.txt
17. 3D RESULT FILE          : r3d_viollet.slf
18. 2D RESULT FILE          : r2d_viollet.slf
19. FORMATTED RESULTS FILE   : profil_actif_v7p0r1_Fr09_Haal_b.txt
20. /-----------------------------------------------------------------
   ----
21. /     OPTIONS GENERALES
22. /-----------------------------------------------------------------
   ----
23. NUMBER OF TIME STEPS = 5000
24. TIME STEP = 0.1
25. /NUMBER OF TIME STEPS = 1300
26. /TIME STEP = 0.25
27. GRAPHIC PRINTOUT PERIOD = 100
```

```
LISTING PRINTOUT PERIOD = 100
NUMBER OF HORIZONTAL LEVELS = 27
/
INITIAL CONDITIONS : 'CONSTANT DEPTH'
INITIAL DEPTH : 0.2
PRESCRIBED FLOWRATES : 0.;0.01
/
/ SLOPE OF 5.30921.10-6
/ LENGTH OF 10 m
/
PRESCRIBED ELEVATIONS : 0.19995;0.
/LOGARITHMIC PROFILE
VELOCITY VERTICAL PROFILES = 2;2
/
TIDAL FLATS = NO
/
VARIABLES FOR 2D GRAPHIC PRINTOUTS = U,V,H,B,S
VARIABLES FOR 3D GRAPHIC PRINTOUTS = Z,U,V,W,RI,NUZ,TA1,NAZ1,EPS,K
MASS-BALANCE = YES
NON-HYDROSTATIC VERSION = YES
PRANDTL NUMBER  = 0.71
KARMAN CONSTANT = 0.41
/-------------------------
/  CONVECTION-DIFFUSION
/-------------------------
SCHEME FOR ADVECTION OF VELOCITIES : 4
SCHEME FOR ADVECTION OF TRACERS    : 4
SCHEME FOR ADVECTION OF K-EPSILON  : 4
/-------------------------
/ K-EPSILON
/-------------------------
HORIZONTAL TURBULENCE MODEL = 3
/-------------------------
/ Viscosite constante
/-------------------------
/ HORIZONTAL TURBULENCE MODEL = 1
/ COEFFICIENT FOR HORIZONTAL DIFFUSION OF VELOCITIES = 5.D-3
/-------------------------
/ K-EPSILON
/ -------------------------
VERTICAL TURBULENCE MODEL = 3
/---------------------------
/ MIXING LENGTH + PRANDTL + VIOLLET
/---------------------------
/ VERTICAL TURBULENCE MODEL = 2
```

```
73. / MIXING LENGTH MODEL = 1
74. / DAMPING FUNCTION = 2
75. /----------------------------
76. /    TERMES SOURCES
77. /----------------------------
78. FRICTION COEFFICIENT FOR THE BOTTOM = 63.4505112
79. TURBULENCE REGIME FOR THE BOTTOM = 1 / LISSE
80. LAW OF BOTTOM FRICTION = 1 / HAALAND
81. /----------------------------
82. /    PROPAGATION
83. /----------------------------
84. MAXIMUM NUMBER OF ITERATIONS FOR PROPAGATION           = 500
85. MAXIMUM NUMBER OF ITERATIONS FOR VERTICAL VELOCITY      = 50
86. MAXIMUM NUMBER OF ITERATIONS FOR DIFFUSION OF VELOCITIES = 200
87. SOLVER FOR VERTICAL VELOCITY = 7
88. ACCURACY FOR DIFFUSION OF VELOCITIES = 1.E-14
89. ACCURACY FOR VERTICAL VELOCITY      = 1.E-14
90. ACCURACY FOR DIFFUSION OF K-EPSILON  = 1.E-16
91. ACCURACY FOR DIFFUSION OF TRACERS    : 1.E-16
92. /
93. IMPLICITATION FOR DEPTH     = 1.
94. /----------------------------
95. /         SEDIMENT
96. /----------------------------
97. NUMBER OF TRACERS : 1
98. NAMES OF TRACERS : 'TEMPERATURE                   '
99. INITIAL VALUES OF TRACERS : 20.
100./MODIFIED BY PRINCI.F
101.TRACERS VERTICAL PROFILES : 1;1
102.PRESCRIBED TRACERS VALUES : 20.;20.
103./0 : PASSIVE; 1 : TEMPERATURE; 2 : SALT ;4 : GIVEN BETA
104.DENSITY LAW : 1
105./ MF
106./BETA EXPANSION COEFFICIENT FOR TRACERS = 2.E-4
107./STANDARD VALUES FOR TRACERS : 20.
108./
109.COEFFICIENT FOR VERTICAL DIFFUSION OF TRACERS = 1.44E-7
110./
111./ NO FRICTION
112.LAW OF FRICTION ON LATERAL BOUNDARIES : 0
113./
114.MASS-LUMPING FOR DEPTH     : 1.
115.MASS-LUMPING FOR DIFFUSION  : 1.
116.IMPLICITATION FOR DIFFUSION : 2.
```

Results

Time =500s

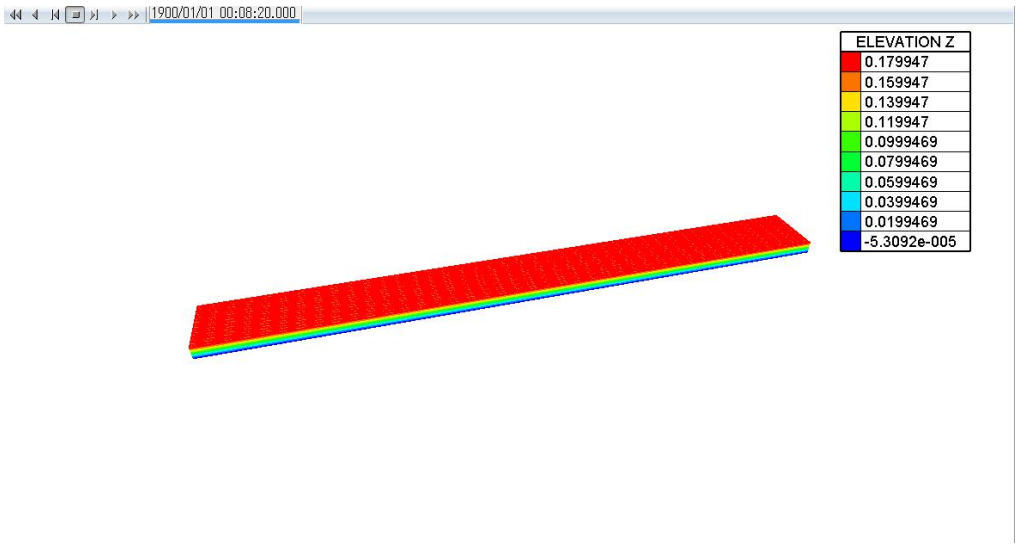
```
================================================================================

ITERATION     5000 TIME    0 D  0 H  8 MN  20.0000 S   (       500.0000 S)
================================================================================

 ELEVATION AT THE BOUNDARY READ                                    IN THE ASCII
FILE
 VALUE OF THE TRACER          1 AT THE BOUNDARIES READ IN THE
                  ASCII FILE
 T
--------------------------------------------------------------------------------

                   ADVECTION STEP
--------------------------------------------------------------------------------

                   ADVECTION STEP
 MURD3D OPTION:  4       1 ITERATIONS
 MURD3D OPTION:  4       1 ITERATIONS
 MURD3D OPTION:  4       1 ITERATIONS
 DIFFUSION OF WN
 GRACJG (BIEF) :        2 ITERATIONS, ABSOLUTE PRECISION:   0.5962028E-14
--------------------------------------------------------------------------------

      PROPAGATION AND DIFFUSION WITH WAVE EQUATION
--------------------------------------------------------------------------------

      PROPAGATION AND DIFFUSION WITH WAVE EQUATION
 GRACJG (BIEF) :        3 ITERATIONS, ABSOLUTE PRECISION:   0.8546384E-06
--------------------------------------------------------------------------------

              DYNAMIC PRESSURE STAGE
--------------------------------------------------------------------------------

              DYNAMIC PRESSURE STAGE
 GRACJG (BIEF) :        0 ITERATIONS, ABSOLUTE PRECISION:   0.1881424E-04
--------------------------------------------------------------------------------

              VELOCITY PROJECTION STEP
--------------------------------------------------------------------------------

              VELOCITY PROJECTION STEP
--------------------------------------------------------------------------------

      ADVECTION-DIFFUSION OF K-EPSILON OR OMEGA STEP
--------------------------------------------------------------------------------
```

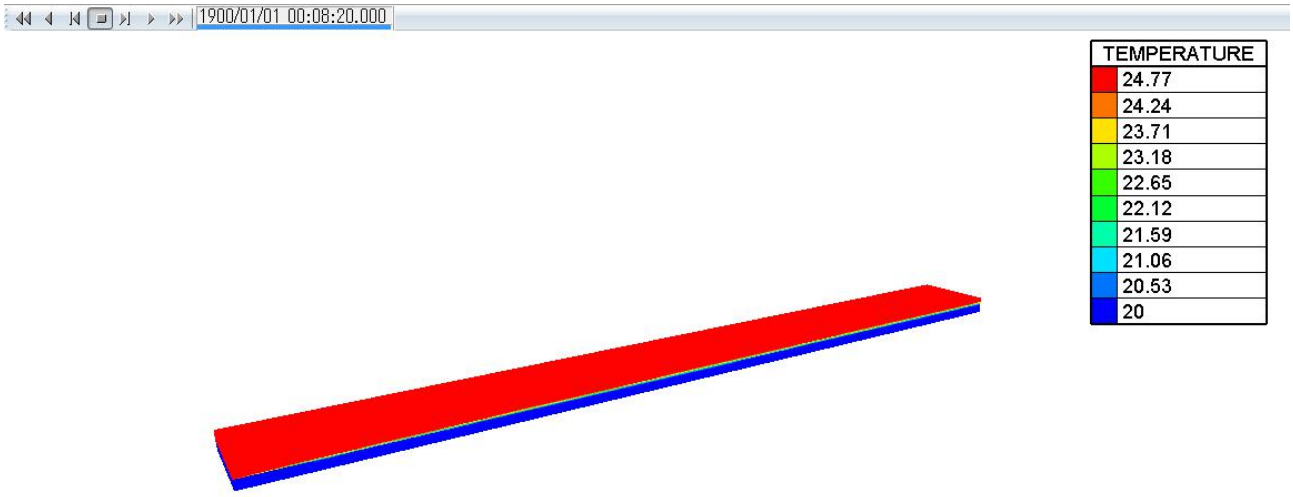
```
      ADVECTION-DIFFUSION OF K-EPSILON OR OMEGA STEP
-------------------------------------------------------------------------------

      ADVECTION-DIFFUSION OF K-EPSILON OR OMEGA STEP
 MURD3D OPTION:  4       1 ITERATIONS
 DIFFUSION OF AKN
 GRACJG (BIEF) :        5 ITERATIONS, ABSOLUTE PRECISION:   0.1344992E-16
 MURD3D OPTION:  4       1 ITERATIONS
 DIFFUSION OF EPN
 GRACJG (BIEF) :        5 ITERATIONS, ABSOLUTE PRECISION:   0.1510353E-16
-------------------------------------------------------------------------------

      ADVECTION-DIFFUSION OF TRACERS
-------------------------------------------------------------------------------

      ADVECTION-DIFFUSION OF TRACERS
 MURD3D OPTION:  4       1 ITERATIONS
 DIFFUSION OF TRN1
 GRACJG (BIEF) :        6 ITERATIONS, RELATIVE PRECISION:   0.2972278E-16
-------------------------------------------------------------------------------

               MASS BALANCE
-------------------------------------------------------------------------------

               MASS BALANCE

  WATER
MASS AT THE PREVIOUS TIME STEP               :     2.000026
MASS AT THE PRESENT TIME STEP                :     2.000027
MASS LEAVING THE DOMAIN DURING THIS TIME STEP :   -0.1021764E-05
ERROR ON THE MASS DURING THIS TIME STEP      :    0.4921648E-07
FLUX BOUNDARY    1:   -0.9989782E-02 M3/S  ( >0 : ENTERING  <0 : EXITING )
FLUX BOUNDARY    2:    0.1000000E-01 M3/S  ( >0 : ENTERING  <0 : EXITING )

  TRACER  1: TEMPERATURE     , UNIT :                * M3)

ADVECTIVE FLUX THROUGH BOUNDARIES OR SOURCES  :   -0.2668144E-03
DIFFUSIVE FLUX THROUGH THE BOUNDARIES         :    -0.000000
MASS AT THE PREVIOUS TIME STEP                :     44.15182
MASS AT THE PRESENT TIME STEP                 :     44.15186
MASS EXITING (BOUNDARIES OR SOURCE)           :   -0.2668144E-04
ERROR ON THE MASS DURING THIS TIME STEP       :   -0.7335730E-05

-------------------------------------------------------------------------------
```

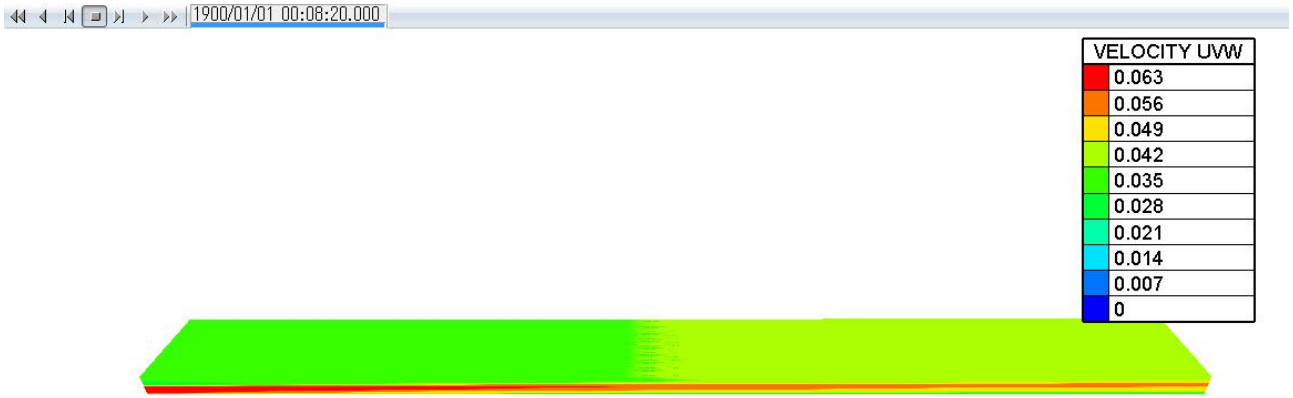
```
--------------------------------------------------------------------------------

               FINAL MASS BALANCE
--------------------------------------------------------------------------------

               FINAL MASS BALANCE
T =         500.0000

--- WATER ---
INITIAL MASS                       :     2.000000
FINAL MASS                         :     2.000027
MASS LEAVING THE DOMAIN (OR SOURCE) :   -0.3026270E-04
MASS LOSS                          :    0.3204819E-05

--- TRACER 1: TEMPERATURE     , UNIT :                * M3)
INITIAL MASS                       :     45.14279
FINAL MASS                         :     44.15186
MASS EXITING (BOUNDARIES OR SOURCE) :     1.045525
MASS LOSS                          :   -0.5458968E-01

 END OF TIME LOOP

 CALL OF P_EXIT IN ITS VOID VERSION

STOP 0                    ************************************

                    *    END OF MEMORY ORGANIZATION:    *
                    ************************************

 CORRECT END OF RUN

 ELAPSE TIME :
                            16  MINUTES
                            36  SECONDS
```

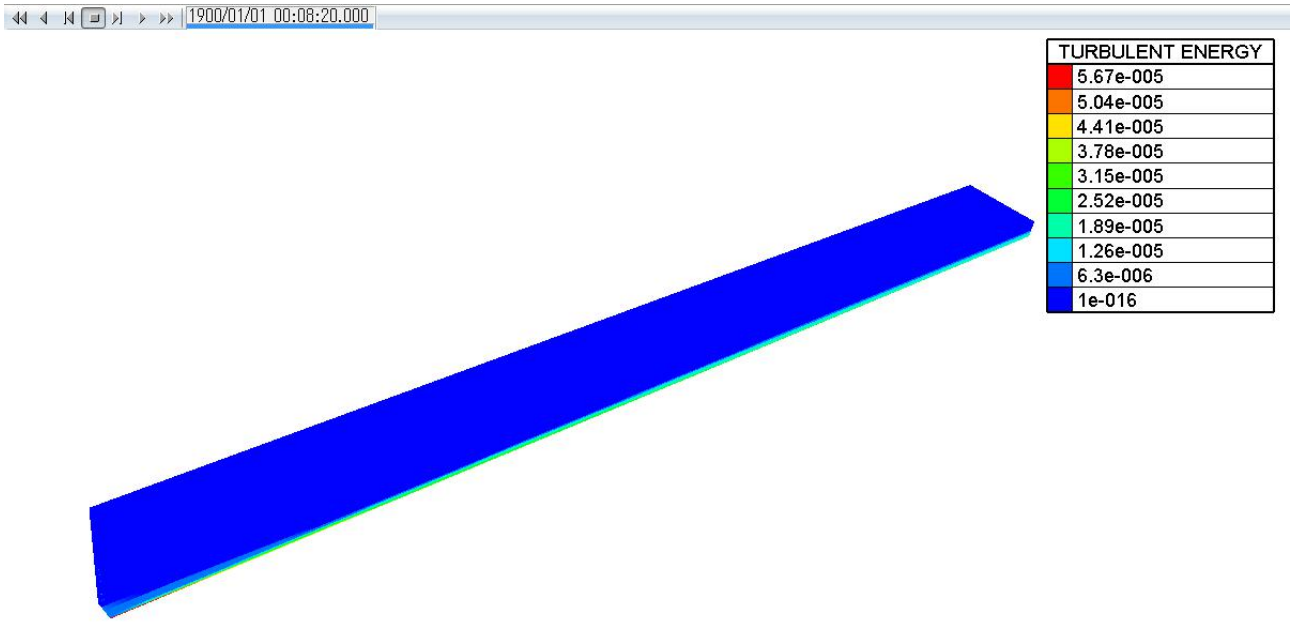
Results:

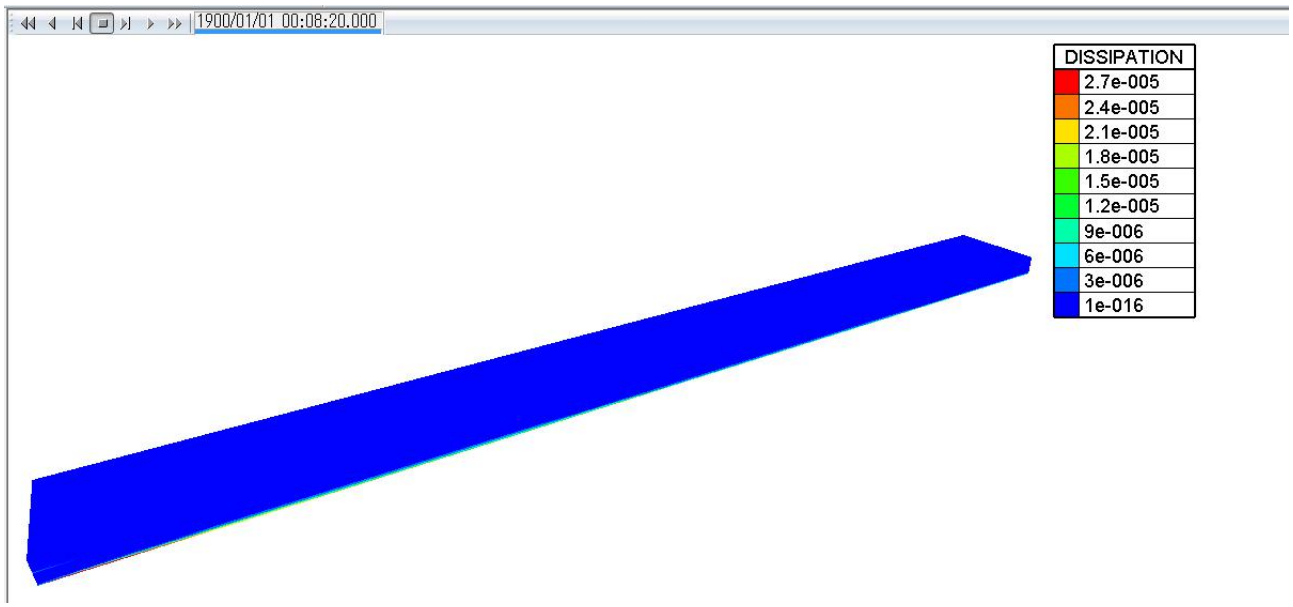
3D RESULTS:



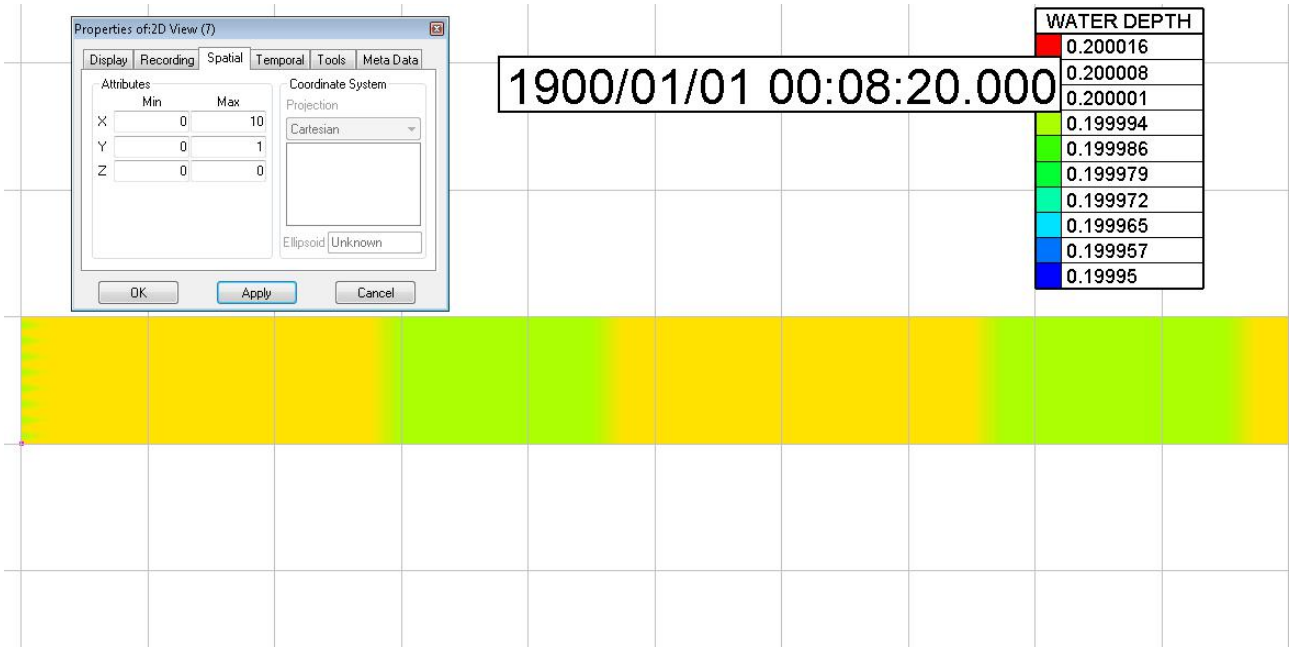

1900/01/01 00:08:20.000
TEMPERATURE
24.77
24.24
23.71
23.18
22.65
22.12
21.59
21.06
20.53
20

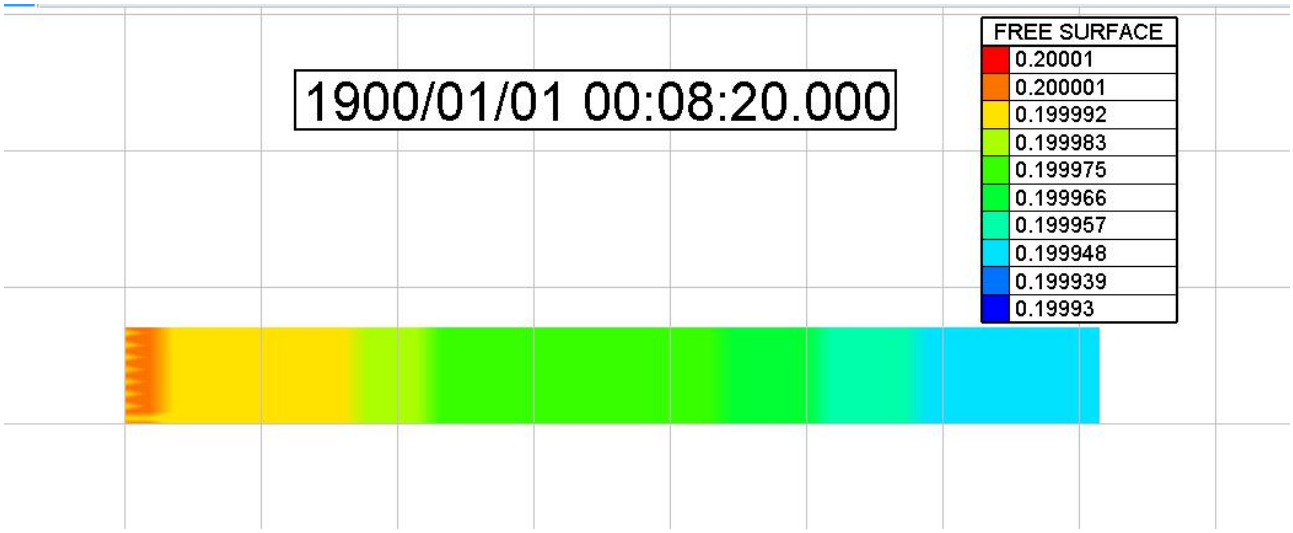

1900/01/01 00:08:20.000
VELOCITY UVW
0.063
0.056
0.049
0.042
0.035
0.028
0.021
0.014
0.007
0

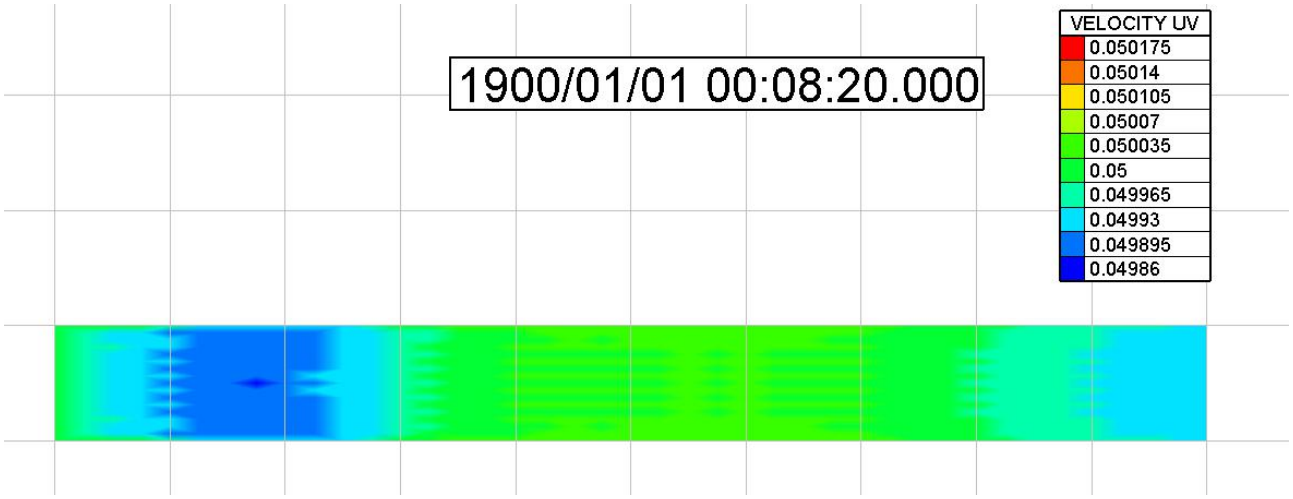

1900/01/01 00:08:20.000
TURBULENT ENERGY
5.67e-005
5.04e-005
4.41e-005
3.78e-005
3.15e-005
2.52e-005
1.89e-005
1.26e-005
6.3e-006
1e-016

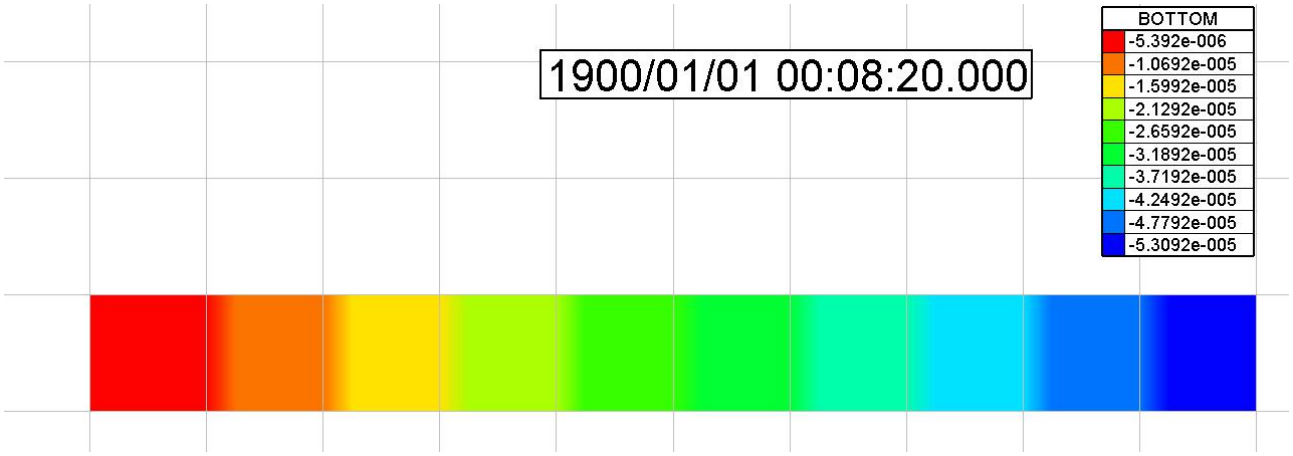

1900/01/01 00:08:20.000
DISSIPATION
2.7e-005
2.4e-005
2.1e-005
1.8e-005
1.5e-005
1.2e-005
9e-006
6e-006
3e-006
1e-016

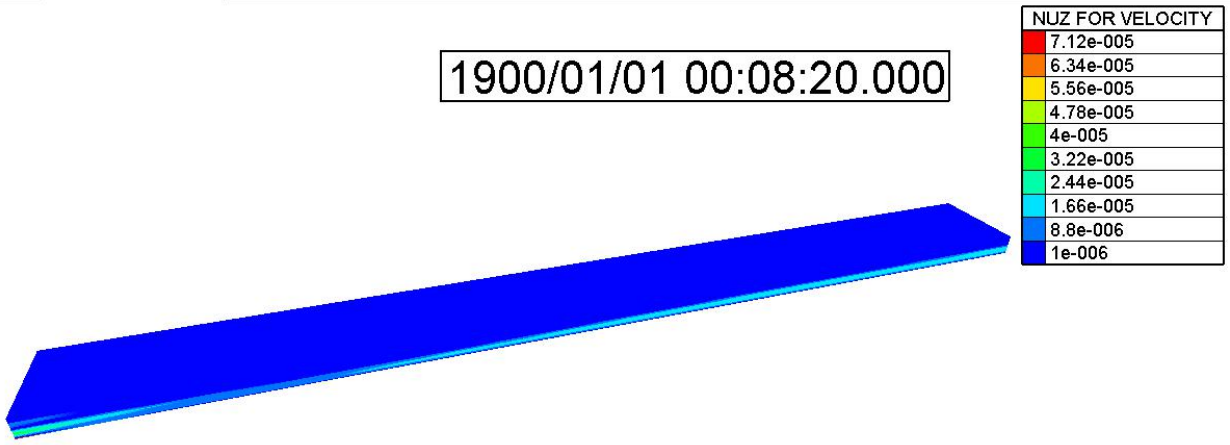
2D views:


Properties of:2D View (7)
Display
Recording
Spatial
Temporal
Tools
Meta Data
Attributes
Min
Max
X
0
10
Y
0
1
Z
0
0
Coordinate System
Projection
Cartesian
Ellipsoid
Unknown
OK
Apply
Cancel
1900/01/01 00:08:20.000
WATER DEPTH
0.200016
0.200008
0.200001
0.199994
0.199986
0.199979
0.199972
0.199965
0.199957
0.19995


1900/01/01 00:08:20.000
FREE SURFACE
0.20001
0.200001
0.199992
0.199983
0.199975
0.199966
0.199957
0.199948
0.199939
0.19993


1900/01/01 00:08:20.000
VELOCITY UV
0.050175
0.05014
0.050105
0.05007
0.050035
0.05
0.049965
0.04993
0.049895
0.04986


1900/01/01 00:08:20.000
BOTTOM
-5.392e-006
-1.0692e-005
-1.5992e-005
-2.1292e-005
-2.6592e-005
-3.1892e-005
-3.7192e-005
-4.2492e-005
-4.7792e-005
-5.3092e-005



Nonlinear Wave

Purpose

This test demonstrates the ability of TELEMAC-3D to simulate the evolution of a monochromatic linear wave over a bar. This test case corresponds to a physical model and measurements published by Dingemans (conditions C) [1].

2.2 Description

We consider a tank 32 m long and 0.3 m wide. The evolution of the topography along the channel is presented on figure 3.13.1. A wave is imposed at the entrance of the channel. The goal is to simulate the evolution of this wave when propagating over the bar.
The simulation is made with and without hydrostatic hypothesis.

2.2.1 Reference

[1] DINGEMANS M.W., Comparison of computations with Boussinesq-like models and laboratory measurements. MAST-G8M note, H1684, Delft Hydraulics, 32 pp. 1994.
[2] BENOIT M., Projet CLAROM-ECOMAC (FICHE CEP&M M06101.99). Modélisation non-linéaire par les équations de Boussinesq de la propagation des vagues non-déferlantes en zone côtière. Rapport EDF-LNHE HP-75/01/069. 2001.

2.2.2 Geometry and Mesh

Bathymetry

Geometry

Channel length = 32 m
Channel width = 0.3 m

Mesh

7,680 triangular elements
5,124 nodes
10 planes regularly spaced

2.2.3 Physical parameters

Diffusion: no
Coriolis: no
Wind: no

2.2.4 Initial and Boundary Conditions

Initial conditions

Initial free surface at level 0.
No velocity
Boundary conditions
Closed lateral boundaries. No bottom friction
Imposed wave at the entrance (amplitude 0.04 m)

2.2.5 General parameters

Time step: 0.0025 s
Simulation duration: 33 s

2.2.6 Numerical parameters

Non-hydrostatic version
Advection of velocities: N-type MURD scheme

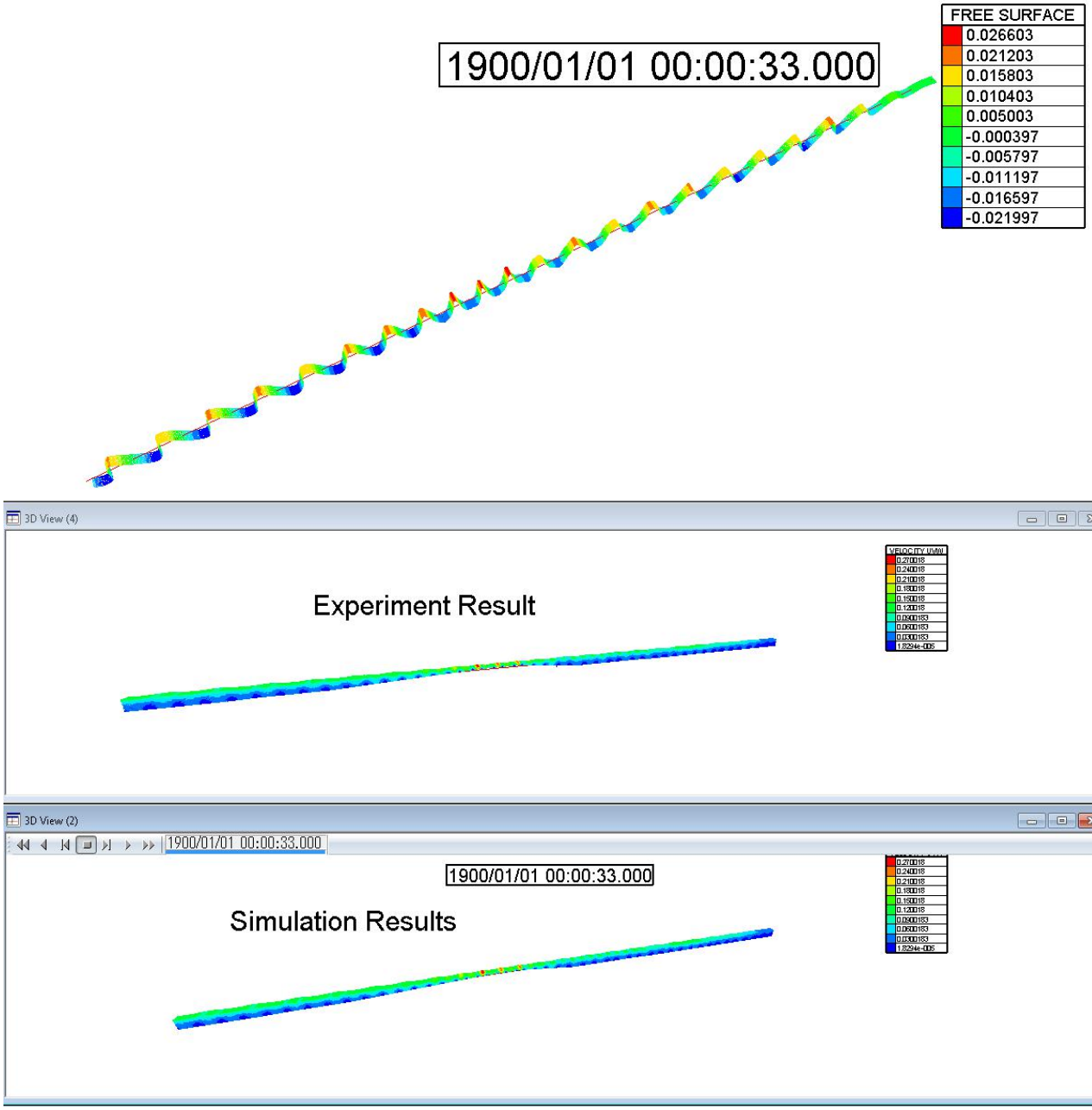
Computational Setting

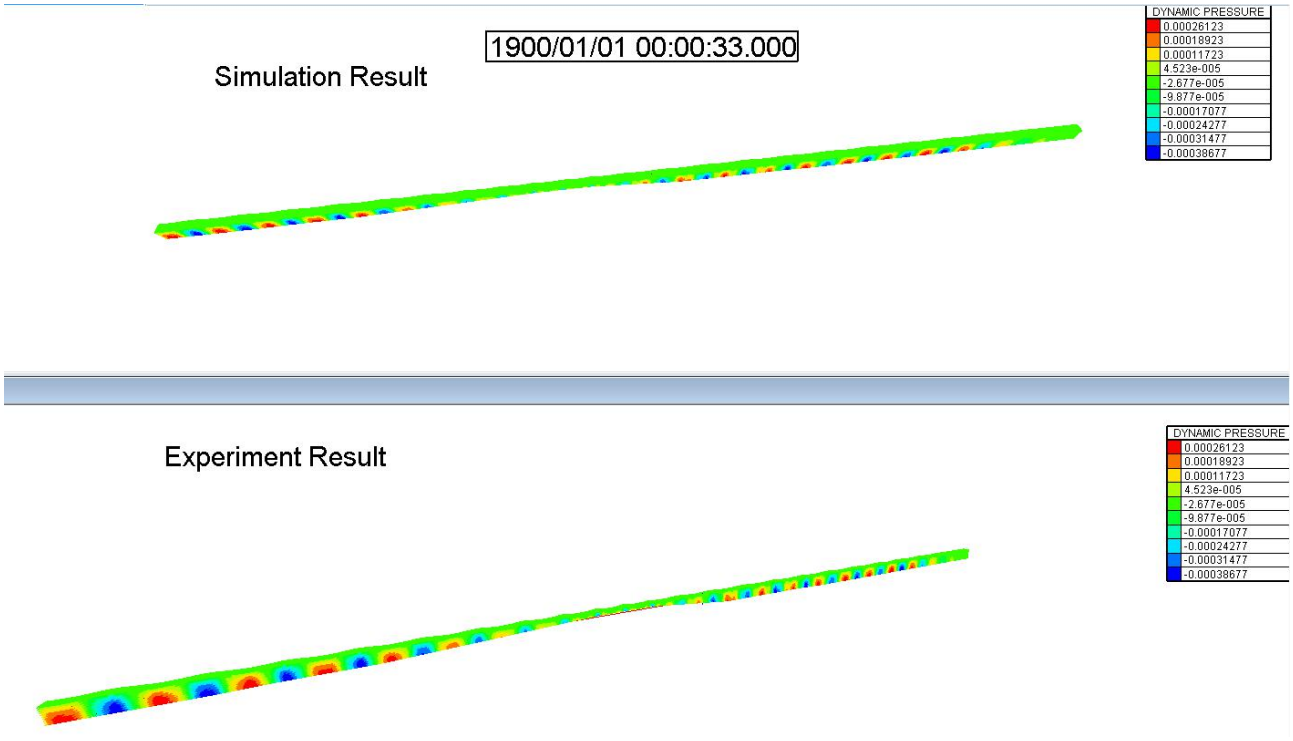
```
/*********************************************************************
****/
/
/          FICHIER DE DECLARATION DES MOTS CLES DU CODE              /
/                           POSTEL 3D                               /
/
/*********************************************************************
****/
/
/
GEOMETRY FILE     : geo_nonlinearwave.slf
3D RESULT FILE    : r3d_nonlinearwave.slf
/
HORIZONTAL CROSS SECTION FILE    : toH
VERTICAL CROSS SECTION FILE      : tov
/
NUMBER OF HORIZONTAL CROSS SECTIONS : 0
NUMBER OF VERTICAL CROSS SECTIONS   : 1
/
NUMBER OF FIRST RECORD FOR CROSS SECTIONS : 1
PRINTOUT PERIOD FOR CROSS SECTIONS : 1
/
REFERENCE LEVEL FOR EACH HORIZONTAL CROSS SECTION : 1;6
ELEVATION FROM REFERENCE LEVEL : 0.;0.
/
ABSCISSAE OF THE VERTICES OF VERTICAL CROSS SECTION 1 :
0.;32.
ORDINATES OF THE VERTICES OF VERTICAL CROSS SECTION 1 :
0.15;0.15
DISTORSION BETWEEN VERTICAL AND HORIZONTAL   : 25.
```

Results

```
================================================================================

ITERATION    13200 TIME    0 D  0 H  0 MN  33.0000 S   (        33.0000 S)
================================================================================


--------------------------------------------------------------------------------

                  ADVECTION STEP
--------------------------------------------------------------------------------

                  ADVECTION STEP
 MURD3D OPTION:  4       1 ITERATIONS
 MURD3D OPTION:  4       1 ITERATIONS
 MURD3D OPTION:  4       1 ITERATIONS
 DIFFUSION OF WN
 GRACJG (BIEF) :        1 ITERATIONS, RELATIVE PRECISION:    0.000000
--------------------------------------------------------------------------------

      PROPAGATION AND DIFFUSION WITH WAVE EQUATION
--------------------------------------------------------------------------------

      PROPAGATION AND DIFFUSION WITH WAVE EQUATION
 RESCJG (BIEF) :       19 ITERATIONS, ABSOLUTE PRECISION:   0.9750091E-06
 RESCJG (BIEF) :        2 ITERATIONS, ABSOLUTE PRECISION:   0.3369164E-07
--------------------------------------------------------------------------------

                  ADVECTION STEP
--------------------------------------------------------------------------------

                  ADVECTION STEP
 MURD3D OPTION:  4       1 ITERATIONS
 MURD3D OPTION:  4       1 ITERATIONS
 MURD3D OPTION:  4       1 ITERATIONS
 DIFFUSION OF WN
 GRACJG (BIEF) :        1 ITERATIONS, RELATIVE PRECISION:    0.000000
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------

      PROPAGATION AND DIFFUSION WITH WAVE EQUATION
--------------------------------------------------------------------------------

      PROPAGATION AND DIFFUSION WITH WAVE EQUATION
 RESCJG (BIEF) :       18 ITERATIONS, ABSOLUTE PRECISION:   0.9131569E-06
 RESCJG (BIEF) :        1 ITERATIONS, ABSOLUTE PRECISION:   0.7740492E-07

 END OF TIME LOOP

 CALL OF P_EXIT IN ITS VOID VERSION

                   ************************************
STOP 0                   *    END OF MEMORY ORGANIZATION:    *

                   ************************************

 CORRECT END OF RUN

 ELAPSE TIME :
                            1  HOURS
                           54  MINUTES
                           55  SECONDS


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```


1900/01/01 00:00:33.000
FREE SURFACE
0.026603
0.021203
0.015803
0.010403
0.005003
-0.000397
-0.005797
-0.011197
-0.016597
-0.021997
3D View (4)
Experiment Result
3D View (2)
1900/01/01 00:00:33.000
1900/01/01 00:00:33.000
Simulation Results



This comparison shows a very good agreement between results and measurements

Tide

Purpose

This test demonstrates the availability of TELEMAC-3D to model the propagation of tide in a maritime domain by computing tidal boundary conditions.

3.2 Description

A coastal area located in the English Channel off the coast of Brittany (in France) close to the real location of the Paimpol-Bréhat tidal farm is modelled to simulate the tide and the tidal currents over this area. Time and space varying boundary conditions are prescribed over liquid boundaries.

3.2.1 Reference

3.2.2 Geometry and Mesh

Bathymetry

Real bathymetry of the area bought from the SHOM (French Navy Hydrographic and Oceanographic Service). c Copyright 2007 SHOM. Produced with the permission of SHOM. Contract

number 67/2007

Geometry

Almost a rectangle with the French coasts on one side 22 km × 24 km

Mesh

4,385 triangular elements

2,386 nodes

11 planes regularly spaced on the vertical

3.2.3 Physical parameters

Vertical turbulence model: mixing length model
Horizontal viscosity for velocity: 10-4 m2/s
Coriolis: yes (constant coefficient over the domain = 1.10 × 10-4 rad/s)
No wind, no atmospheric pressure, no surge and nor waves

3.2.4 Initial and Boundary Conditions

Initial conditions
Constant elevation
No velocity
Boundary conditions
Elevation and horizontal velocity boundary conditions computed by TELEMAC-3D from an harmonic constants database (JMJ from LNH).

3.2.5 General parameters

Time step: 20 s
Simulation duration: 90,000 s = 25 h

3.2.6 Numerical parameters

Non-hydrostatic version
Advection for velocities: Characteristics method
Thompson method with calculation of characteristics for open boundary conditions
Free Surface Gradient Compatibility = 0.5 (not 0.9) to prevent on wiggles
Tidal flats with correction of Free Surface by elements, treatments to have h ≥ 0

Computational Setting

```
TITLE = 'TELEMAC 3D: tide'
/-------------------------------------------------
/     GENERAL OPTIONS
/-------------------------------------------------
FORTRAN FILE                    : 'user_fortran'
BOUNDARY CONDITIONS FILE         : geo_tide.cli
GEOMETRY FILE                   : geo_tide.slf
HARMONIC CONSTANTS FILE          : har_tide-jmj.txt
3D RESULT FILE                  : r3d_tide-jmj_type.slf
2D RESULT FILE                  : r2d_tide-jmj_type.slf
/-------------------------------------------------
VARIABLES FOR 2D GRAPHIC PRINTOUTS            : U,V,S,H,M
VARIABLES FOR 3D GRAPHIC PRINTOUTS            : Z,U,V,W
TIME STEP                                    : 20.
NUMBER OF TIME STEPS                          : 4500
GRAPHIC PRINTOUT PERIOD                       : 45
LISTING PRINTOUT PERIOD                       : 450
MASS-BALANCE                                 : YES
/-------------------------------------------------
INITIAL CONDITIONS                            : 'CONSTANT ELEVATION'
INITIAL ELEVATION                             : 9.8
/-------------------------------------------------
```

```
51. NUMBER OF HORIZONTAL LEVELS                    : 11
52. MESH TRANSFORMATION                           : 1
53. TIDAL FLATS                                  : YES
54. OPTION FOR THE TREATMENT OF TIDAL FLATS        : 1
55. / TO HAVE POSITIVE DEPTHS EVERYWHERE
56. TREATMENT OF NEGATIVE DEPTHS                   : 2
57. OPTION FOR LIQUID BOUNDARIES                   : 2
58. /-------------------------------------------------
59. /                     TIDE CONDITIONS
60. /-------------------------------------------------
61. OPTION FOR TIDAL BOUNDARY CONDITIONS           : 3
62. TIDAL DATA BASE                               : 1
63. COEFFICIENT TO CALIBRATE TIDAL RANGE           : 1.01
64. COEFFICIENT TO CALIBRATE SEA LEVEL             : 5.58
65. /-------------------------------------------------
66. /    SOURCE TERMS
67. /-------------------------------------------------
68. NUMBER OF BOTTOM SMOOTHINGS                    : 2
69. LAW OF BOTTOM FRICTION                        : 3
70. FRICTION COEFFICIENT FOR THE BOTTOM            : 25.
71. VERTICAL TURBULENCE MODEL                      : 2
72. MIXING LENGTH MODEL                           : 3
73. HORIZONTAL TURBULENCE MODEL                    : 1
74. COEFFICIENT FOR HORIZONTAL DIFFUSION OF VELOCITIES : 1.E-4
75. CORIOLIS                                      : YES
76. CORIOLIS COEFFICIENT                           : 1.10E-4
77. /-------------------------------------------------
78. /   ADVECTION-DIFFUSION
79. /-------------------------------------------------
80. SCHEME FOR ADVECTION OF VELOCITIES             : 1
81. NON-HYDROSTATIC VERSION                       : YES
82. FREE SURFACE GRADIENT COMPATIBILITY            : 0.5
83. /
84. ACCURACY FOR DIFFUSION OF VELOCITIES           : 1.E-6
85. MAXIMUM NUMBER OF ITERATIONS FOR DIFFUSION OF VELOCITIES : 500
86. PRECONDITIONING FOR DIFFUSION OF VELOCITIES    : 34
87. /-------------------------------------------------------------------
   ---/
88. /    PROPAGATION                                                    /
89. /-------------------------------------------------------------------
   ---/
90. MAXIMUM NUMBER OF ITERATIONS FOR PROPAGATION       : 500
91. SOLVER FOR PROPAGATION                            : 2
92. MAXIMUM NUMBER OF ITERATIONS FOR PPE               : 500
93. ACCURACY FOR PPE                                  : 1.E-6
```

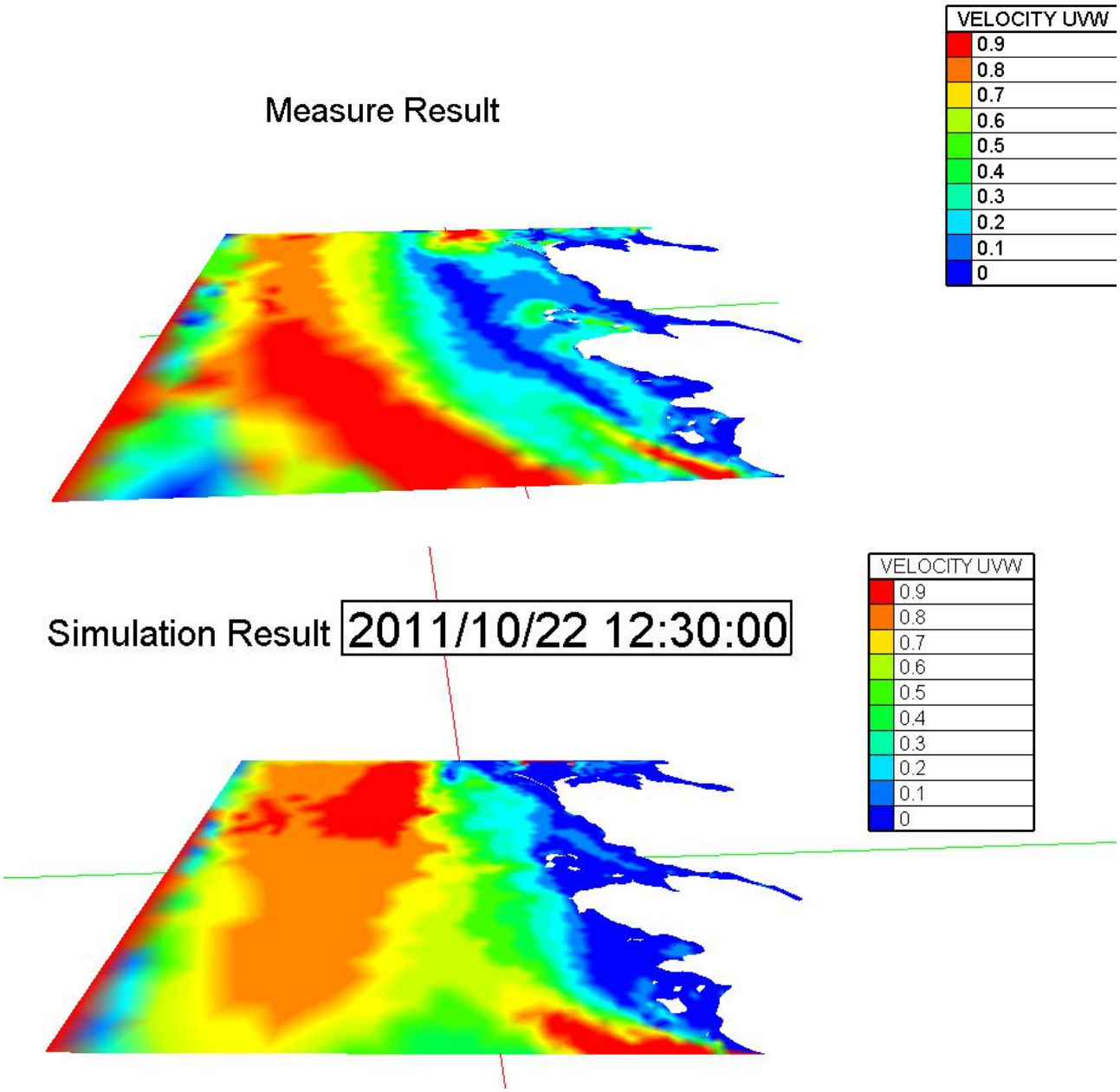
```
94. PRECONDITIONING FOR PPE                                    : 17
95. IMPLICITATION FOR DEPTH                                    : 1.
96. /
97. MASS-LUMPING FOR DEPTH                                     : 1.
98. MASS-LUMPING FOR DIFFUSION                                  : 1.
```

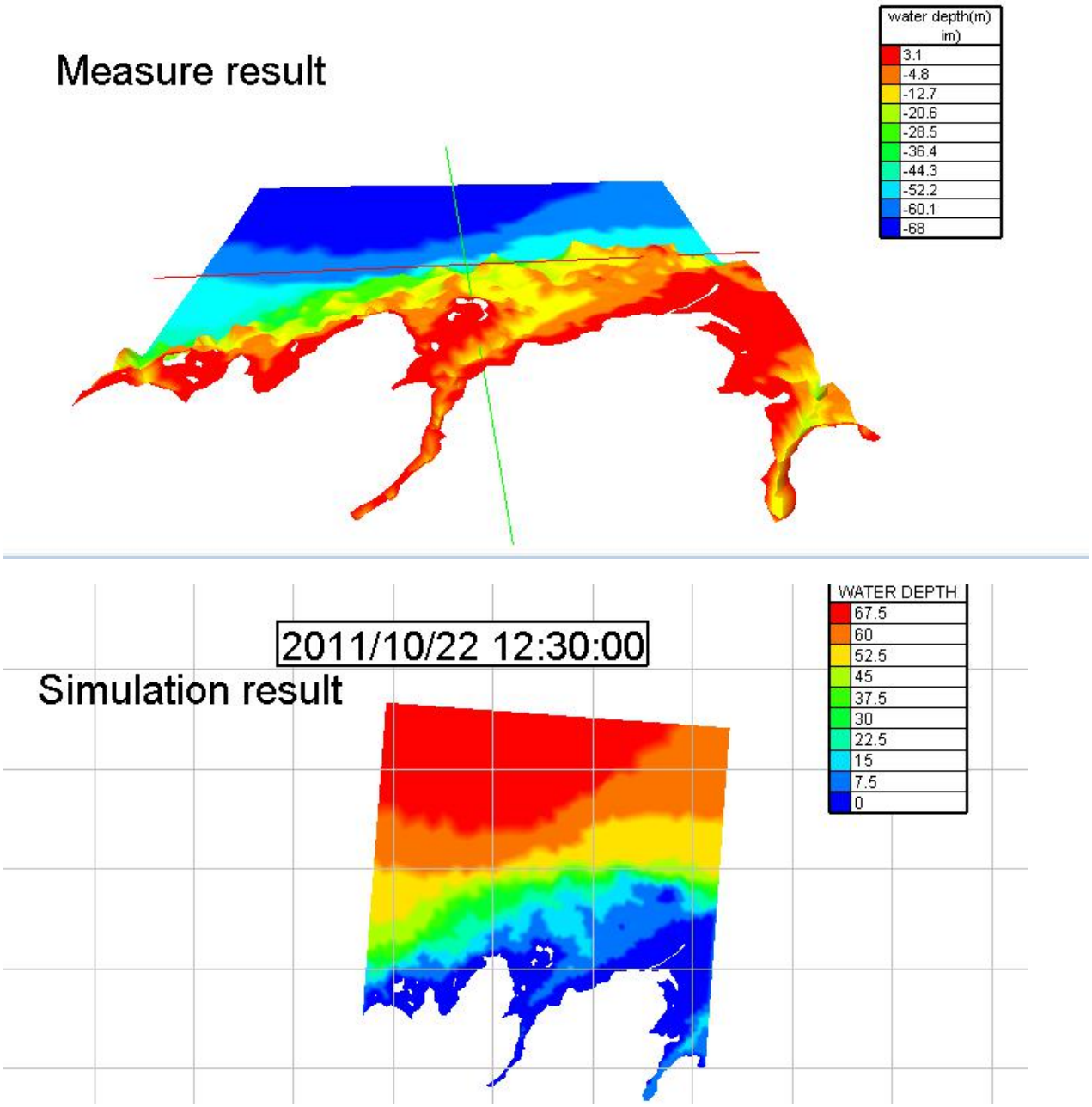
3.3 Results

```
------------------------------------------------------------------------------

ITERATION     2250 TIME    0 D 12 H 30 MN   0.0000 S   (      45000.0000 S)
==============================================================================

------------------------------------------------------------------------------

                 ADVECTION STEP
------------------------------------------------------------------------------

                 ADVECTION STEP
 DIFFUSION OF WN
 GRACJG (BIEF) :       1 ITERATIONS, RELATIVE PRECISION:   0.7612000E-09
------------------------------------------------------------------------------

      PROPAGATION AND DIFFUSION WITH WAVE EQUATION
------------------------------------------------------------------------------

      PROPAGATION AND DIFFUSION WITH WAVE EQUATION
 RESCJG (BIEF) :       9 ITERATIONS, RELATIVE PRECISION:   0.5749686E-06
 POSITIVE DEPTHS OBTAINED IN     5 ITERATIONS
------------------------------------------------------------------------------

              DYNAMIC PRESSURE STAGE
------------------------------------------------------------------------------

              DYNAMIC PRESSURE STAGE
 GRACJG (BIEF) :       2 ITERATIONS, RELATIVE PRECISION:   0.2442248E-06
------------------------------------------------------------------------------

              VELOCITY PROJECTION STEP
------------------------------------------------------------------------------

              VELOCITY PROJECTION STEP
------------------------------------------------------------------------------

              MASS BALANCE
------------------------------------------------------------------------------

              MASS BALANCE
```

```
                MASS BALANCE
--------------------------------------------------------------------------------

                MASS BALANCE

  WATER
MASS AT THE PREVIOUS TIME STEP                :    0.1609674E+11
MASS AT THE PRESENT TIME STEP                 :    0.1609753E+11
MASS LEAVING THE DOMAIN DURING THIS TIME STEP :     460358.1
ERROR ON THE MASS DURING THIS TIME STEP       :    -1254000.
FLUX BOUNDARY    1:    -23017.91     M3/S  ( >0 : ENTERING  <0 : EXITING )

--------------------------------------------------------------------------------

                FINAL MASS BALANCE
--------------------------------------------------------------------------------

                FINAL MASS BALANCE
T =       45000.0000

--- WATER ---
INITIAL MASS                        :    0.1595576E+11
FINAL MASS                          :    0.1609753E+11
MASS LEAVING THE DOMAIN (OR SOURCE) :    0.4138485E+10
MASS LOSS                           :   -0.4280257E+10

 END OF TIME LOOP

 CALL OF P_EXIT IN ITS VOID VERSION

STOP 0                    ************************************

                  *    END OF MEMORY ORGANIZATION:    *
                  ************************************

 CORRECT END OF RUN

 ELAPSE TIME :
                           4  MINUTES
                          17  SECONDS
```

3D


VELOCITY UVW
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Measure Result
Simulation Result
2011/10/22 12:30:00
VELOCITY UVW
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0



Littoral

Purpose

This test case illustrates the setup of a three-way coupling problem waves, currents and sediment transport.

1.2 Description of the problem

A wave, current and sediment transport simulation in a straight, uniform stretch of coastline is considered. The beach is located at y = 200 m, the sloping bed is imposed in subroutine corfon. The offshore depth is 10 m. This is the classical test case of a rectilinear beach with sloping bed The model allows to calculate the littoral transport. ! This test case illustrates the effect of waves which is :

• to generate the current induced littoral current parallel to the beach
• to increase the sand transport rate using the Bijker sand transport formula.

1.3 Physical parameters

1.3.1 Geometry and Mesh

A domain of 200 ×1000 m2 is considered, with a regular mesh with elements size of the order Δx = 20 m and Δy = 5 m The beach is 1000 m long, 200 m wide The beach slope (Y=200m) is 5% and defined in corfon.f The water depth along the open boundary (Y=0) is h=10m We use a trianglular regular grid

1.4 Initial and Boundary Conditions

1.4.1 Wave conditions

Incoming waves (waves height , period and directions) are imposed offshore at y = 0, such that Hs= 1 m, Tp = 8 s. The Jonswap spectrum is used. The waves direction is 30 deg relative to the y−axis. The mesh is as shown on Figure 5.1

1.5 Numerical parameters

⇒ Offshore (Y=0): Offshore wave imposed/no littoral current/no set up

Tomawac: The wave height is imposed on the offshore boundary (5 4 4) (Hs=1m), for a wave period (Tp=8s).
Telemac2D: The current and free surface are imposed to 0 along the offshore boundary (5 5 5).

t2d_littoral.cas

```
1.  /DEBUGGER = 1
2.  /-----------------------------------------------------------------
   ----
3.  /
4.  /          STEERING FILE
5.  /
6.  /          TELEMAC
7.  /-----------------------------------------------------------------
   ----
8.  /                  SETTINGS
9.  /-----------------------------------------------------------------
   ----
10. /
11. COUPLING WITH : 'TOMAWAC,SISYPHE'
12. /
13. WAVE DRIVEN CURRENTS      = YES
14. TOMAWAC STEERING FILE     = 'tom_littoral.cas'
15. COUPLING PERIOD FOR TOMAWAC = 10
16. /
17. SISYPHE STEERING FILE     = 'sis_littoral.cas'
18. COUPLING PERIOD FOR SISYPHE = 10
19. /
20. FORTRAN FILE             = 'T2D_user_fortran'
21. BOUNDARY CONDITIONS FILE  = 'geo_t2d_littoral.cli'
22. GEOMETRY FILE            = 'geo_littoral.slf'
```

```
23. RESULTS FILE             = 't2d_littoral.slf'
24. /-------------------------------------------------------------------
   ----
25. / INITIAL CONDITIONS
26. /-------------------------------------------------------------------
   ----
27. INITIAL CONDITIONS = 'COTE NULLE'
28. /
29. /-------------------------------------------------------------------
   ----
30. /                       GENERAL OPTIONS
31. /-------------------------------------------------------------------
   ----
32. /
33. TITLE = 'LITTORAL'
34. /
35. VARIABLES FOR GRAPHIC PRINTOUTS = H,S,B,U,V
36. /
37. TIME STEP              = 10.
38. NUMBER OF TIME STEPS    = 100
39. GRAPHIC PRINTOUT PERIOD = 100
40. LISTING PRINTOUT PERIOD = 10
41. /-------------------------------------------------------------------
   ----
42. INFORMATION ABOUT SOLVER          = YES
43. DIFFUSION OF VELOCITY             = YES
44. TURBULENCE MODEL                  = 4
45. VELOCITY DIFFUSIVITY              = 1.e-6
46. CORIOLIS                          = NO
47. /
48. /-----------------------------------------------
49. /        BOUNDARY CONDITIONS
50. /-----------------------------------------------
51. /
52. BOTTOM SMOOTHINGS           = 0
53. LAW OF BOTTOM FRICTION      = 5
54. FRICTION COEFFICIENT        = 0.05
55. /
56. PRESCRIBED ELEVATIONS       = 0.
57. /-----------------------------------------
58. /         NUMERICAL OPTIONS
59. /-----------------------------------------
60. /
61. DISCRETIZATIONS IN SPACE = 12;11
62. /
```

```
63. TIDAL FLATS   = NO
64. OPTION FOR THE TREATMENT OF TIDAL FLATS = 1
65. H CLIPPING    = NO
66. /
67. MASS-BALANCE  = YES
68. /
69. SOLVER ACCURACY                        = 1e-05
70. MAXIMUM NUMBER OF ITERATIONS FOR SOLVER = 100
71. /
72. TYPE OF ADVECTION                           = 1;5
73. NUMBER OF SUB-ITERATIONS FOR NON-LINEARITIES = 1
74. /
75. PROPAGATION                  = YES
76. IMPLICITATION FOR DEPTH      = 0.6
77. IMPLICITATION FOR VELOCITY   = 0.6
78. /
79. PRECONDITIONING          = 2
80. /
81. SOLVER                  = 7
82. SOLVER OPTION            = 4
83. MASS-LUMPING ON H        = 1.
84. MASS-LUMPING ON VELOCITY  = 0.9
85. /
86. MATRIX STORAGE           = 3
```

tom_littoral.cas

```
1.  DEBUGGER = 1
2.  /*****************************************************************
   **
3.  /            STEERING FILE
4.  /            TOMAWAC
5.  /
6.  /*****************************************************************
   **
7.  /----------------------------------------------------------------
   --
8.  /     FILES
9.  /----------------------------------------------------------------
   --
10. FORTRAN FILE ='Tom_user_fortran'
11. GEOMETRY FILE = 'geo_littoral.slf'
12. BOUNDARY CONDITIONS FILE = 'geo_tom_littoral.cli'
13. 2D RESULTS FILE = 'tom_littoral.slf'
14. GLOBAL RESULT FILE = 'glob.slf'
15. PUNCTUAL RESULTS FILE ='littoral2D.spe'
16. ABSCISSAE OF SPECTRUM PRINTOUT POINTS= 500
```

```
17. ORDINATES OF SPECTRUM PRINTOUT POINTS= 100
18. /----------------------------------------------------------------
   --
19. / INPUTS - OUTPUTS
20. /----------------------------------------------------------------
   --
21. TITLE = 'LITTORAL'
22. PERIOD FOR GRAPHIC PRINTOUTS = 1
23. VARIABLES FOR 2D GRAPHIC PRINTOUTS = 'HM0,DMOY,TPR5,ZF,WD,FX,FY,UX,UY'
24. PERIOD FOR LISTING PRINTOUTS = 10
25. /----------------------------------------------------------------
   --
26. / DISCRETISATION
27. /----------------------------------------------------------------
   --
28. MINIMAL FREQUENCY = 0.05
29. FREQUENTIAL RATIO = 1.1007
30. /NOMBRE DE FREQUENCES = 25
31. NUMBER OF FREQUENCIES = 12
32. /NOMBRE DE DIRECTIONS = 36
33. NUMBER OF DIRECTIONS = 12
34. TIME STEP = 10.
35. / SO FAR NECESSARY FOR VALIDATION
36. NUMBER OF TIME STEP = 100
37. /----------------------------------------------------------------
   --
38. / INITIAL CONDITIONS
39. /----------------------------------------------------------------
   --
40. INITIAL STILL WATER LEVEL = 10.
41. TYPE OF INITIAL DIRECTIONAL SPECTRUM = 6
42. INITIAL SIGNIFICANT WAVE HEIGHT = 0.5
43. INITIAL PEAK FREQUENCY = 0.125
44. INITIAL PEAK FACTOR = 3.3
45. INITIAL ANGULAR DISTRIBUTION FUNCTION = 1
46. INITIAL WEIGHTING FACTOR FOR ADF = 1.0
47. INITIAL MAIN DIRECTION 1 = 30.
48. INITIAL DIRECTIONAL SPREAD 1 = 3
49. /----------------------------------------------------------------
   --
50. / BOUNDARY CONDITIONS
51. /----------------------------------------------------------------
   --
52. TYPE OF BOUNDARY DIRECTIONAL SPECTRUM = 6
53. BOUNDARY SIGNIFICANT WAVE HEIGHT = 1.
```

```
54. BOUNDARY PEAK FREQUENCY = 0.125
55. BOUNDARY PEAK FACTOR = 3.3
56. BOUNDARY MAIN DIRECTION 1 = 30.
57. BOUNDARY DIRECTIONAL SPREAD 1 = 3.
58. /-----------------------------------------------------------------
   --
59. / OPTIONS
60. /-----------------------------------------------------------------
   --
61. MINIMUM WATER DEPTH = 0.05
62. INFINITE DEPTH = NO
63. CONSIDERATION OF SOURCE TERMS = YES
64. CONSIDERATION OF A WIND = NO
65. BOTTOM FRICTION DISSIPATION = 1
66. /COEFFICIENT DE FROTTEMENT SUR LE FOND = 0.042
67. / Valeur par defaut: 0.038
68. /===========DEFERLEMENT
69. NUMBER OF BREAKING TIME STEPS = 5
70. /=========== 1 : Battjes et Janssen (1978)
71. /=========== 2 : Thornton et Guza (1983)
72. DEPTH-INDUCED BREAKING DISSIPATION = 1
```

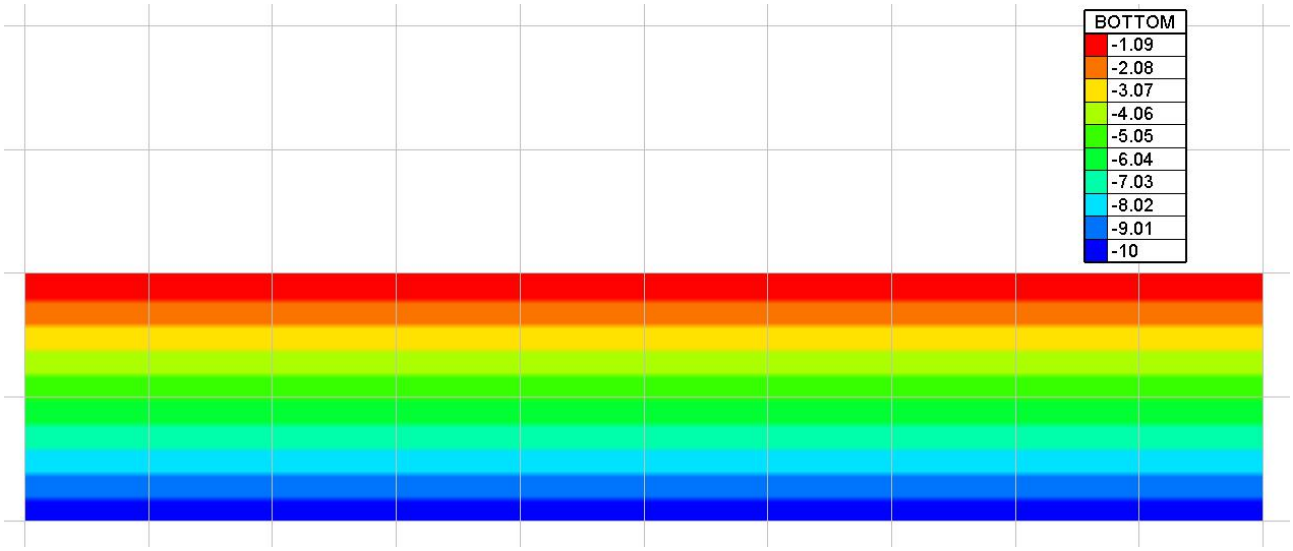
sis_littoral.cas

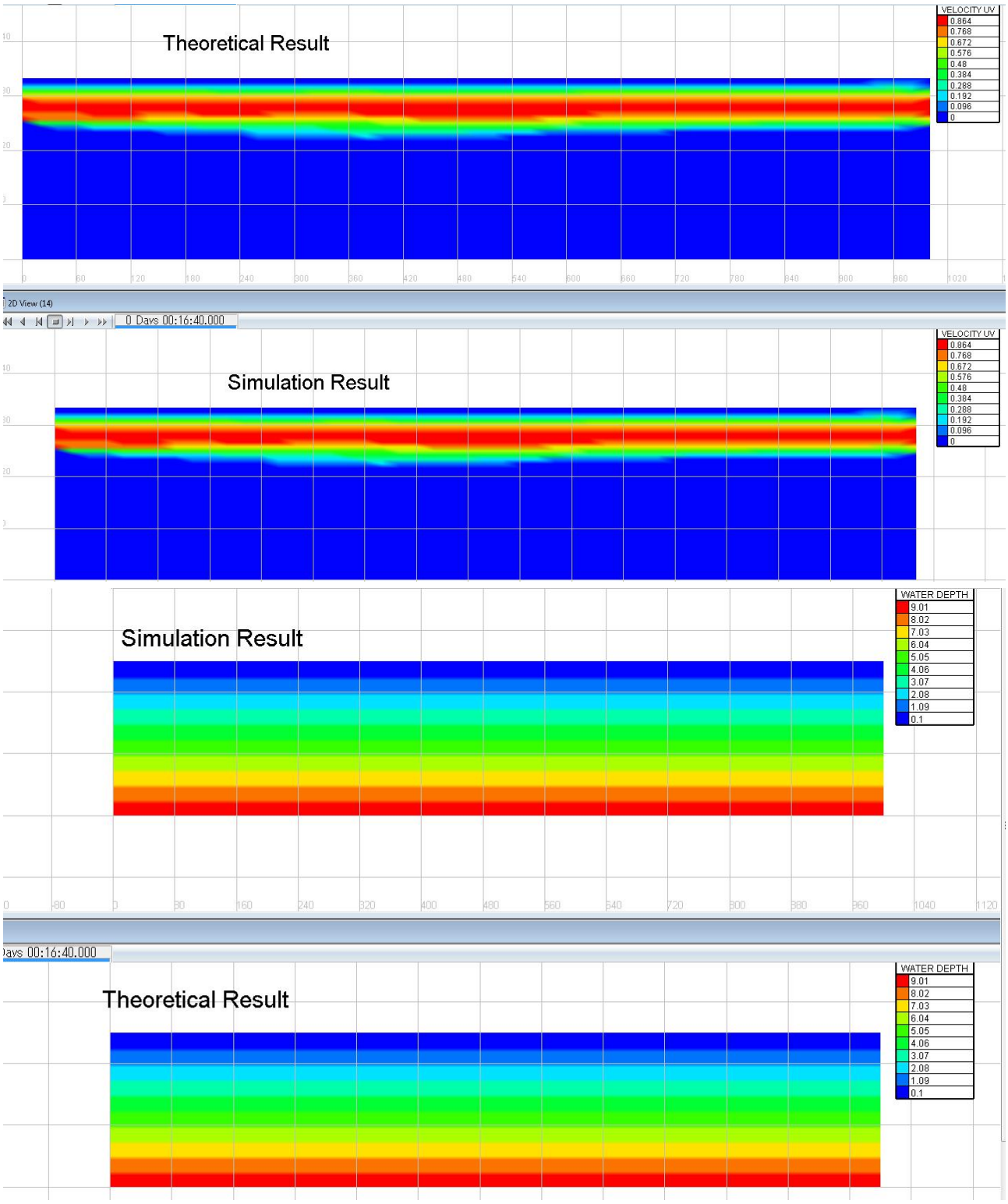
```
1.  /
2.  TITLE = 'LITTORAL'
3.  /
4.  / FILES (1)
5.  / _______________
6.  /mode sisyphe seul
7.  GEOMETRY FILE = 'geo_littoral.slf'
8.  BOUNDARY CONDITIONS FILE = 'geo_sis_littoral.cli'
9.  RESULTS FILE ='sis_littoral.slf'
10. /
11. / FRICTION
12. /_____________________
13. /
14. LAW OF BOTTOM FRICTION = 5
15. FRICTION COEFFICIENT = 0.05
16. RATIO BETWEEN SKIN FRICTION AND MEAN DIAMETER = 3
17. /
18. / INPUTS - OUTPUTS
19. / ___________________________________________
20. /
21. VARIABLES FOR GRAPHIC PRINTOUTS = U,V,H,M,W,X,TOB
22. /
23. MASS-BALANCE = YES
```

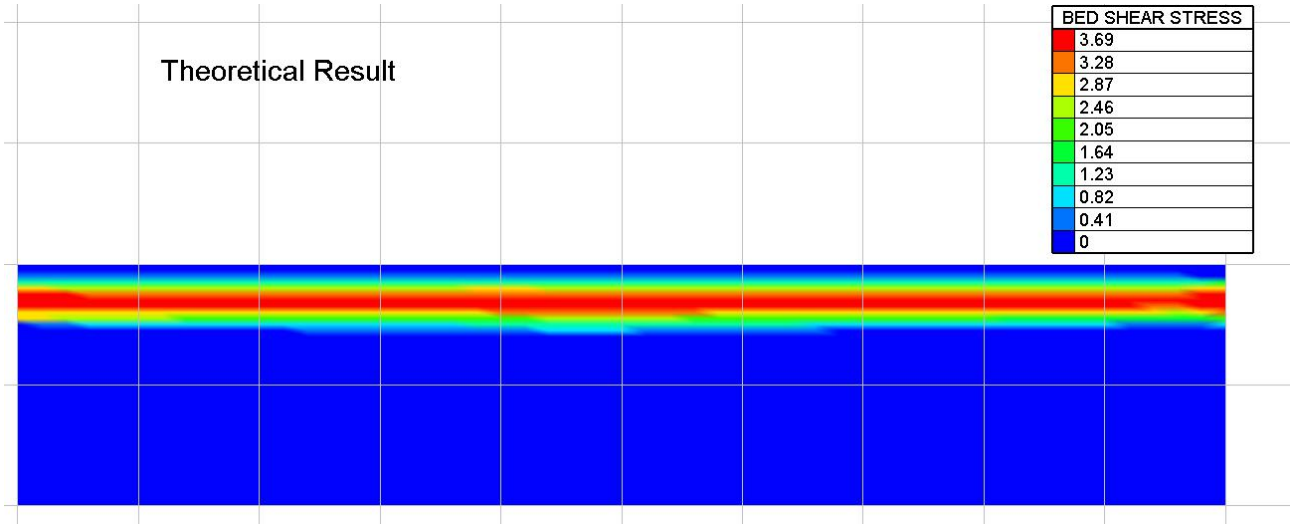
```
/ meme que pour fichier cas Telemac
GRAPHIC PRINTOUT PERIOD = 1
LISTING PRINTOUT PERIOD = 1
NUMBER OF TIME STEPS  = 10
TIME STEP = 10.
/
/ NUMERICAL PARAMETERS (1)
/ ------------------------
/
ZERO = 1e-12
TETA = 0.5
/
/TRANSPORT SOLIDE (1)
/ ____________________
/
BED-LOAD TRANSPORT FORMULA = 5
EFFECT OF WAVES = YES
/ attention seule la deviation est prise en compte
/ dans Koch et Flochstra
SLOPE EFFECT = YES
BETA = 0.
/
MEAN DIAMETER OF THE SEDIMENT = 0.0003
NON COHESIVE BED POROSITY = 0.375
/

```

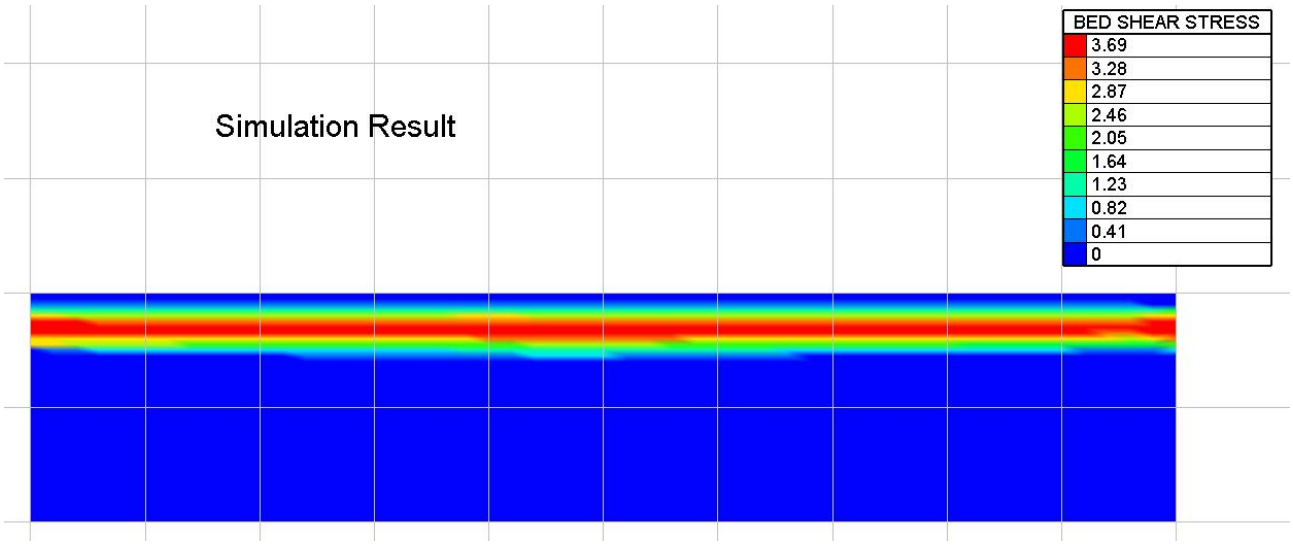
```
================================================================================

 ITERATION      100    TIME: 16 MN  40.0000 S   (     1000.0000 S)
--------------------------------------------------------------------------------

                         ADVECTION STEP
--------------------------------------------------------------------------------

                    DIFFUSION-PROPAGATION STEP
 GMRES (BIEF) :       40 ITERATIONS, RELATIVE PRECISION:   0.8380505E-05
--------------------------------------------------------------------------------

                      BALANCE OF WATER VOLUME
     VOLUME IN THE DOMAIN :    1000866.     M3
     FLUX BOUNDARY    1:    -0.000000     M3/S  ( >0 : ENTERING  <0 : EXITING )
     FLUX BOUNDARY    2:    -51.08085     M3/S  ( >0 : ENTERING  <0 : EXITING )
     FLUX BOUNDARY    3:     49.27144     M3/S  ( >0 : ENTERING  <0 : EXITING )
     RELATIVE ERROR IN VOLUME AT T =        1000.     S :   -0.6863431E-04
--------------------------------------------------------------------------------

                  FINAL BALANCE OF WATER VOLUME

     RELATIVE ERROR CUMULATED ON VOLUME:    0.1982935E-02

     INITIAL VOLUME              :     1000250.     M3
     FINAL VOLUME                :     1000866.     M3
     VOLUME THAT ENTERED THE DOMAIN:     2600.866     M3  ( IF <0 EXIT )
     TOTAL VOLUME LOST           :     1984.653     M3

================================================================================

 ITERATION      100    TIME: 16 MN  40.0000 S     (   1000.0000 S)

 BEDLOAD EQUATION SOLVED IN     1 ITERATIONS

 MASS-BALANCE (IN VOLUME, INCLUDING VOID):
 SUM OF THE EVOLUTIONS :   -0.1011285     M3
 BOUNDARY   1 BEDLOAD FLUX =    0.1557902E-01  ( M3/S  >0 = ENTERING )
 BOUNDARY   2 BEDLOAD FLUX =   -0.1659030E-01  ( M3/S  >0 = ENTERING )
 TOTAL      BEDLOAD FLUX =   -0.1011285E-02  ( M3/S  >0 = ENTERING )
 LOST VOLUME                    :   -0.1127151E-10 M3  ( IF <0 EXIT )

 SUM OF THE CUMULATED EVOLUTIONS :     1.287402
 VOLUME THAT ENTERED THE DOMAIN  :     1.287402     M3  ( IF <0 EXIT )
 LOST VOLUME                    :   -0.3130496E-10 M3  ( IF <0 EXIT )
```

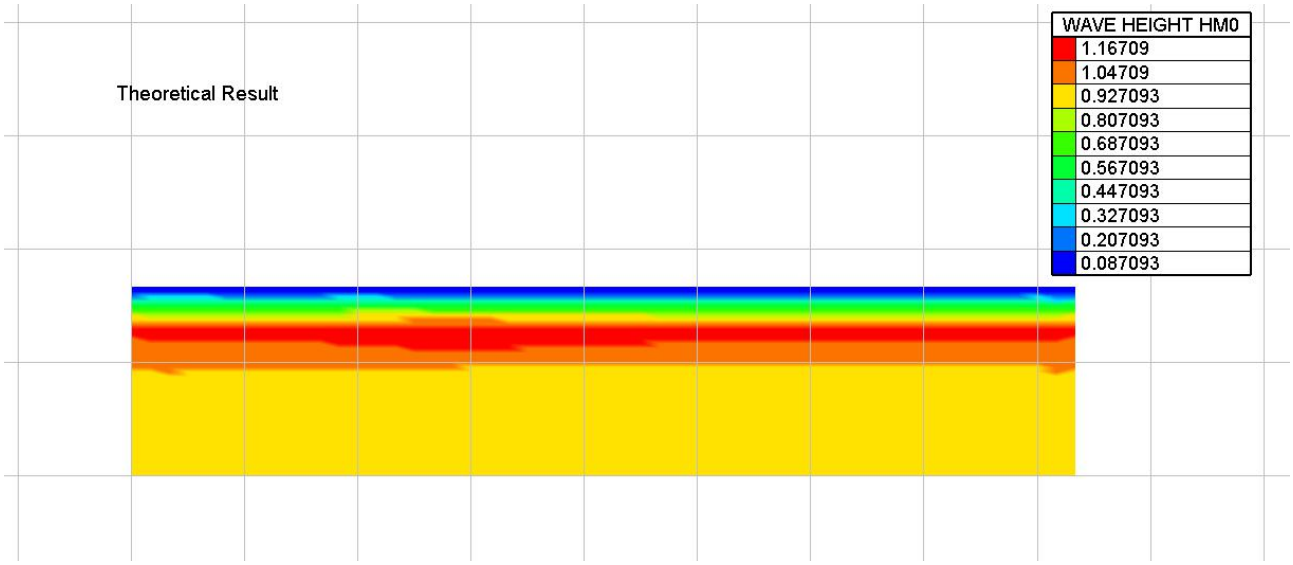
```
 MAXIMAL EVOLUTION       :    0.2182550E-03 NODE  :    211
 MINIMAL EVOLUTION       :   -0.5207454E-03 NODE  :     83
 TOTAL MAXIMAL EVOLUTION :    0.2184226E-02 NODE  :    289
 TOTAL MINIMAL EVOLUTION :   -0.1141621E-01 NODE  :     84

 END OF TIME LOOP

 CALL OF P_EXIT IN ITS VOID VERSION

                   **********************************
                   *       MEMORY CLEANUP           *
                   **********************************

                   ***************************************
                   *    END OF MEMORY ORGANIZATION:      *
                   ***************************************
STOP 0

                   ***************************************
                   *    END OF MEMORY ORGANIZATION:      *
                   ***************************************

 CORRECT END OF RUN

 ELAPSE TIME :
                         17  SECONDS
```

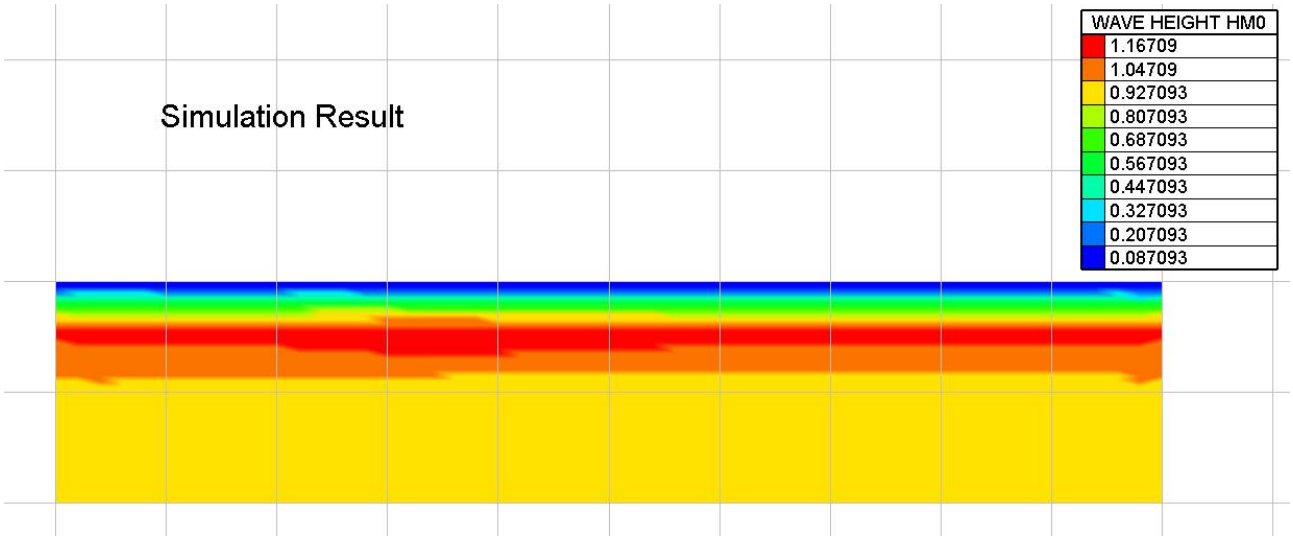


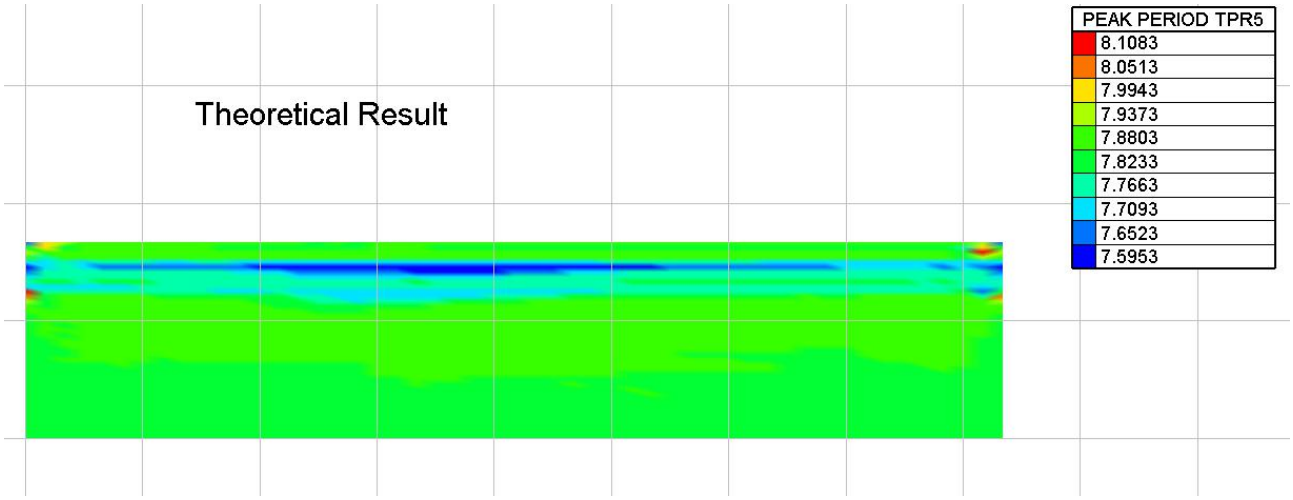

Theoretical Result
VELOCITY UV
0.864
0.768
0.672
0.576
0.48
0.384
0.288
0.192
0.096
0
2D View (14)
0 Days 00:16:40.000
Simulation Result
VELOCITY UV
0.864
0.768
0.672
0.576
0.48
0.384
0.288
0.192
0.096
0
Simulation Result
WATER DEPTH
9.01
8.02
7.03
6.04
5.05
4.06
3.07
2.08
1.09
0.1
Days 00:16:40.000
Theoretical Result
WATER DEPTH
9.01
8.02
7.03
6.04
5.05
4.06
3.07
2.08
1.09
0.1

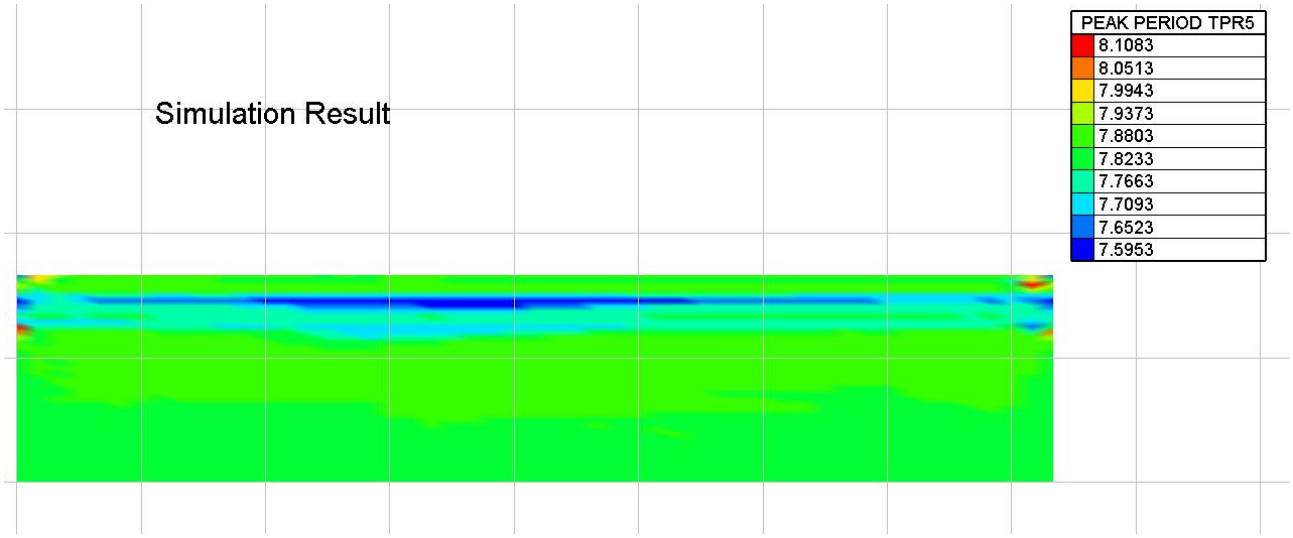

Theoretical Result
BED SHEAR STRESS
3.69
3.28
2.87
2.46
2.05
1.64
1.23
0.82
0.41
0

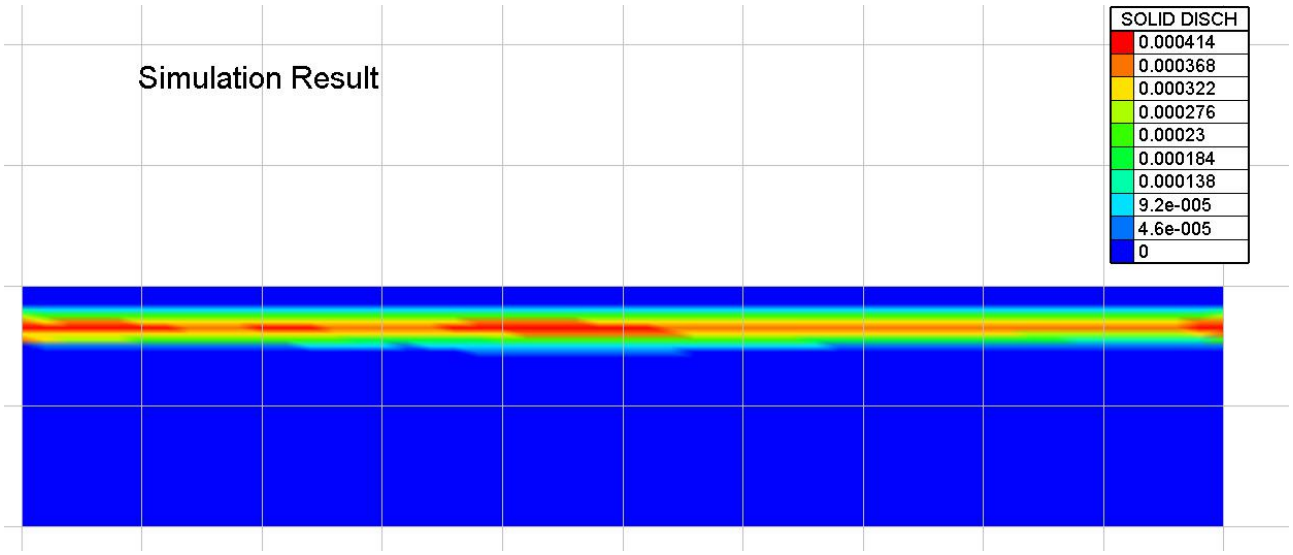

Simulation Result
BED SHEAR STRESS
3.69
3.28
2.87
2.46
2.05
1.64
1.23
0.82
0.41
0

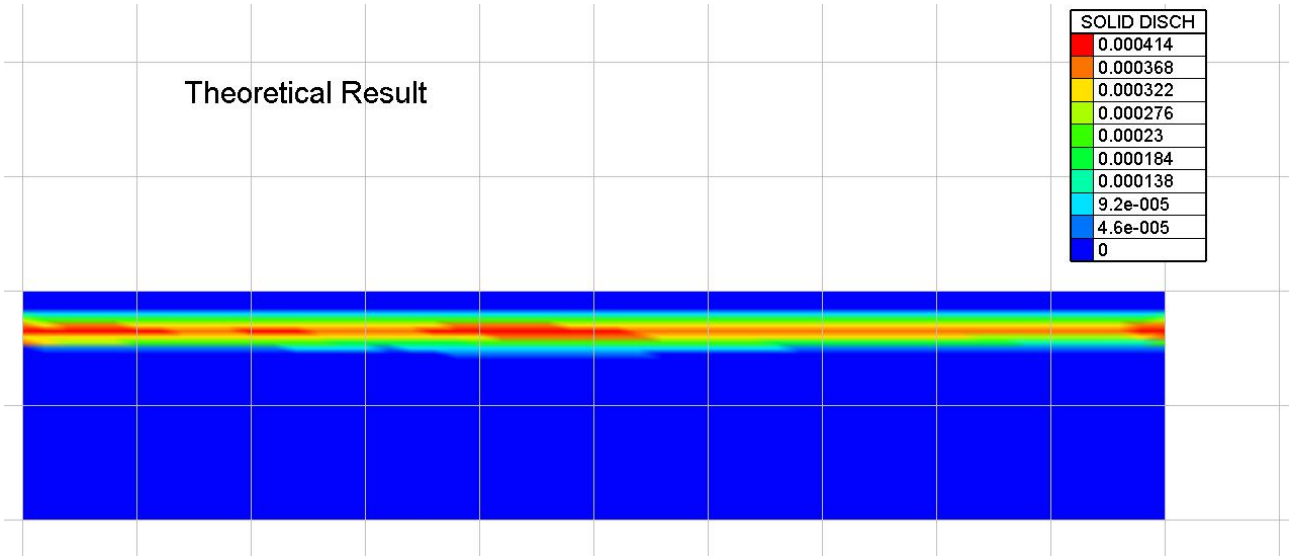

Theoretical Result
WAVE HEIGHT HM0
1.16709
1.04709
0.927093
0.807093
0.687093
0.567093
0.447093
0.327093
0.207093
0.087093

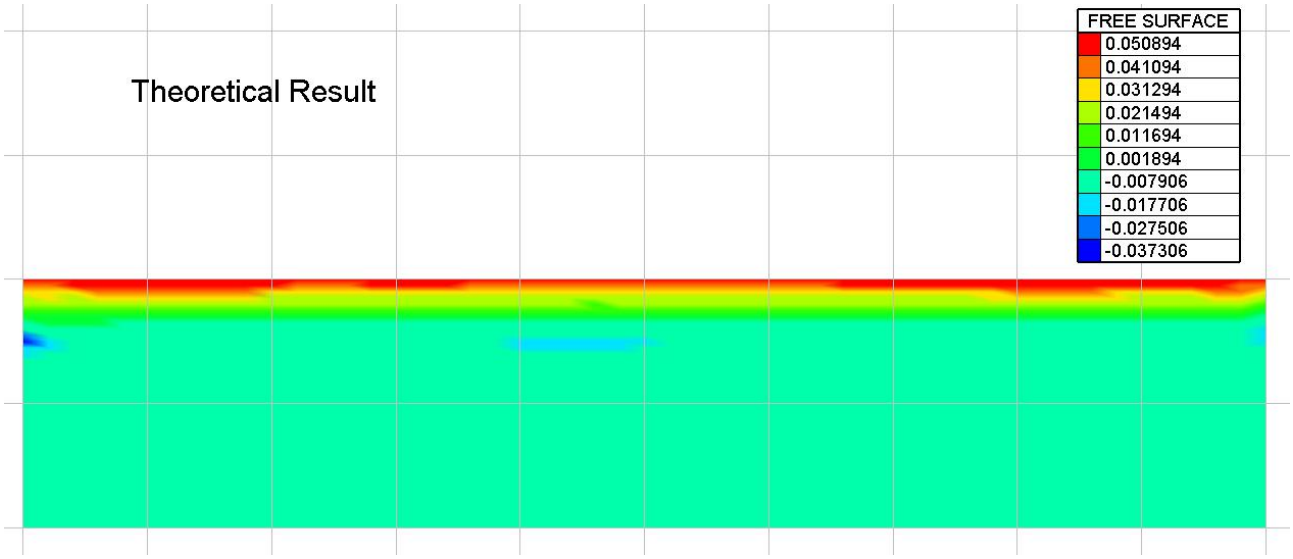

Simulation Result
WAVE HEIGHT HM0
1.16709
1.04709
0.927093
0.807093
0.687093
0.567093
0.447093
0.327093
0.207093
0.087093

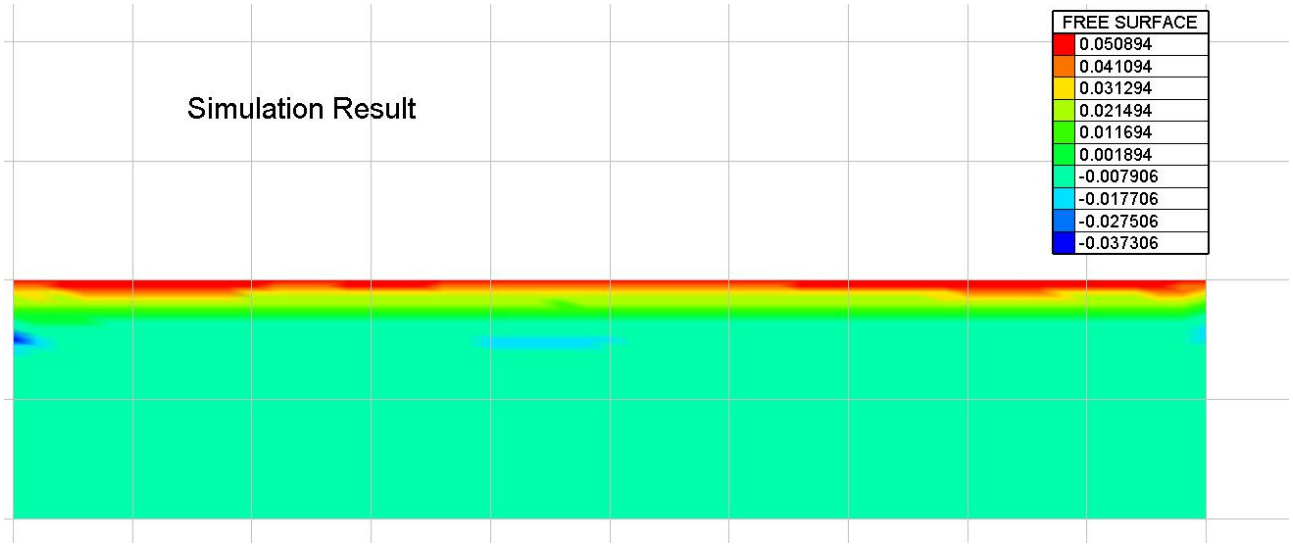

Theoretical Result
PEAK PERIOD TPR5
8.1083
8.0513
7.9943
7.9373
7.8803
7.8233
7.7663
7.7093
7.6523
7.5953

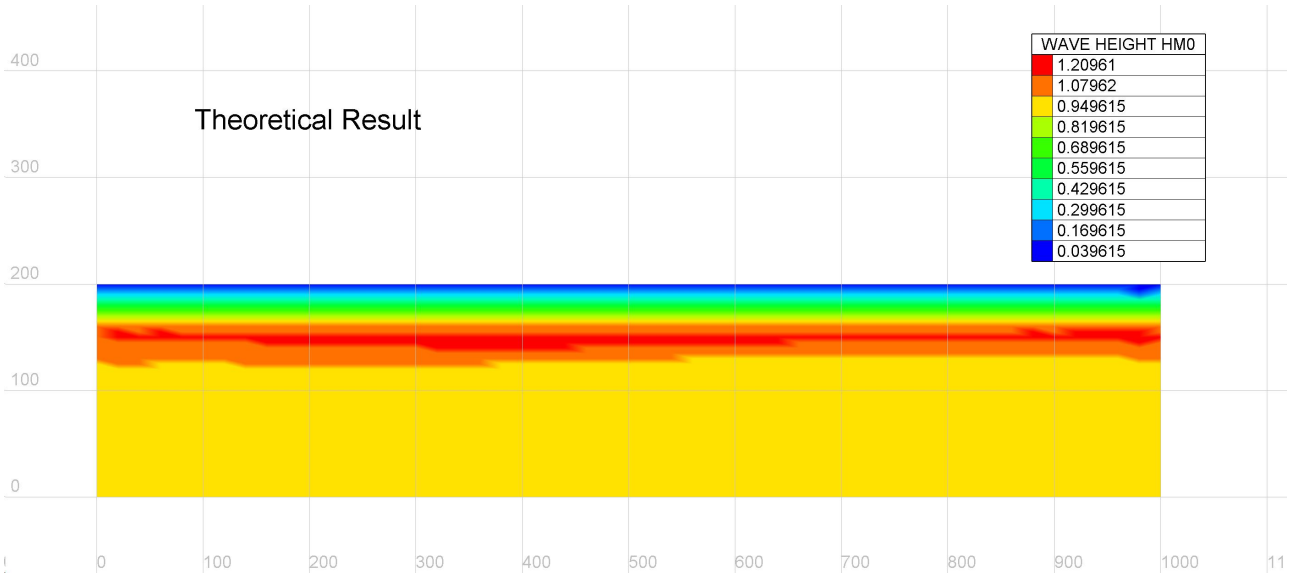

Simulation Result
PEAK PERIOD TPR5
8.1083
8.0513
7.9943
7.9373
7.8803
7.8233
7.7663
7.7093
7.6523
7.5953

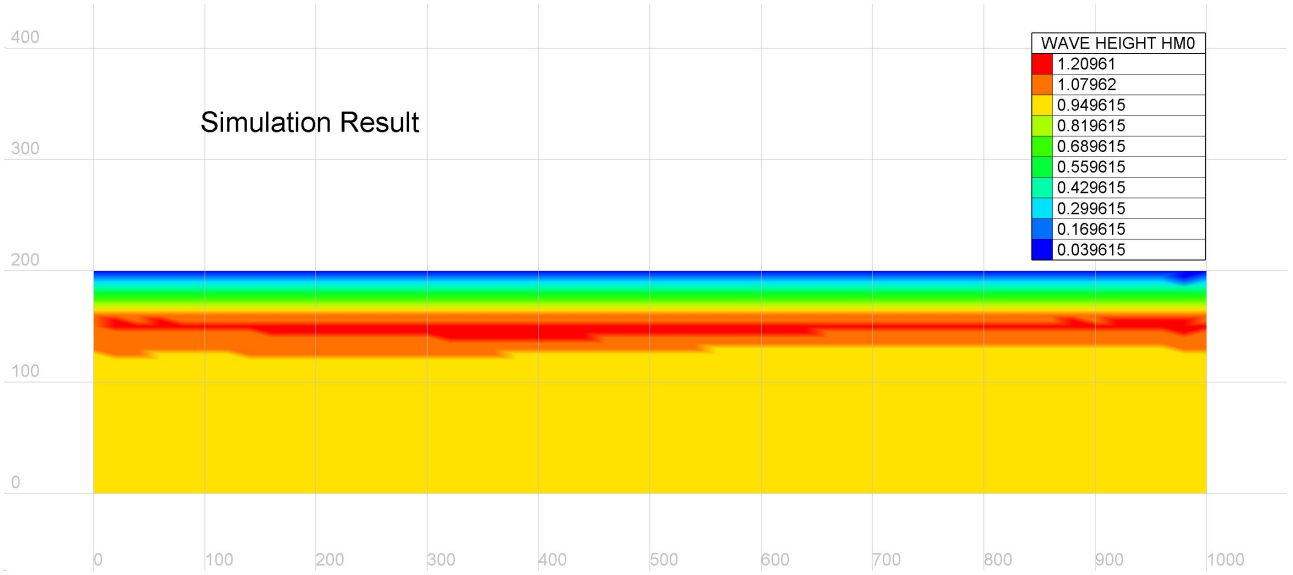

Simulation Result
SOLID DISCH
0.000414
0.000368
0.000322
0.000276
0.00023
0.000184
0.000138
9.2e-005
4.6e-005
0

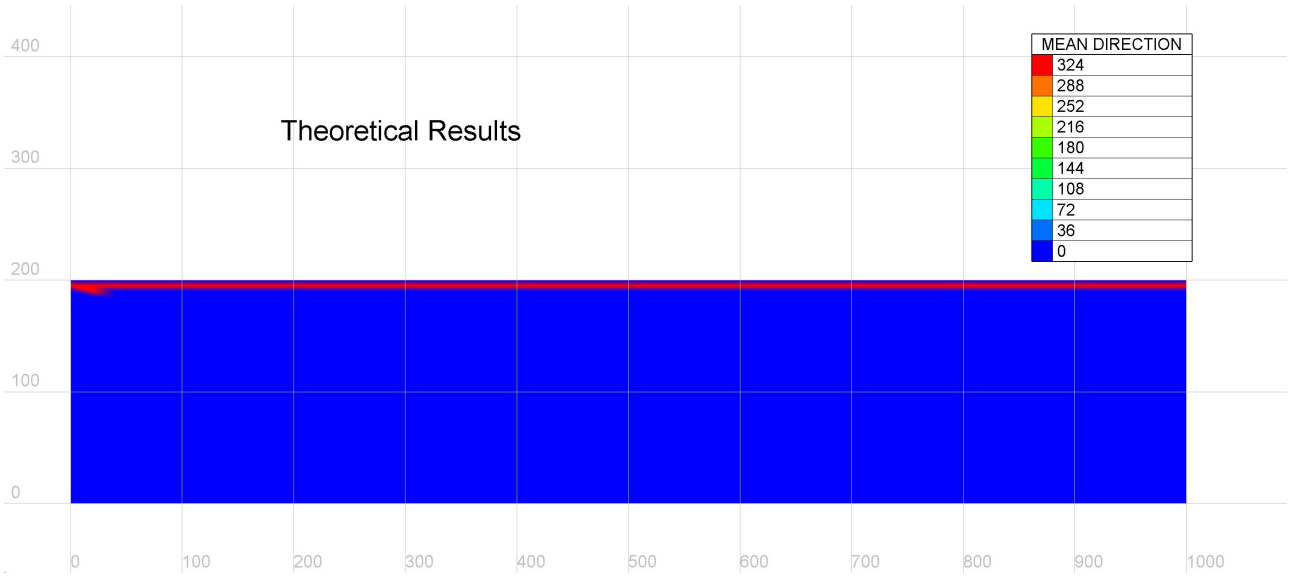

Theoretical Result
SOLID DISCH
0.000414
0.000368
0.000322
0.000276
0.00023
0.000184
0.000138
9.2e-005
4.6e-005
0

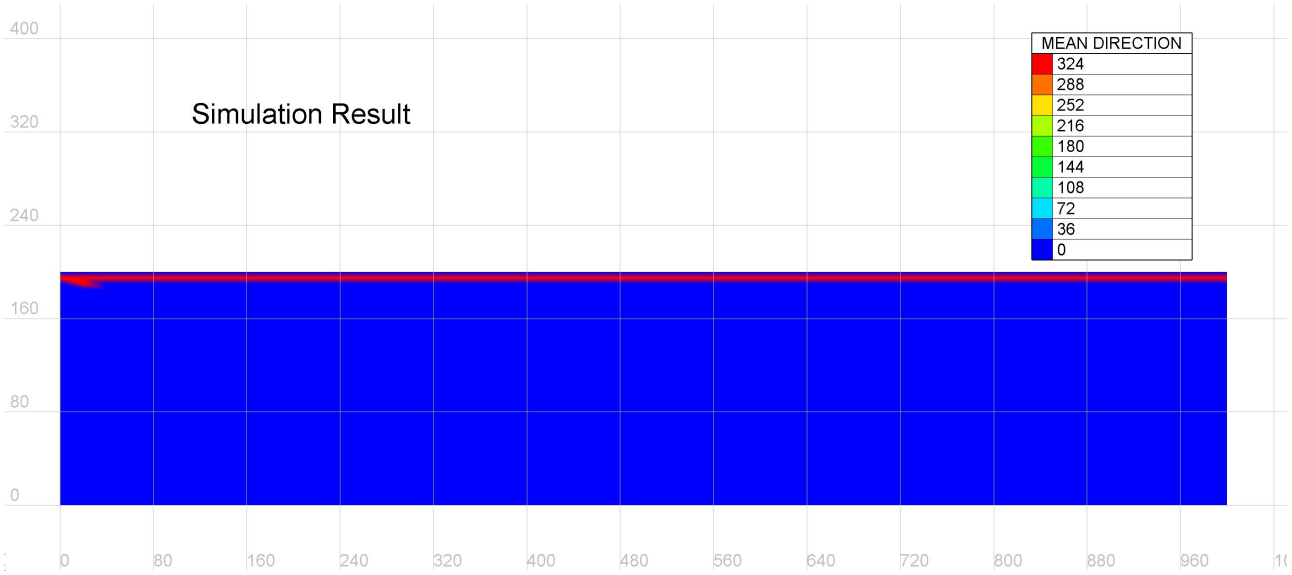

Theoretical Result
FREE SURFACE
0.050894
0.041094
0.031294
0.021494
0.011694
0.001894
-0.007906
-0.017706
-0.027506
-0.037306

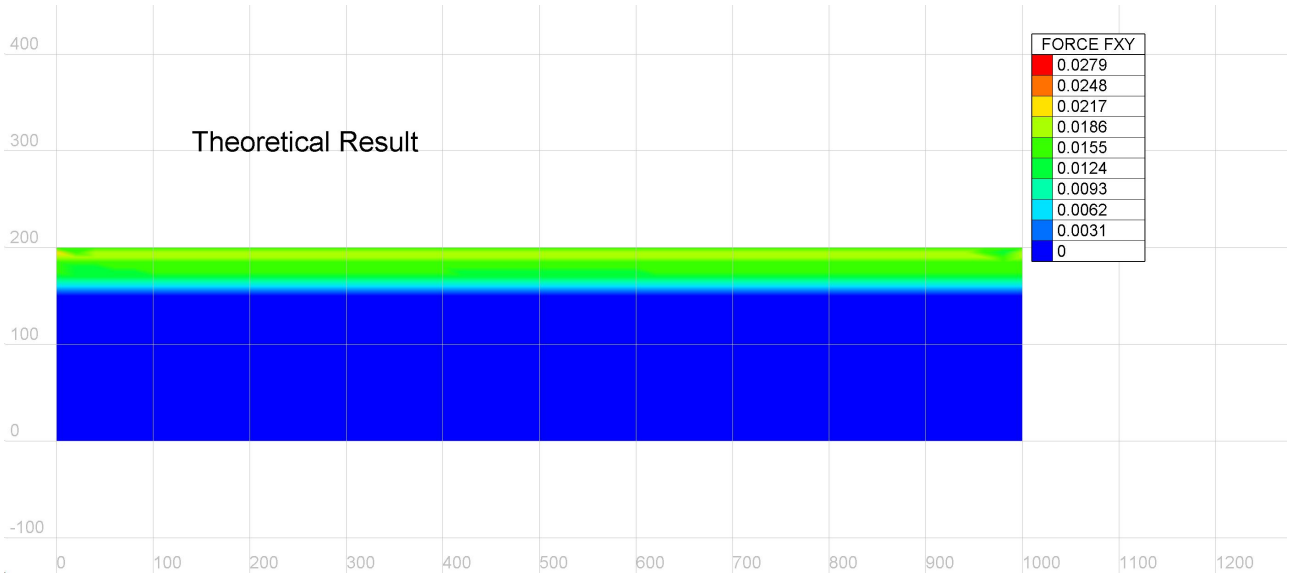

Simulation Result
FREE SURFACE
0.050894
0.041094
0.031294
0.021494
0.011694
0.001894
-0.007906
-0.017706
-0.027506
-0.037306

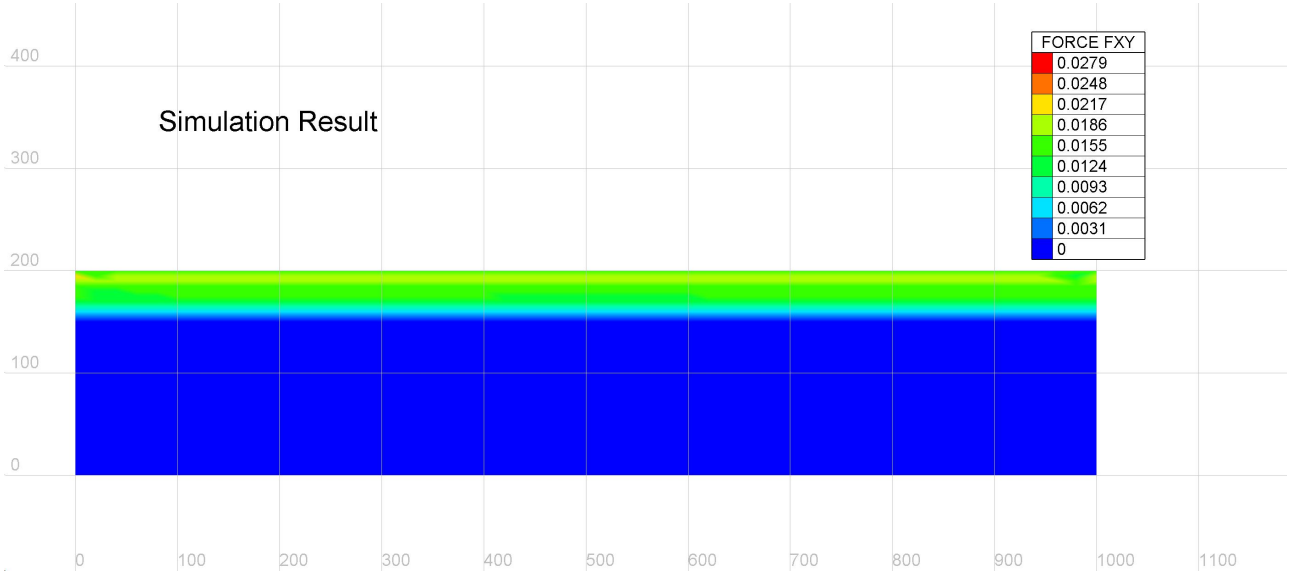

Theoretical Result
WAVE HEIGHT HM0
1.20961
1.07962
0.949615
0.819615
0.689615
0.559615
0.429615
0.299615
0.169615
0.039615
400
300
200
100
0
0
100
200
300
400
500
600
700
800
900
1000


Simulation Result
WAVE HEIGHT HM0
1.20961
1.07962
0.949615
0.819615
0.689615
0.559615
0.429615
0.299615
0.169615
0.039615
400
300
200
100
0
0
100
200
300
400
500
600
700
800
900
1000


Theoretical Results
MEAN DIRECTION
324
288
252
216
180
144
108
72
36
0
400
300
200
100
0
0
100
200
300
400
500
600
700
800
900
1000


Simulation Result
MEAN DIRECTION
324
288
252
216
180
144
108
72
36
0
400
320
240
160
80
0
0
80
160
240
320
400
480
560
640
720
800
880
960


Theoretical Result
FORCE FXY
0.0279
0.0248
0.0217
0.0186
0.0155
0.0124
0.0093
0.0062
0.0031
0
400
300
200
100
0
-100
0
100
200
300
400
500
600
700
800
900
1000
1100
1200


Simulation Result
FORCE FXY
0.0279
0.0248
0.0217
0.0186
0.0155
0.0124
0.0093
0.0062
0.0031
0
400
300
200
100
0
0
100
200
300
400
500
600
700
800
900
1000
1100