Governing Equations in Granular Flow

0.1 Governing Equations in Granular Flow

The governing equations for the mass and momentum balances can be written as:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial z} = 0; \tag{1}$$

0.1.1 Momentum Balance in the x Direction

$$\rho\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial z}\right) - \rho g \sin\theta + \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xz}}{\partial z} = 0; \tag{2}$$

0.1.2 Momentum Balance in the z Direction

$$\rho\left(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial z}\right) = \rho g \cos\theta + \frac{\partial \sigma_{xz}}{\partial x} + \frac{\partial \sigma_{zz}}{\partial z}; \tag{3}$$

where σ is the stress tensor of the material and g is the force due to gravity.

0.2 Non-Dimensionalization

Let L be the characteristic length scale of spatial variations along the slope (in the x direction) and H the typical flow thickness. The shallow-water approximation assumes that the ratio $\epsilon = H/L$ is small:

$$\epsilon = \frac{H}{L} \ll 1; \tag{4}$$

The dimensionless variables are then chosen as follows:

$$x = \tilde{x}L, \tag{5}$$
$$z = \tilde{z}H, \tag{6}$$
$$t = \tilde{t}\frac{L}{U}, \tag{7}$$
$$u = \tilde{u}U, \tag{8}$$
$$v = \tilde{v}\epsilon U, \tag{9}$$
$$\sigma_{xx} = \tilde{\sigma}_{xx}\rho g H, \tag{10}$$
$$\sigma_{zz} = \tilde{\sigma}_{zz}\rho g H, \tag{11}$$
$$\sigma_{xz} = \tilde{\sigma}_{xz}\rho g H, \tag{12}$$

where $U = \sqrt{gH}$ is a typical flow velocity along the slope.
Using these dimensionless variables, the governing equations become:

$$\frac{\partial \tilde{u}}{\partial \tilde{x}} + \frac{\partial \tilde{v}}{\partial \tilde{z}} = 0; \tag{13}$$

$$\epsilon\left(\frac{\partial\tilde{u}}{\partial\tilde{t}}+\tilde{u}\frac{\partial\tilde{u}}{\partial\tilde{x}}+\tilde{v}\frac{\partial\tilde{u}}{\partial\tilde{z}}\right)=\sin\theta+\epsilon\frac{\partial\tilde{\sigma}_{xx}}{\partial\tilde{x}}+\frac{\partial\tilde{\sigma}_{xz}}{\partial\tilde{z}}; \tag{14}$$

$$\epsilon^2\left(\frac{\partial\tilde{v}}{\partial\tilde{t}}+\tilde{u}\frac{\partial\tilde{v}}{\partial\tilde{x}}+\tilde{v}\frac{\partial\tilde{v}}{\partial\tilde{z}}\right)=-\cos\theta+\epsilon\frac{\partial\tilde{\sigma}_{xz}}{\partial\tilde{x}}+\frac{\partial\tilde{\sigma}_{zz}}{\partial\tilde{z}}; \tag{15}$$

For $\epsilon \to 0$, the momentum equation in the vertical direction reduces to $\frac{\partial\tilde{\sigma}_{zz}}{\partial\tilde{z}} = \cos\theta$. Therefore, in the shallow-water approximation, the vertical normal stress is given by the hydrostatic balance. Integrating this equation with the zero-stress condition at the free surface gives:

$$\sigma_{zz}=-\rho g\cos\theta(h(x,t)-z); \tag{16}$$

0.2.1 Depth-Averaged Mass Conservation Equation

$$\frac{\partial h}{\partial t}=-\frac{\partial h\overline{u}}{\partial x}; \tag{17}$$

where $\overline{u}$ is the average depth velocity and h is the thickness of the granular layer.

0.2.2 Depth-Averaged Momentum Conservation Equation

$$\frac{\partial h\overline{u}}{\partial t}+\alpha\frac{\partial h\overline{u}^2}{\partial x}=gh\cos\theta\left(\tan\theta-\frac{\tau_b}{\rho gh\cos\theta}-k\frac{\partial h}{\partial x}\right); \tag{18}$$

where τ_b is the basal shear stress and θ is the slope at which the granular fluid is flowing down.

0.2.3 Assumptions

These equations are derived under the following assumptions:

1. An incompressible medium.
2. The shallow-layer approximation.
3. An assumption on the shape of the velocity profile through the parameter α.
4. Proportionality between the normal stresses along the x and z directions, $\sigma_{xx} = k\sigma_{zz}$.

0.2.4 Friction Law

$$\tau_b=\mu_b\rho gh\cos\theta; \tag{19}$$

where $\rho gh\cos\theta$ is the basal normal stress and μ_b is an effective coefficient of friction between the flowing layer and the bottom.

$$\mu_b(\overline{u},h)=\mu_1+\frac{(\mu_2-\mu_1)}{(2I_0h\sqrt{\phi gh\cos\theta+5d\overline{u}})}(5d\overline{u}); \tag{20}$$

The basal friction is simply written as $\mu_b = \mu(I_b)$, where I_b is the inertial number evaluated at the plane. To express I_b as a function of $\overline{u}$ and h, we assume that the velocity profile is locally at equilibrium.

$$I_b = \frac{5d\overline{u}}{2h\sqrt{g\phi h\cos\theta}}; \tag{21}$$

0.2.5 Simplification of the Momentum Conservation Equation

After substituting the value of τ_b and μ_b, we get:

$$\frac{\partial h\overline{u}}{\partial t} + \alpha\frac{\partial h\overline{u}^2}{\partial x} = gh\cos\theta\left(\tan\theta - \mu_b - k\frac{\partial h}{\partial x}\right); \tag{22}$$

Further simplification:

$$\overline{u}\frac{\partial h}{\partial t} + h\frac{\partial \overline{u}}{\partial t} + \alpha\frac{\partial h\overline{u}^2}{\partial x} = gh\cos\theta\left(\tan\theta - \mu_b - k\frac{\partial h}{\partial x}\right); \tag{23}$$

0.3 PDEPE

$$\frac{\partial h}{\partial t} = -u\frac{\partial h}{\partial x} - h\frac{\partial u}{\partial x}; \tag{24}$$

After substituting the above equation in equation (22):

$$h\frac{\partial u}{\partial t} = -u\left(-u\frac{\partial h}{\partial x} - h\frac{\partial u}{\partial x}\right) - \alpha\frac{\partial h\overline{u}^2}{\partial x} + gh\cos\theta\left(\tan\theta - \mu_b - k\frac{\partial h}{\partial x}\right); \tag{25}$$

After substituting $\mu_b = \frac{\tau_b}{\rho gh\cos\theta}$ in the above equation and simplifying, we get:

$$h\frac{\partial u}{\partial t} = (u^2\frac{\partial h}{\partial x} + uh\frac{\partial u}{\partial x}) - \alpha(u^2\frac{\partial h}{\partial x} + 2uh\frac{\partial u}{\partial x}) + gh\cos\theta\left(\tan\theta - \frac{\tau_b}{\rho gh\cos\theta} - k\frac{\partial h}{\partial x}\right); \tag{26}$$

$$h\frac{\partial u}{\partial t} = gh\cos\theta\left(\tan\theta - \frac{\tau_b}{\rho gh\cos\theta} - k\frac{\partial h}{\partial x}\right) + u^2(1-\alpha)\frac{\partial h}{\partial x} + uh(1-2\alpha)\frac{\partial u}{\partial x}; \tag{27}$$

PDEPE variables Case 1:

1. $c = h$;
2. $m = 0$;
3. $f = 0$;
4. $s = gh\cos\theta\left(\tan\theta - \frac{\tau_b}{\rho gh\cos\theta} - k\frac{\partial h}{\partial x}\right) + u^2(1-\alpha)\frac{\partial h}{\partial x} + uh(1-2\alpha)\frac{\partial u}{\partial x}$;

$$\frac{\partial h}{\partial t} = -\frac{\partial h\overline{u}}{\partial x}; \tag{28}$$

$$c = 1; \quad s = -h\frac{\partial u}{\partial x} - u\frac{\partial h}{\partial x}.; \quad f = 0; \quad m = 0; \tag{29}$$

0.4 Algorithms

0.4.1 Granular Flow Simulation Algorithm

The following algorithm describes the process of simulating granular flow using the PDE solver in MATLAB:

0.4.2 Function `pdex2pde` Algorithm

The following algorithm describes the function `pdex2pde` used in the MATLAB simulation to define the PDE system:

0.4.3 Function `pdex2ic` Algorithm

The following algorithm describes the function `pdex2ic` used in the MATLAB simulation to specify the initial conditions for the PDE system:

0.4.4 Function `pdex2bc` Algorithm

The following algorithm describes the function `pdex2bc` used in the MATLAB simulation to specify the boundary conditions for the PDE system:

0.5 Method 1: PDEPE Solver

The PDEPE solver in MATLAB is designed to solve partial differential equations (PDEs) of the form:

$$c(x,t,u,\frac{\partial u}{\partial x})\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f(x,t,u,\frac{\partial u}{\partial x})\right) + s(x,t,u,\frac{\partial u}{\partial x})$$

where $u(x,t)$ is the dependent variable, x and t are the independent variables, and m is a constant that specifies the symmetry of the problem. The functions c, f, and s are defined by the specific PDE system being solved.

0.5.1 Description of PDEPE Variables

For the equations provided:

1. c corresponds to the coefficient of $\frac{\partial u}{\partial t}$.
2. m represents the symmetry of the problem. In your case, $m = 0$ indicates no symmetry.
3. f is the flux term, which generally depends on u and its spatial derivative $\frac{\partial u}{\partial x}$.
4. s is the source term, which includes all other terms of the PDE that are independent of the time derivative.

Algorithm 1: Granular Flow Simulation

[1] **Initialization:** Set global variables:
$h_{\text{initial}}, v_{x\text{initial}}, \text{bin}_h, \text{bin}_t, I_{\text{not}}, \mu_s, \mu_m, d, \theta, \phi, g, \alpha, k, h_{\text{initial0}}, v_{x\text{initial0}}, L_x$

Set simulation time step $ts = 10$

Set system properties and parameters:

$g = 9.81$ (gravity)

$\theta = 30$ (slope angle)

$d = 0.001$

$L_x = 1000 \cdot d$

Rheological model parameters: $I_{\text{not}} = 1$

$\mu_s = 0.1$

$\mu_m = 0.75$

$\phi = 0.58$

$\alpha = \frac{5}{4}$

$k = 1$

Initial values: $\text{bin}_h = 100$

$\text{bin}_t = 100$

$h_{\text{initial}} = 39.d \cdot \text{ones}(1, \text{bin}_h)$

$v_{x\text{initial}} = 5 \cdot \text{ones}(1, \text{bin}_h)$

$h_{\text{initial}}(1) = 39 \cdot d$

$v_{x\text{initial}}(1) = 5$

$h_{\text{initial0}} = 39 \cdot d$

$v_{x\text{initial0}} = 5$

Simulation:

Initialize spatial and time vectors: x and t

Initialize counter count = 1 and time = 0

Set ODE solver options with tolerances and maximum step size

Create output directory if it does not exist

while *time* < 500 **do**

Set symmetry parameter $m = 0$

Solve PDE using `pdepe` with functions `pdex2pde`, `pdex2ic`, `pdex2bc`, and options

if *Exception occurs* **then**

else

Display warning and exit loop

Extract solutions for velocity and height

Update count and time

Update h_{initial} and $v_{x\text{initial}}$ with latest values

Write data to file with updated filename

Plot and save the figure showing height and velocity

Algorithm 2: Function `pdex2pde`

[1] **Inputs:** x (spatial variable)
t (time variable)
u (vector of unknowns [velocity, height])
$DuDx$ (vector of derivatives [velocity gradient, height gradient])
Global Variables: $I_{\text{not}}, \mu_s, \mu_m, d, \theta, \phi, g, \alpha, k$
Extract Variables:
$v_x = u(1)$ (velocity)
$h = u(2)$ (height)
$\frac{dv_x}{dy} = DuDx(1)$ (velocity gradient)
$\frac{dh}{dy} = DuDx(2)$ (height gradient)
Compute Intermediate Quantities: $I = \frac{5 \cdot d \cdot v_x}{2 \cdot h \cdot \sqrt{\phi \cdot g \cdot h \cdot \cos(\theta)}}$
$\mu_b = \mu_s + \frac{\mu_m - \mu_s}{1 + \frac{I_{\text{not}}}{I}}$
Define PDEPE Coefficients:
For the height equation:
$c1 = h$

$s1 = g \cdot h \cdot \cos(\theta) \cdot (\tan(\theta) - \mu_b) - k \cdot g \cdot h \cdot \cos(\theta) \cdot \frac{dh}{dx} + v_x^2 \cdot (1 - \alpha) \cdot \frac{dh}{dx} + v_x \cdot h \cdot (1 - 2 \cdot \alpha) \cdot \frac{dv_x}{dx}$
$F1 = 0$
For the velocity equation:
$C2 = 1$
$S2 = -h \cdot \frac{dv_x}{dx} - v_x \cdot \frac{dh}{dx}$
$F2 = 0$
Outputs:
$C = [C1; C2]$
$S = [S1; S2]$
$F = [F1; F2]$

Algorithm 3: Function `pdex2ic`

[1] **Inputs:** x (spatial variable)
Global Variables: $\text{bin\_h}, L_x, \text{vx\_initial}, \text{h\_initial}, \text{time}$
Compute Bin Index:
$s = \frac{x - 0}{\frac{L_x}{\text{bin\_h} - 1}} + 1$
$s = \text{round}(s)$ (bin index)
Extract Initial Values:
$f_0 = \text{h\_initial}(s)$ (initial height)
$v_0 = \text{vx\_initial}(s)$ (initial velocity)
Define Initial Condition Vector: $u_0 = [v_0; f_0]$
Output: u_0 (initial condition vector)

Algorithm 4: Function `pdex2bc`

[1] **Inputs:** x_l (left boundary position) u_l (solution vector at left boundary) x_r (right boundary position) u_r (solution vector at right boundary) t (time)
Global Variables: $\text{h\_initial0}, \text{vx\_initial0}$
Define Left Boundary Conditions: $p_l = [u_l(1) - \text{vx\_initial0}; u_l(2) - \text{h\_initial0}]$
$q_l = [0; 0]$
Define Right Boundary Conditions:
$p_r = [0; 0]$
$q_r = [1; 1]$
Output: p_l (left boundary conditions)
q_l (left boundary fluxes)
p_r (right boundary conditions)
q_r (right boundary fluxes)

0.5.2 PDEPE Variables in the Given Equations

Given the derived equations:

$$h\frac{\partial u}{\partial t} = gh\cos\theta\left(\tan\theta - \frac{\tau_b}{\rho gh\cos\theta} - k\frac{\partial h}{\partial x}\right) + u^2(1-\alpha)\frac{\partial h}{\partial x} + uh(1-2\alpha)\frac{\partial u}{\partial x}$$

This can be identified with the PDEPE form where:

$$\begin{aligned} c(x,t,u,\frac{\partial u}{\partial x}) &= h, \\ m &= 0, \\ f(x,t,u,\frac{\partial u}{\partial x}) &= 0, \\ s(x,t,u,\frac{\partial u}{\partial x}) &= gh\cos\theta\left(\tan\theta - \frac{\tau_b}{\rho gh\cos\theta} - k\frac{\partial h}{\partial x}\right) + u^2(1-\alpha)\frac{\partial h}{\partial x} + uh(1-2\alpha)\frac{\partial u}{\partial x}. \end{aligned}$$

For the second PDE:

$$\frac{\partial h}{\partial t} = -\frac{\partial h\overline{u}}{\partial x}$$

This corresponds to:

$$\begin{aligned} c(x,t,u,\frac{\partial u}{\partial x}) &= 1, \\ m &= 0, \\ f(x,t,u,\frac{\partial u}{\partial x}) &= 0 \\ s(x,t,u,\frac{\partial u}{\partial x}) &= -h\frac{\partial u}{\partial x} - u\frac{\partial h}{\partial x}. \end{aligned}$$

0.5.3 Explanation of p, q, f, s, and c

In the context of PDEPE, the variables are as follows:

- $c(x, t, u, \frac{\partial u}{\partial x})$: The coefficient of the time derivative $\frac{\partial u}{\partial t}$. It controls the scaling of the time evolution.
- m: A symmetry parameter where $m = 0$ denotes Cartesian coordinates, $m = 1$ denotes cylindrical symmetry, and $m = 2$ denotes spherical symmetry.
- $f(x, t, u, \frac{\partial u}{\partial x})$: The flux term, which typically depends on u and its spatial derivative $\frac{\partial u}{\partial x}$. It represents the flow of the quantity u.
- $s(x, t, u, \frac{\partial u}{\partial x})$: The source term, representing any additional sources or sinks in the system, not associated with the time derivative.

In your equations, these variables are explicitly defined as:

Equation 1: $c = h, \quad m = 0, \quad f = 0, \quad s = gh\cos\theta\left(\tan\theta - \frac{\tau_b}{\rho gh\cos\theta} - k\frac{\partial h}{\partial x}\right) + u^2(1-\alpha)\frac{\partial h}{\partial x} + uh(1$

Equation 2: $c = 1, \quad m = 0, \quad f = -h\overline{u}, \quad s = 0.$

0.6 MATLAB Code

```
clear all; close all; clc;
global h_initial vx_initial bin_h bin_t I_not mu_s mu_m d theta phi g alpha k h_initial0 vx_initial0 L_x
ts = 10; % Reduce time step

%% System variables
g = 9.81;
theta = 25;

%% System properties
d = 0.001;
L_x = 1000*d;

%% Rheological model parameters
I_not = 1;
mu_s = 0.1;
mu_m = 0.75;
phi = 0.58;
alpha = 5/4; % Bagnold velocity profile
k = 1; % Neglecting normal stress differences

%% Initial values
bin_h = 100;
bin_t = 100;
h_initial =39*d*ones(1, bin_h);
vx_initial = 5*ones(1, bin_h);
h_initial(1) = 39* d;
vx_initial(1) = 5;
h_initial0 = 39* d;
vx_initial0 = 5;

%% Initialization
y = linspace(0, L_x, bin_h);
```

```
t = linspace(0, ts, bin_t);

count = 1;
time = 0;

% ODE solver options
options = odeset('RelTol', 1e-4, 'AbsTol', 1e-6, 'MaxStep', 1e-2); % Adjusted tolerances and max step

output_dir = 'Time';
if ~exist(output_dir, 'dir')
    status = mkdir(output_dir);
    if status == 0
        error('Failed to create directory: %s', output_dir);
    else
        disp(['Directory created: ', output_dir]);
    end
else
    disp(['Directory already exists: ', output_dir]);
end

%% Simulation loop
while time < 500
    m = 0; % Symmetry parameter for PDEPE

    % Solve the PDE
    try
        sol = pdepe(m, @pdex2pde, @pdex2ic, @pdex2bc, y, t, options);
    catch ME
        warning('Time integration has failed. Solution is available at requested time points up to t=%.2f.\nError: %s', time, ME.message);
        break;
    end

    % Extract solutions
    velocity = sol(:,:,1);
    height = sol(:,:,2);

    count = count + 1;
    time = time + ts;
    disp(['Current simulation time: ', num2str(time)]);

    %% Properties calculation
    vx = velocity(end, :); % Update velocity, vx after time step ts
    h = height(end, :);

    %% Restricting f to be in between 0 to 1
    h_initial = h;
    vx_initial = vx;

    % Update data
    data = [y', vx', h'];

    % Write data to file
    filename = fullfile(output_dir, sprintf('properties-%05d.txt', count)); % Adjusted filename format
```

```
    disp(['Writing data to file: ', filename]);
    try
        writematrix(data, filename, 'Delimiter', ' ');
    catch ME
        error('Failed to write file: %s\nError: %s', filename, ME.message)
            ;
    end

    % Plot and save the figure
    figure;
    subplot(2, 1, 1);
    plot(y, h,'linewidth',2);
    title(sprintf('Height at time = %.2f', time));
    xlabel('Position y');
    ylabel('Height h');

    subplot(2, 1, 2);
    plot(y, vx,'linewidth',2);
    title(sprintf('Velocity at time = %.2f', time));
    xlabel('Position y');
    ylabel('Velocity vx');

    saveas(gcf, fullfile(output_dir, sprintf('plot-%05d.png', count)));
    close
end
```

Boundary condition function of PDEPE

```
    function [pl ,ql ,pr ,qr] = pdex2bc(xl ,ul ,xr ,ur ,t)
    global time h_initial0 vx_initial0
    pl = [ul(1) - vx_initial0; ul(2) - h_initial0];
    % pl = [ul(1); ul(2)];
    ql = [0; 0];

    pr = [0; 0];
    qr = [1; 1];
end
```

Initial condition Function of PDEPE

```
    function u0 = pdex2ic(x)
    global bin_h L_x vx_initial h_initial time

    s = (x - 0) / ((L_x) / (bin_h - 1)) + 1;
    s = round(s); % bin index

    f0 = h_initial(s);
    v0 = vx_initial(s);
    u0 = [v0; f0];
end
```

PDEPE Function

```
    function [C, F, S] = pdex2pde(x, t, u, DuDx)
    global I_not mu_s mu_m d theta phi g alpha k

    vx = u(1);
    h = u(2);
```

```
    dvx_dy = DuDx(1);
    dh_dy = DuDx(2);

    I = 5 * d * vx / (2 * h * sqrt(phi * g * h * cosd(theta)));

    mu_b = mu_s + (mu_m - mu_s) / (1 + I_not / I);

    C1 = h ;
    S1 =  g * h * cosd(theta) * (tand(theta) - mu_b) - k * g * h * cosd(
       theta) * dh_dy+vx^2*(1-alpha)*dh_dy+vx*h*(1-2*alpha)*dvx_dy;
    F1 = 0;

    C2 = 1;
    S2 = -h*dvx_dy-vx*dh_dy;
    F2 = 0;

    C = [C1; C2];
    S = [S1; S2];
    F = [F1; F2];
end
```

0.7 steady state code

```
clear all; close all; clc;
global h_initial V_0 I_not mu_s mu_m d phi g alpha k L_x

% Parameters
g = 9.81;

% System properties
d = 0.001;
L_x = 1000* d;

% Rheological model parameters
I_not = 0.434;
mu_s = tand(20.16);
mu_m = tand(37.65);
phi = 0.58;
alpha = 5/4; % Bagnold velocity profile
k = 1; % Neglecting normal stress differences

% Discretization parameters
bin_h = 100;
h_initial = 25 * d * ones(1, bin_h);
V_0 = 5*sqrt(g*d) * ones(1, bin_h);
m = h_initial .* V_0; % Mass per unit width

% Time-stepping parameters
dx = L_x / bin_h; % Step size in space

% Theta values to simulate
theta_values = [25, 22, 24, 26, 28, 30];

% Initialize figure
```

```matlab
figure;
hold on;

% Color map for different theta values
colors = lines(length(theta_values));

% Loop over theta values
results = struct();
for t_idx = 1:length(theta_values)
    theta = theta_values(t_idx); % Current theta in degrees

    % Initialize variables
    V_new = V_0;
    h_new = h_initial;

    % Initial calculation for rheological properties
    I = 5 * d .* V_new ./ (2 .* h_new .* sqrt(phi * g .* h_new * cosd(
        theta)));
    mu_b = mu_s + (mu_m - mu_s) ./ (1 + I_not ./ I);

    % ODE solving loop over the spatial domain
    for i = 1:bin_h
        if i == 1
            % Skip update for the first point (initial condition)
            continue;
        end

        % Define the ODE for the current spatial step
        ode = @(x, V) V * cosd(theta) * (tand(theta) - mu_b(i)) ...
                   / (alpha * V^3 - cosd(theta) * m(i));

        % Solve ODE using numerical method (e.g., ode45)
        [x, V_temp] = ode45(@(x, V) ode(x, V), [0, dx], V_new(i-1));

        % Update velocity
        V_new(i) = V_temp(end);

        % Update height using continuity equation
        h_new(i) = (h_initial(i) * V_0(i)) / V_new(i);

        % Recalculate I and mu_b for the next iteration
        I(i) = 5 * d .* V_new(i) ./ (2 .* h_new(i) .* sqrt(phi * g .*
            h_new(i) * cosd(theta)));
        mu_b(i) = mu_s + (mu_m - mu_s) ./ (1 + I_not ./ I(i));
    end

    % Store data for current theta
    results(t_idx).theta = theta;
    results(t_idx).V = V_new;
    results(t_idx).h = h_new;

    % Plot results for velocity
    subplot(2, 1, 1);
    plot(linspace(0, L_x, bin_h), V_new, 'LineWidth', 2, 'Color', colors(
        t_idx, :));
    hold on;

```

```
    % Plot results for height
    subplot(2, 1, 2);
    plot(linspace(0, L_x, bin_h), h_new, 'LineWidth', 2, 'Color', colors(t_idx, :));
    hold on;
end

% Customize velocity profile subplot
subplot(2, 1, 1);
xlabel('length(m)');
ylabel('Velocity V (m/s)');
title('Velocity Profile');
legend(arrayfun(@(t) sprintf('\\theta = %d  ', t), theta_values, 'UniformOutput', false), ...
       'Location', 'best');
grid on;

% Customize height profile subplot
subplot(2, 1, 2);
xlabel('length (m)');
ylabel('Height h (m)');
title('Height Profile');
legend(arrayfun(@(t) sprintf('\\theta = %d  ', t), theta_values, 'UniformOutput', false), ...
       'Location', 'best');
grid on;

% Save results to a file
save('flow_results.mat', 'results');
disp('Simulation complete. Results saved to flow_results.mat.');
```

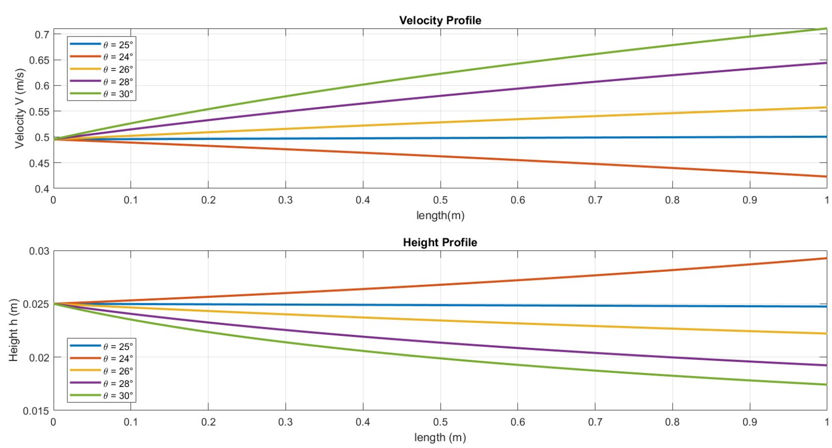

Figure 1: at $v = 5\sqrt{gd}$ and h=25*d steady state plot

Newton-Raphson Method

The Newton-Raphson method is an iterative technique used to find the roots of a real-valued function. Given a function $f(x)$ and its derivative $f'(x)$, the method uses the following iterative formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

where:

- x_n is the current approximation,
- x_{n+1} is the next approximation,
- $f(x_n)$ is the value of the function at x_n,
- $f'(x_n)$ is the value of the derivative of the function at x_n.

Procedure

1. **Initial Guess**: Start with an initial guess x_0 for the root.
2. **Iteration**: Apply the iterative formula to compute $x_1, x_2, \ldots$ until convergence.
3. **Convergence**: The process is repeated until the difference between successive approximations is smaller than a predetermined tolerance level, i.e., $|x_{n+1} - x_n| < \epsilon$, where ϵ is a small positive number.

0.8 Depth-Averaged Mass Conservation and Momentun Conservation Equations

$$\frac{\partial h}{\partial t} = -\frac{\partial h\overline{u}}{\partial x}; \tag{30}$$

$$h\frac{\partial \overline{u}}{\partial t} = -\overline{u}\frac{\partial h}{\partial t} - \alpha\frac{\partial h\overline{u}^2}{\partial x} + gh\cos\theta\left(\tan\theta - \mu_b - k\frac{\partial h}{\partial x}\right); \tag{31}$$

0.8.1 Residual and Jacobian Matrix Formation

After discretizing the given equations with respect to x and t, we obtain two nonlinear equations at each grid point. Given $Nx = n$ and $Nt = m$, this results in a total of $2 \times n \times m$ nonlinear equations and $2 \times n \times m$ unknowns. These equations can be solved using an iterative method like the Newton-Raphson method, which requires forming a residual vector and a Jacobian matrix.

Residual Vector Formation

The residual vector $\mathbf{R}$ is formed by evaluating the left-hand sides of the nonlinear equations at each grid point:

$$R_1(i,j) = h_{i,j+1} - h_{i,j-1} + 2\Delta t\left[h_{i,j}\left(\frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x}\right) + u_{i,j}\left(\frac{h_{i+1,j} - h_{i-1,j}}{2\Delta x}\right)\right]$$

$$R_2(i,j) = \left(h_{i,j}\left(\frac{u_{i,j+1}-u_{i,j-1}}{2\Delta t}\right)\right)+\left(u_{i,j}\left(\frac{h_{i,j+1}-h_{i,j-1}}{2\Delta t}\right)\right)+\alpha\left[2h_{i,j}u_{i,j}\left(\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}\right)+u_{i,j}^2\left(\frac{h_{i+1,j}}{2}\right.\right.$$

The residual vector $\mathbf{R}$ is then constructed by stacking all $R_1(i,j)$ and $R_2(i,j)$ values for each $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

Jacobian Matrix Formation

The Jacobian matrix $\mathbf{J}$ consists of partial derivatives of the residuals with respect to the unknown variables. For each residual $R_1(i,j)$ and $R_2(i,j)$, the elements of the Jacobian matrix are calculated as:

$$J_{(i,j),(k,l)} = \frac{\partial R_1(i,j)}{\partial h_{k,l}}, \quad \frac{\partial R_1(i,j)}{\partial u_{k,l}}, \quad \frac{\partial R_2(i,j)}{\partial h_{k,l}}, \quad \frac{\partial R_2(i,j)}{\partial u_{k,l}}$$

The Jacobian matrix $\mathbf{J}$ has a size of $2nm \times 2nm$, and its elements are populated by the derivatives of the residuals with respect to the unknowns. The differentiation is performed with respect to the variables $h_{i,j}$ and $u_{i,j}$.

Differentiation of the Equations

Differentiating the first equation $R_1(i,j)$ with respect to $h_{k,l}$ and $u_{k,l}$:

$$\frac{\partial R_1(i,j)}{\partial h_{k,l}} = \begin{cases} 1 & \text{if } k = i, l = j+1 \text{ or } l = j-1 \\ -\frac{2\Delta t}{2\Delta x} \cdot u_{i,j} & \text{if } k = i+1 \text{ or } k = i-1 \\ \frac{2\Delta t}{2\Delta x} \cdot u_{i,j} & \text{if } k = i \text{ and } l = j \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\partial R_1(i,j)}{\partial u_{k,l}} = \begin{cases} -\frac{2\Delta t}{2\Delta x} \cdot h_{i,j} & \text{if } k = i+1 \text{ or } k = i-1 \\ \frac{2\Delta t}{2\Delta x} \cdot h_{i,j} & \text{if } k = i \text{ and } l = j \\ 0 & \text{otherwise} \end{cases}$$

Differentiating the second equation $R_2(i,j)$ with respect to $h_{k,l}$ and $u_{k,l}$ involves applying the product rule due to the terms with both $h_{i,j}$ and $u_{i,j}$:

$$\frac{\partial R_2(i,j)}{\partial h_{k,l}} = (\text{similar process as above, involving each term of } R_2(i,j))$$

$$\frac{\partial R_2(i,j)}{\partial u_{k,l}} = (\text{similar process as above, involving each term of } R_2(i,j))$$

The differentiation is done term by term, considering the dependencies on $h_{i,j}$ and $u_{i,j}$ and applying the chain rule where necessary.

Final System of Equations

Once the residual vector $\mathbf{R}$ and Jacobian matrix $\mathbf{J}$ are formed, the Newton-Raphson update is applied iteratively:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{J}^{-1}\mathbf{R}$$

where $\mathbf{x}$ is the vector of unknowns $h_{i,j}$ and $u_{i,j}$.

0.9 Boundary conditions

0.9.1 Boundary condition at x=0

$$v = v_{initial}$$

(32)

$$h = h_{initial} \tag{33}$$

0.9.2 Boundary condition at x=L

$$\frac{\partial v}{\partial x} = 0; \tag{34}$$

$$\frac{\partial h}{\partial x} = 0; \tag{35}$$

Algorithm 5: Newton-Raphson Update for Boundary Conditions and Residuals Calculation

Input: Matrices h, u; Integers n, m; Scalars Δt, Δx, α, g, ϕ, θ, μ_{b1}, μ_{b2}, $I0$, d, k, h_{bc}, u_{bc}

Output: Updated matrices h and u

Initialize:

$size \leftarrow 2 \cdot n \cdot m$

$F \leftarrow \mathbf{0}_{size}$

$J \leftarrow$ Jacobian matrix for boundary conditions and residuals

$delta\_X \leftarrow$ Vector of Newton-Raphson updates

$x \leftarrow$ Vector of spatial coordinates for plotting

while *Convergence criteria not met* **do**

  Calculate residuals F

  Calculate Jacobian matrix J

  $delta\_X \leftarrow -J \cdot F$

  for $i \leftarrow 1$ **to** n **do**

    for $j \leftarrow 1$ **to** m **do**

      $h(i,j) \leftarrow h(i,j) + delta\_X[(i-1) \cdot m + j]$

      $u(i,j) \leftarrow u(i,j) + delta\_X[n \cdot m + (i-1) \cdot m + j]$

  Display 'Updated h:'

  Display h

  Display 'Updated u:'

  Display u

  Plot h and u:

  - Create a figure with two subplots.
  - Plot h vs. x in the first subplot.
  - Plot u vs. x in the second subplot.
  - Add labels, titles, and legends as needed.

  Check for convergence criteria.

return h, u

Algorithm 6: Algorithm for Boundary Conditions and Residuals Calculation

Input: Integers n, m; Matrices h, u; Scalars h_{bc}, u_{bc}, d, $g_\phi \cos\theta$, $I0$, μ_{b1}, μ_{b2}, k, g, α, θ, Δt, Δx

Output: Vector F of residuals

Initialize:

$size \leftarrow 2 \cdot n \cdot m$

$F \leftarrow \mathbf{0}_{size}$

$idx \leftarrow 1$

for $i \leftarrow 1$ **to** n **do**

  for $j \leftarrow 1$ **to** m **do**

    if $i == 1$ **then**

      $h_{i-1} \leftarrow h_{bc}$

      $u_{i-1} \leftarrow u_{bc}$

    else

      $h_{i-1} \leftarrow h(i-1, j)$

      $u_{i-1} \leftarrow u(i-1, j)$

    if $i == n$ **then**

      $h_{i+1} \leftarrow h(i, j)$

      $u_{i+1} \leftarrow u(i, j)$

    else

      $h_{i+1} \leftarrow h(i+1, j)$

      $u_{i+1} \leftarrow u(i+1, j)$

    if $j == 1$ **then**

      $h_{j-1} \leftarrow h_{bc}$

      $u_{j-1} \leftarrow u_{bc}$

    else

      $h_{j-1} \leftarrow h(i, j-1)$

      $u_{j-1} \leftarrow u(i, j-1)$

    if $j == m$ **then**

      $h_{j+1} \leftarrow h(i, j)$

      $u_{j+1} \leftarrow u(i, j)$

    else

      $h_{j+1} \leftarrow h(i, j+1)$

      $u_{j+1} \leftarrow u(i, j+1)$

    $h_i \leftarrow h(i, j)$

    $u_i \leftarrow u(i, j)$

    $I_b \leftarrow \frac{5 \cdot d \cdot u_i}{2 \cdot h_i \cdot \sqrt{g_\phi \cos\theta \cdot h_i}}$

    $\mu_b \leftarrow \mu_{b1} + \left(\frac{\mu_{b2} - \mu_{b1}}{I0/I_b}\right) + 1$

    $F[idx] \leftarrow h_{j+1} - h_{j-1} + 2 \cdot \Delta t \cdot \left(\frac{u_{j+1} - 2 \cdot u_j + u_{j-1}}{\Delta x^2} + \text{RHS term}\right)$

    $F[idx+1] \leftarrow u_{j+1} - u_{j-1} + \Delta t \cdot \left(\frac{g}{\Delta x} \cdot (h_{j+1} - h_{j-1}) - \text{RHS term}\right)$

    $idx \leftarrow idx + 2$

return F

Algorithm 7: Jacobian Calculation

Input: Matrices h, u; Integers n, m; Scalars Δt, Δx, α, g, ϕ, θ, μ_{b1}, μ_{b2}, $I0$, d, k, h_{bc}, u_{bc}

Output: Jacobian matrix J

Initialize:

$J \leftarrow$ sparse matrix of size $2nm \times 2nm$

$g_\phi \cos\theta \leftarrow g \cdot \phi \cdot \cos(\theta)$

$idx \leftarrow 1$

for $i \leftarrow 1$ **to** n **do**

  for $j \leftarrow 1$ **to** m **do**

    $h_i \leftarrow h(i,j)$

    $u_i \leftarrow u(i,j)$

    $I_b \leftarrow \frac{5 \cdot d \cdot u_i}{2 \cdot h_i \cdot \sqrt{g_\phi \cos\theta \cdot h_i}}$

    $\mu_b \leftarrow \mu_{b1} + \left(\frac{\mu_{b2}-\mu_{b1}}{I0/I_b}\right) + 1$

    $dF1\_dh_{ij} \leftarrow 2 \cdot \Delta t \cdot \left(\frac{u_{i+1}-u_{i-1}}{2 \cdot \Delta x}\right)$

    $dF1\_dh_{ij-1} \leftarrow -1$

    $dF1\_dh_{ij+1} \leftarrow 1$

    $dF1\_dh_{i-1j} \leftarrow -2 \cdot \Delta t \cdot \left(\frac{u_i}{2 \cdot \Delta x}\right)$

    $dF1\_du_{ij} \leftarrow 2 \cdot \Delta t \cdot \left(\frac{h_{i+1}-h_{i-1}}{2 \cdot \Delta x}\right)$

    $J(idx, idx) \leftarrow dF1\_dh_{ij}$

    $J(idx, idx-1) \leftarrow dF1\_dh_{ij-1}$

    $J(idx, idx+1) \leftarrow dF1\_dh_{ij+1}$

    $J(idx, idx-m) \leftarrow dF1\_dh_{i-1j}$

    $J(idx, idx+m) \leftarrow dF1\_dh_{i+1j}$

    $J(idx, idx+n \cdot m) \leftarrow dF1\_du_{ij}$

    $idx \leftarrow idx+1$

    $dF2\_dh_{ij} \leftarrow \frac{u_i \cdot (u_{i+1}-u_{i-1})}{2 \cdot \Delta t}$

    $dF2\_du_{ij} \leftarrow \frac{h_{j+1}-h_{j-1}}{2 \cdot \Delta t}$

    $dF2\_du_{i-1j} \leftarrow -\frac{h_{j+1}-h_{j-1}}{2 \cdot \Delta t}$

    $dF2\_du_{i+1j} \leftarrow -\frac{h_{j+1}-h_{j-1}}{2 \cdot \Delta t}$

    $dF2\_dh_{j-1} \leftarrow -\frac{u_i^2}{2 \cdot \Delta t}$

    $dF2\_dh_{j+1} \leftarrow -\frac{u_i^2}{2 \cdot \Delta t}$

    $dF2\_du_{i-1j} \leftarrow -\frac{h_{j+1}-h_{j-1}}{2 \cdot \Delta x}$

    $dF2\_du_{i+1j} \leftarrow -\frac{h_{j+1}-h_{j-1}}{2 \cdot \Delta x}$

    $J(idx, idx) \leftarrow dF2\_dh_{ij}$

    $J(idx, idx+n \cdot m) \leftarrow dF2\_du_{ij}$

    $J(idx, idx-m) \leftarrow dF2\_dh_{i-1j}$

    $J(idx, idx+m) \leftarrow dF2\_dh_{i+1j}$

    $J(idx, idx-n \cdot m-m) \leftarrow dF2\_du_{i-1j}$

    $J(idx, idx-n \cdot m+m) \leftarrow dF2\_du_{i+1j}$

    $idx \leftarrow idx+1$

return J

0.10 MATLAB Code

```
% Parameters
phi = 0.58;
alpha = 5/4;
g = 9.81;
theta = 25;
mu_b1 = tand(20.16);
mu_b2 = tand(37.65);
I0 =  0.434;
d = 0.001;
k = 1.0;
L = 5000 * d;
Nx = 50;
delta_x = L / (Nx - 1);
x = linspace(0, L, Nx);
T = 10;
Nt = 100;
delta_t = T / (Nt - 1);
t = linspace(0, T, Nt);

n = Nx;
m = Nt;
h = 39 * d * ones(n, m); % Initial values of h
u = 5 * ones(n, m); % Initial values of u
h_bc = 39 * d;
u_bc = 5;

% Calculate residuals
F = calculate_residuals(h, u, n, m, delta_t, delta_x, alpha, g, phi, theta
    , mu_b1, mu_b2, I0, d, k, h_bc, u_bc);

% Calculate Jacobian matrix
J = calculate_jacobian(h, u, n, m, delta_t, delta_x, alpha, g, phi, theta,
    mu_b1, mu_b2, I0, d, k, h_bc, u_bc);

% Newton-Raphson update
delta_X = -J^(-1) *F;


for i = 1:n-1
    for j = 1:m
        h(i+1, j) = h(i, j) +delta_X((i - 1) * m + j);
        u(i+1, j) = u(i, j) + delta_X(n * m + (i - 1) * m + j);
    end
end

% Display updated h and u
disp('Updated h:');

disp('Updated u:');
disp(u);

% After the loop for updating h and u:

% Specify the directory to save the plots
```

```
output_dir = 'Time';

% Create the folder if it doesn't exist
if ~exist(output_dir, 'dir')
    mkdir(output_dir);
end

% Loop through all time steps (from t = 1 to T)
for t_idx = 1:m % m is the number of time steps
    % Extract the appropriate column for the current time step
    figure;

    % Plot h vs x
    subplot(2, 1, 1);
    plot(x, h(:, t_idx));
    xlabel('x');
    ylabel('h');
    title(['h vs x at t = ' num2str(t(t_idx)) ' sec']);

    % Plot u vs x
    subplot(2, 1, 2);
    plot(x, u(:, t_idx));
    xlabel('x');
    ylabel('u');
    title(['u vs x at t = ' num2str(t(t_idx)) ' sec']);

    % Save the figure in the specified folder
    saveas(gcf, fullfile(output_dir, ['h_u_t' num2str(t(t_idx)) '.png']));
        % Save as PNG
end
```

Calculating Residual Matrix Function

```
    function F = calculate_residuals(h, u, n, delta_t, delta_x, alpha, g,
       phi, theta, mu_b1, mu_b2, I0, d, k, h_bc, u_bc)
    F = zeros(2*n*n, 1);
    g_phi_cos_theta = g * phi * cosd(theta);
    idx = 1;
    for i = 1:n
        for j = 1:n
            if i-1 == 0
                h_im1 = h_bc;
                u_im1 = u_bc;
            else
                h_im1 = h(i-1, j);
                u_im1 = u(i-1, j);
            end
            if i+1 == n+1
                h_ip1 = h(i, j);
                u_ip1 = u(i, j);
            else
                h_ip1 = h(i+1, j);
                u_ip1 = u(i+1, j);
            end
            if j-1 == 0
                h_jm1 = h_bc;
                u_jm1 = u_bc;
```

```
            else
                h_jm1 = h(i, j-1);
                u_jm1 = u(i, j-1);
            end
            if j+1 == n+1
                h_jp1 = h(i, j);
                u_jp1 = u(i, j);
            else
                h_jp1 = h(i, j+1);
                u_jp1 = u(i, j+1);
            end

            hi = h(i,j);
            ui = u(i,j);
            Ib = (5 * d * ui) / (2 * hi * sqrt(g_phi_cos_theta * hi));
            mu_b = mu_b1 + (mu_b2 - mu_b1) /  (I0 /Ib)+1;

            F(idx) = h_jp1 - h_jm1 + 2*delta_t*(hi*(u_ip1 - u_im1)/(2*delta_x)) + 2*delta_t*(ui*(h_ip1 - h_im1)/(2*delta_x));
            idx = idx + 1;

            F(idx) = hi*(u_jp1 - u_jm1)/(2*delta_t) + ui*(h_jp1 - h_jm1)/(2*delta_t) ...
                     + alpha*(2*hi*ui*(u_ip1 - u_im1)/(2*delta_x) + ui^2*(h_ip1 - h_im1)/(2*delta_x)) ...
                     - g*hi*cosd(theta)*(tand(theta) - mu_b - k*(h_ip1 - h_im1)/(2*delta_x));
            idx = idx + 1;
        end
    end
end
```

Calculating Jacobian Matrix Function

```
    function J = calculate_jacobian(h, u, n, delta_t, delta_x, alpha, g, phi, theta, mu_b1, mu_b2, I0, d, k, h_bc, u_bc)
    J = zeros(2*n*n, 2*n*n);
    g_phi_cos_theta = g * phi * cosd(theta);
    idx = 1;
    for i = 1:n
        for j = 1:n
            if i-1 == 0
                h_im1 = h_bc;
                u_im1 = u_bc;
            else
                h_im1 = h(i-1, j);
                u_im1 = u(i-1, j);
            end
            if i+1 == n+1
                h_ip1 = h(i, j);
                u_ip1 = u(i, j);
            else
                h_ip1 = h(i+1, j);
                u_ip1 = u(i+1, j);
            end
            if j-1 == 0
                h_jm1 = h_bc;
```

```
                u_jm1 = u_bc;
            else
                h_jm1 = h(i, j-1);
                u_jm1 = u(i, j-1);
            end
            if j+1 == n+1
                h_jp1 = h(i, j);
                u_jp1 = u(i, j);
            else
                h_jp1 = h(i, j+1);
                u_jp1 = u(i, j+1);
            end

            hi = h(i,j);
            ui = u(i,j);
            Ib = (5 * d * ui) / (2 * hi * sqrt(g_phi_cos_theta * hi));
            mu_b = mu_b1 + ((mu_b2 - mu_b1) / (I0/Ib))+1;

            % Calculate partial derivatives for F1
            dF1_dhij = 2*delta_t*(u_ip1 - u_im1)/(2*delta_x);
            dF1_dhijm1 = -1;
            dF1_dhijp1 = 1;
            dF1_dhim1j = -2*delta_t*(ui/(2*delta_x));
            dF1_dhip1j = 2*delta_t*(ui/(2*delta_x));
            dF1_duij = 2*delta_t*(h_ip1 - h_im1)/(2*delta_x);
            dF1_duim1j = -2*delta_t*(hi/(2*delta_x));
            dF1_duip1j = 2*delta_t*(hi/(2*delta_x));

            % Fill in Jacobian for F1
            J(idx, (i-1)*n + j) = dF1_dhij;
            if j-1 > 0
                J(idx, (i-1)*n + j-1) = dF1_dhijm1;
            end
            if j+1 <= n
                J(idx, (i-1)*n + j+1) = dF1_dhijp1;
            end
            if i-1 > 0
                J(idx, (i-2)*n + j) = dF1_dhim1j;
            end
            if i+1 <= n
                J(idx, i*n + j) = dF1_dhip1j;
            end
            J(idx, n*n + (i-1)*n + j) = dF1_duij;
            if i-1 > 0
                J(idx, n*n + (i-2)*n + j) = dF1_duim1j;
            end
            if i+1 <= n
                J(idx, n*n + i*n + j)= dF1_duip1j;
            end
            idx = idx + 1;

            % Calculate partial derivatives for F2
            dF2_dhij = (u_jp1 - u_jm1)/(2*delta_t) + alpha*(2*u(i,j)*(u_ip1 - u_im1)/(2*delta_x) + u(i,j)^2/(2*delta_x)) - g*cosd(theta)*(-mu_b - k*(h_ip1 - h_im1)/(2*delta_x));
            dF2_dhijm1 = -u(i,j)/(2*delta_t);
            dF2_dhijp1 = u(i,j)/(2*delta_t);
```

```
            dF2_dhim1j = -alpha*(2*u(i,j)*(u_ip1 - u_im1)/(2*delta_x) + u(
                i,j)^2/(2*delta_x));
            dF2_dhip1j = alpha*(2*u(i,j)*(u_ip1 - u_im1)/(2*delta_x) + u(i
                ,j)^2/(2*delta_x));
            dF2_duij = hi*(h_jp1 - h_jm1)/(2*delta_t) + 2*alpha*(hi*(u_ip1
                 - u_im1)/(2*delta_x));
            dF2_duijm1 = -hi/(2*delta_t);
            dF2_duijp1 = hi/(2*delta_t);
            dF2_duim1j = -alpha*2*hi*u(i,j)/(2*delta_x);
            dF2_duip1j = alpha*2*hi*u(i,j)/(2*delta_x);

            % Fill in Jacobian for F2
            J(idx, (i-1)*n + j) = dF2_dhij;
            if j-1 > 0
                J(idx, (i-1)*n + j-1) = dF2_dhijm1;
            end
            if j+1 <= n
                J(idx, (i-1)*n + j+1) = dF2_dhijp1;
            end
            if i-1 > 0
                J(idx, (i-2)*n + j) = dF2_dhim1j;
            end
            if i+1 <= n
                J(idx, i*n + j) = dF2_dhip1j;
            end
            J(idx, n*n + (i-1)*n + j) = dF2_duij;
            if j-1 > 0
                J(idx, n*n + (i-1)*n + j-1) = dF2_duijm1;
            end
            if j+1 <= n
                J(idx, n*n + (i-1)*n + j+1) = dF2_duijp1;
            end
            if i-1 > 0
                J(idx, n*n + (i-2)*n + j) = dF2_duim1j;
            end
            if i+1 <= n
                J(idx, n*n + i*n + j) = dF2_duip1j;
            end
            idx = idx + 1;
        end
    end
end
```

0.11 Plots at different value of h and v at theta equal to 25

0.11.1 newton rapson plot

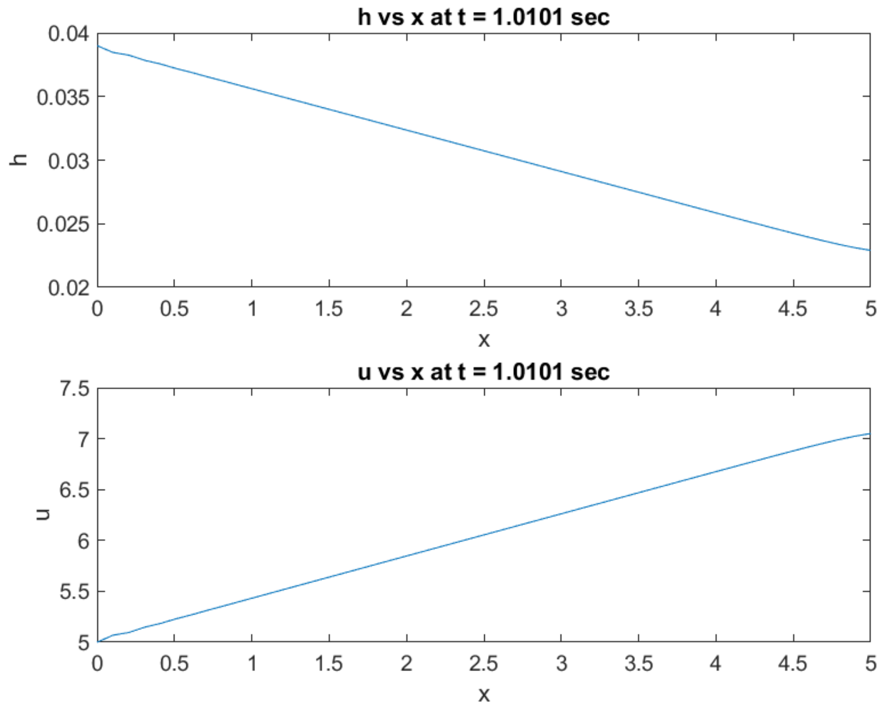

Figure 2: at v=5 and h=39*d

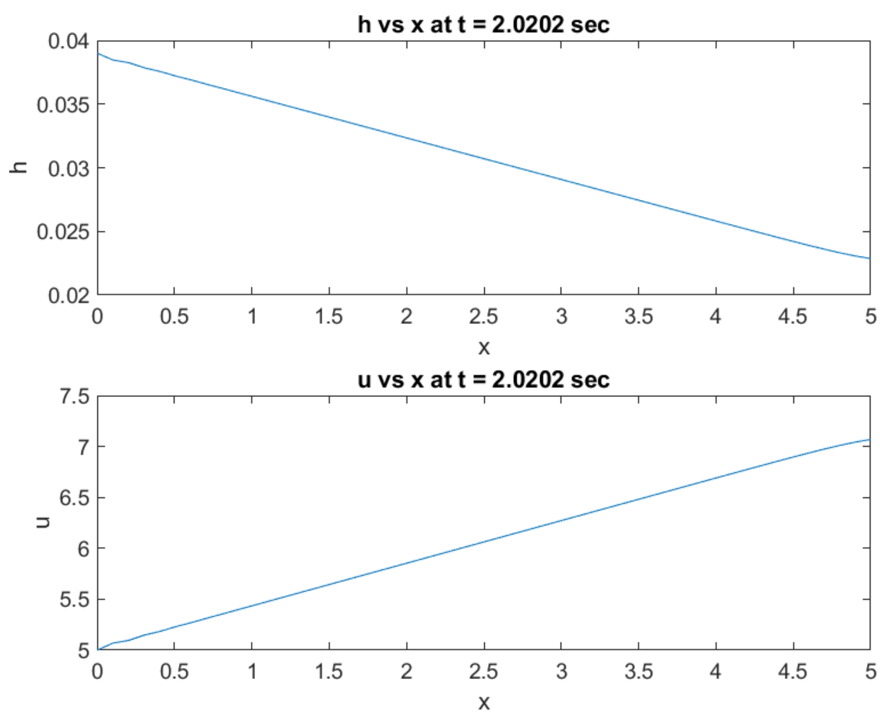

Figure 3: at v=5 and h=39*d

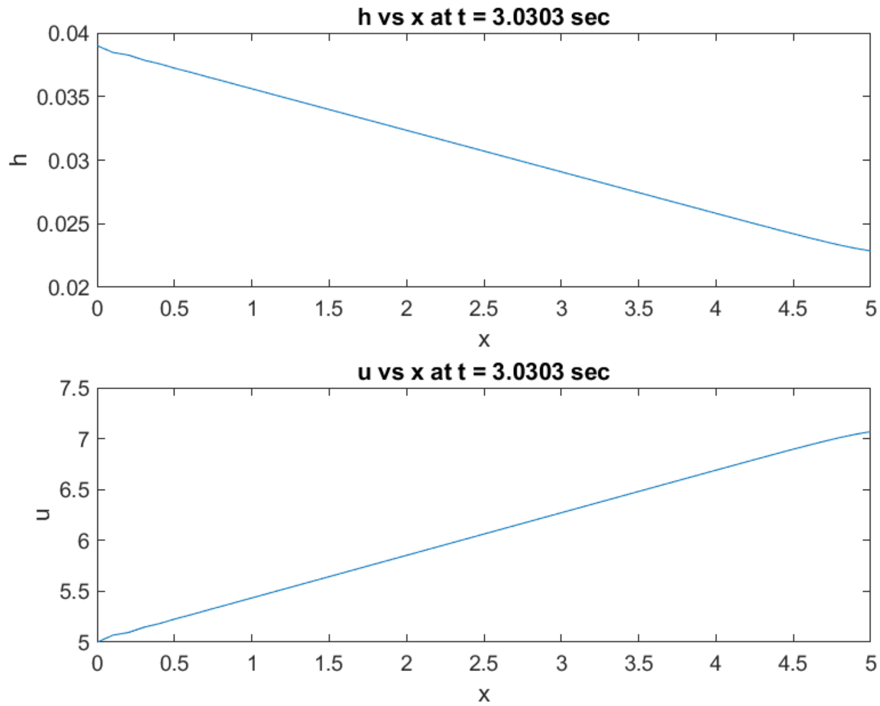

Figure 4: at v=5 and g=39*d

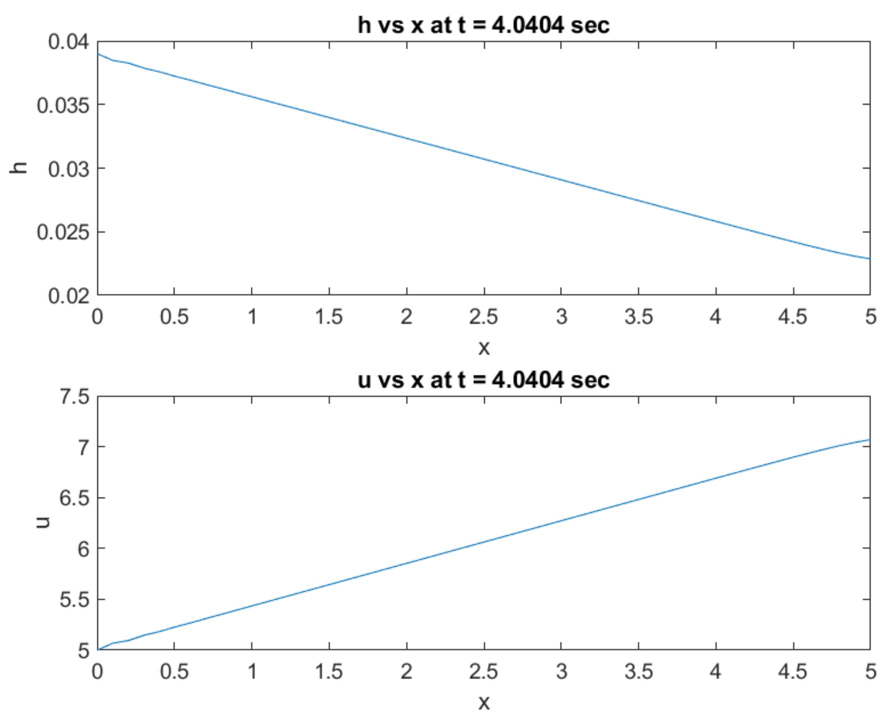

Figure 5: at v=5 and h=39*d

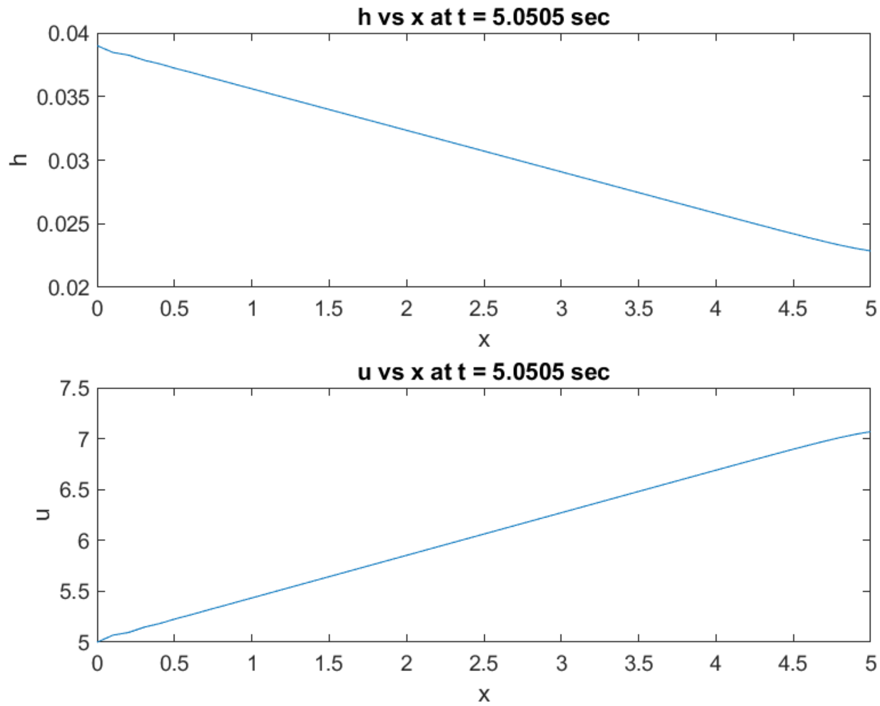

Figure 6: at v=5 and h=39*d

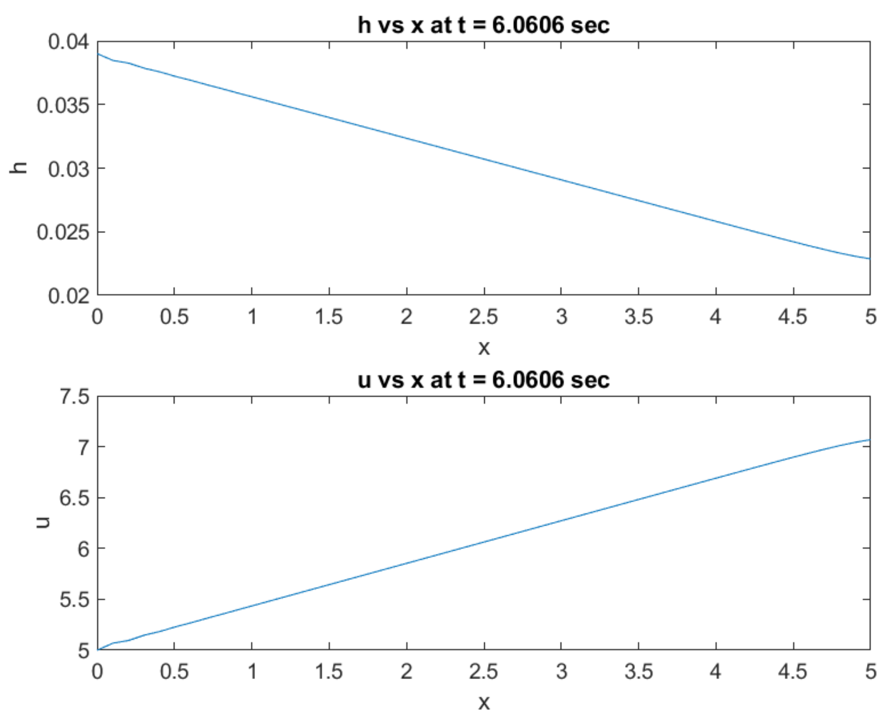

Figure 7: at v=5 and h=39*d

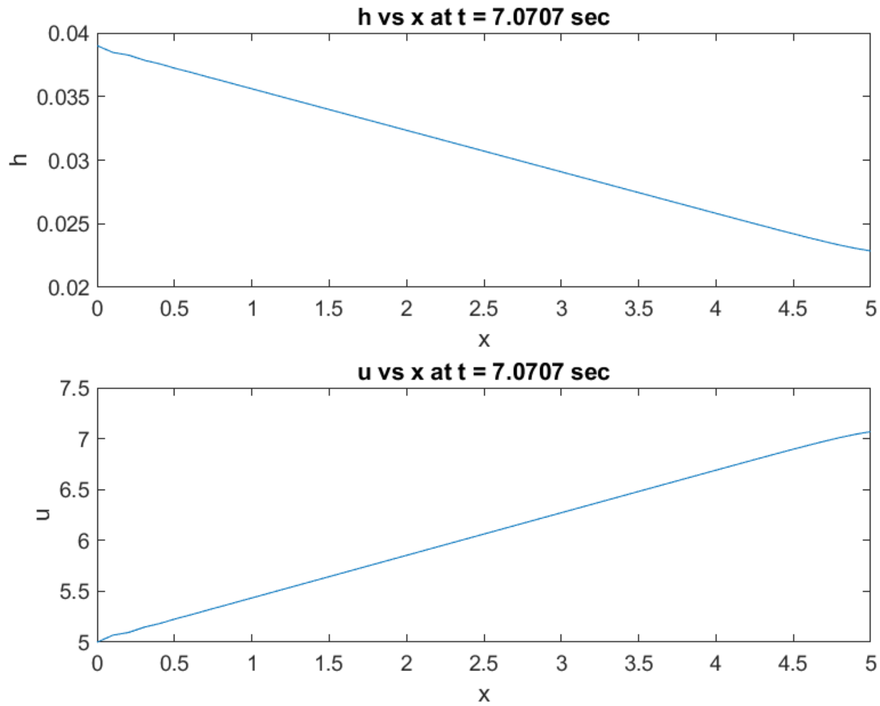

Figure 8: at v=5 and h=39*d

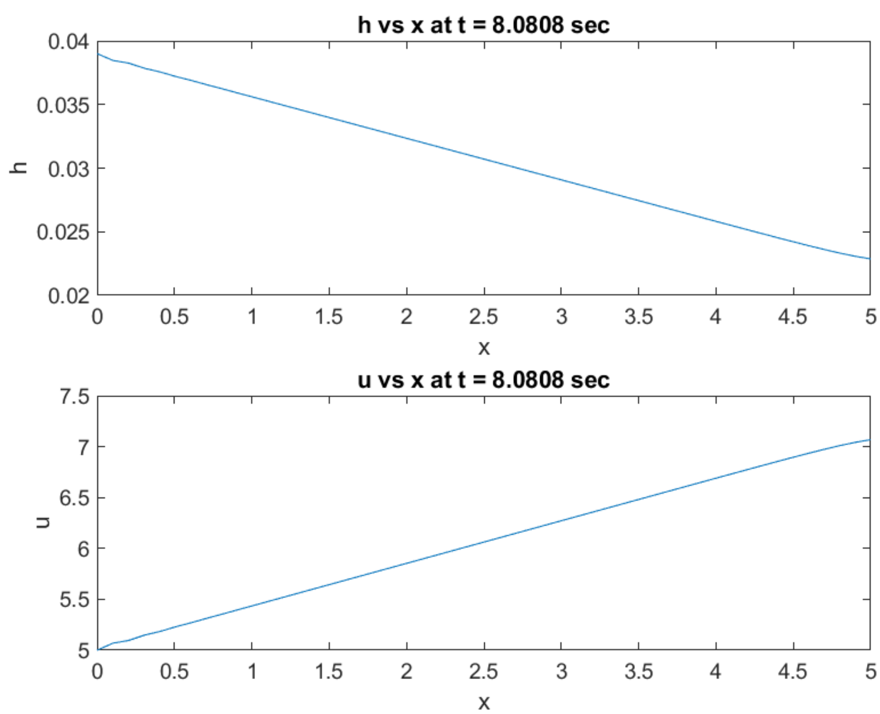

Figure 9: at v=5 and h=39*d

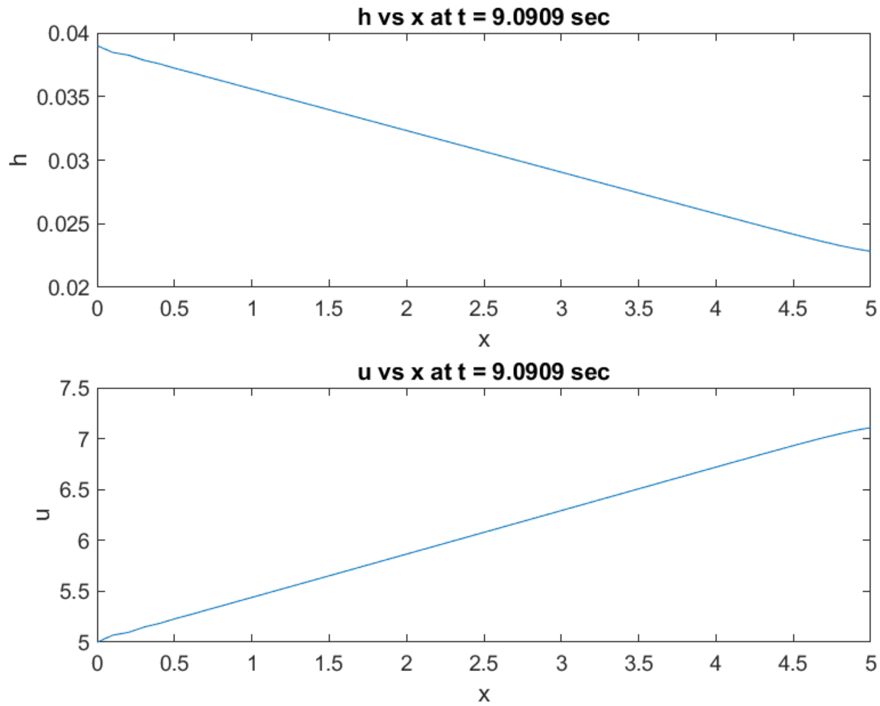

Figure 10: at v=5 and h=39*d

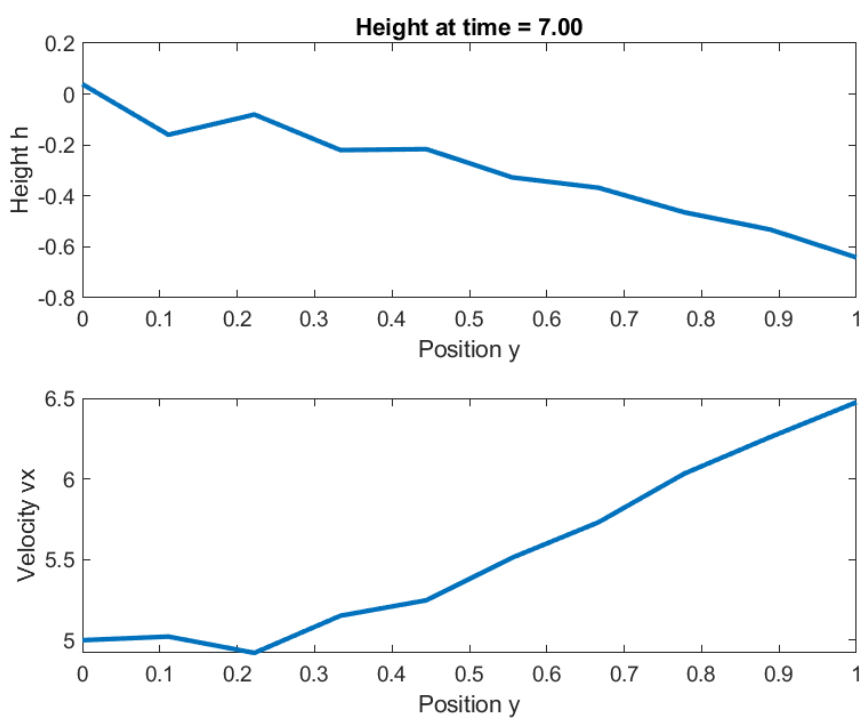

Figure 11: at v=5 and h=39*d PDEPE plots

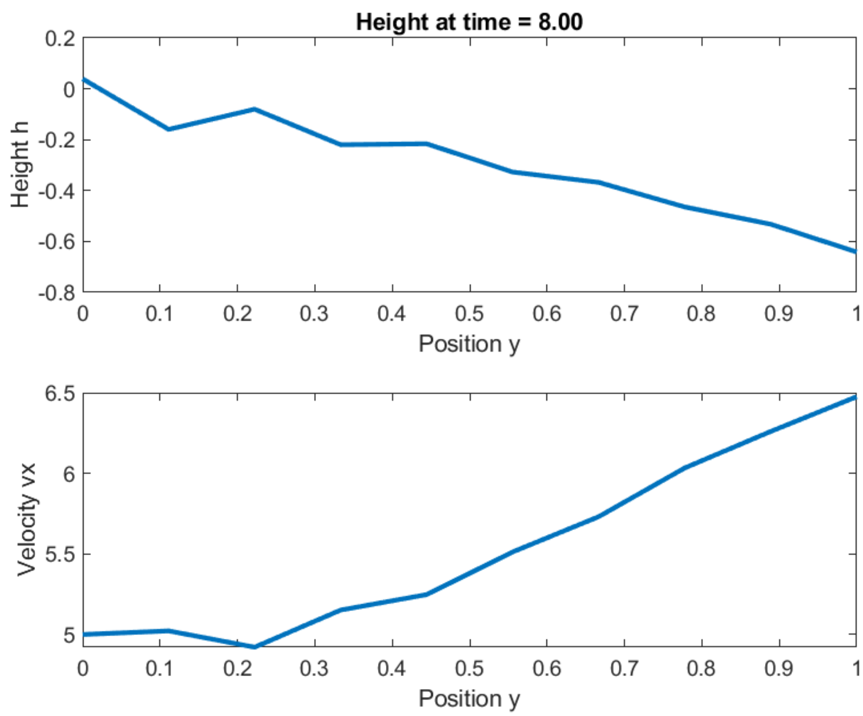

Figure 12: at v=5 and h=39*d PDEPE plots

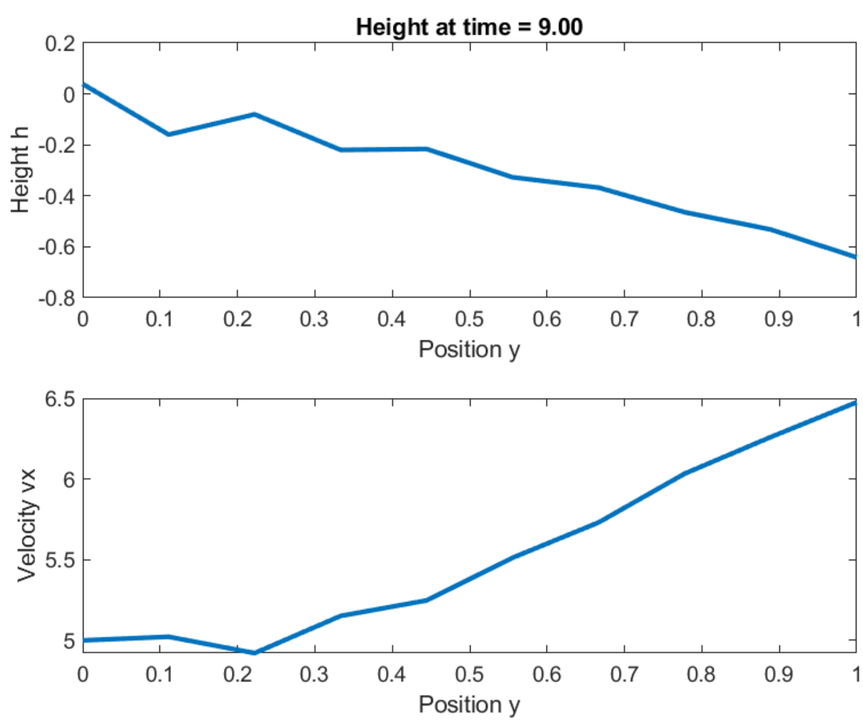

Figure 13: at v=5 and h=39*d PDEPE plots